ISI/SR-78-13
May 1978

ARPA ORDER NO. 2223

PROTECTION ANALYSIS:

Final Report

Richard Bisbey

Dennis Hollingworth

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
ISI/SR-78-13

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*
Protection Analysis: Final Report

5. TYPE OF REPORT & PERIOD COVERED
Research

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR*(s)*
Richard Bisbey II
Dennis Hollingworth

8. CONTRACT OR GRANT NUMBER*(s)*
DAHC 15 72 C 0308

9. PERFORMING ORGANIZATION NAME AND ADDRESS
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA Order #2223

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE
May 1978

13. NUMBER OF PAGES
30

14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*
---

15. SECURITY CLASS. *(of this report)*
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*
This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*
---

18. SUPPLEMENTARY NOTES
---

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
access control, computer security, error analysis, error-driven evaluation, error types, operating system security, protection evaluation, protection policy, software security

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

The Protection Analysis project was initiated at ISI by ARPA IPTO to further understand operating system security vulnerabilities and, where possible, identify automatable techniques for detecting such vulnerabilities in existing system software. The primary goal of the project was to make protection evaluation both more effective and more economical by decomposing it into more manageable and methodical subtasks so as to drastically reduce the requirement for protection expertise and make it as independent as possible of the skills and motivation of the actual individuals involved. The project focused on near-term solutions to the problem of improving the security of existing and future operating systems in an attempt to have some impact on the security of the systems which would be in use over the next ten years.

A general strategy was identified, referred to as "pattern-directed protection evaluation" and tailored to the problem of evaluating existing systems. The approach provided a basis for categorizing protection errors according to their security-relevant properties; it was successfully applied for one such category to the MULTICS operating system, resulting in the detection of previously unknown security vulnerabilities.

ISI/SR-78-13
May 1978

ARPA ORDER NO. 2223

PROTECTION ANALYSIS:

Final Report

Richard Bisbey

Dennis Hollingworth

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223.

VIEWS AND CONCLUSIONS CONTAINED IN THIS STUDY ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF THE UNIVERSITY OF SOUTHERN CALIFORNIA OR ANY OTHER PERSON OR AGENCY CONNECTED WITH IT.

THIS DOCUMENT APPROVED FOR PUBLIC RELEASE AND SALE: DISTRIBUTION IS UNLIMITED.

CONTENTS

Abstract *v*
1. Project Background and Context *1*
2. Project Description *3*
    Collection of Raw Error Data *6*
    Development of Raw Error Patterns *6*
    Development of Generalized Patterns *7*
    Feature Extraction *8*
    Comparison Process *10*
3. Redirection of Research *12*
    Error Categorization *13*
    Analysis of Individual Categories *13*
4. Conclusions and Future Resource Directions *16*
References *18*
Appendix A *19*
Appendix B *21*

ABSTRACT

The Protection Analysis project was initiated at ISI by ARPA IPTO to further understand operating system security vulnerabilities and, where possible, identify automatable techniques for detecting such vulnerabilities in existing system software. The primary goal of the project was to make protection evaluation both more effective and more economical by decomposing it into more manageable and methodical subtasks so as to drastically reduce the requirement for protection expertise and make it as independent as possible of the skills and motivation of the actual individuals involved. The project focused on near-term solutions to the problem of improving the security of existing and future operating systems in an attempt to have some impact on the security of the systems which would be in use over the next ten years.

A general strategy was identified, referred to as "pattern-directed protection evaluation" and tailored to the problem of evaluating existing systems. The approach provided a basis for categorizing protection errors according to their security-relevant properties; it was successfully applied for one such category to the MULTICS operating system, resulting in the detection of previously unknown security vulnerabilities.

1. PROJECT BACKGROUND AND CONTEXT

When general purpose resource-sharing operating systems became available, system customers (both governmental agencies and private firms) naturally wished to exploit fully the economies such systems offered in processing sensitive together with nonsensitive information. Responding to customers' pressure, the systems' manufacturers at first claimed that the hardware and software mechanisms supporting resource sharing would also (with perhaps minor alterations) provide sufficient protection and isolation to permit multiprogramming of sensitive and nonsensitive programs and data. A skeptical technical community challenged this claim and proved it false. Relatively cursory inspection of selected operating systems by "tiger teams" (individuals brought together specifically to attempt to penetrate a target operating system) established that the protection offered fell far short of that required if multiprogramming of sensitive and nonsensitive programs and information were to be permitted [And+71, Bran73]. The protection mechanisms functioned adequately when users exercised prescribed system functions in approximately the prescribed way, but could not resist the system penetrator who looked for unusual or extraordinary means to avoid access checking.

Lacking some of today's insight and knowledge, various manufacturers attempted to retrofit their existing operating systems for security by simply correcting the individual implementation errors and obvious design oversights that contributed to their system's security deficiencies. Critical analysis of these systems, however, established that piecemeal efforts to secure an existing general-purpose operating system were unlikely to succeed [Abb+76, Att+76, BelW74, HolG74, Mcph74].

Out of this early floundering came an appreciation that the security problem was much more difficult to deal with than expected. Furthermore, a number of disturbing issues surfaced:

1. The question of what constituted an appropriate degree of security and how this is determined for a computer system had not been adequately addressed. Indeed, the notion of security was itself difficult to formalize in the context of computer systems, i.e., it was a research issue in its own right. Intuitive statements such as "the system should not allow an unauthorized user to access information he had no right to access" somehow had to be translated into specific assertions about specific operating system objects.

2. No methodology existed for insuring that a given system's design was complete with respect to a particular security policy which might be chosen, i.e., that there were not substantial or significant areas where the desired protection policy could simply be circumvented or ignored.

3. Existing operating systems were poorly structured when it came to security and integrity, usually having grown from early releases to patched, error-ridden monoliths of interconnected code and tables.

4. Efforts to correct known errors were as likely as not to introduce an equal number of new errors, merely manifested in other ways. This became painfully evident during the system penetration activities conducted in conjunction with security retrofit efforts.

5. Program verification techniques would ultimately have to be applied to insure that operating system code functioned correctly and according to specification. However, existing techniques could handle only relatively small pieces of code, limited data types, and relatively simple data structures and data accessing schemes--nothing within an order of magnitude of the size and complexity of an operating system as then structured and implemented.

While these and other issues were troublesome enough with regard to future systems, they were particularly troublesome in light of the large inventory of systems in the DoD and private sector. It had been suggested that an existing operating system would have to be restructured if any substantial improvement in the security afforded was to be effected or if program verification techniques were to be successfully applied. However, restructuring of an existing system (in many cases tantamount to redesign of the system) meant committing substantial resources and rewriting a considerable amount of code. It was also apparent that this could be considered only for a few special systems such as MULTICS and VM/370, which were already well-structured with the access control mechanisms at the innermost level of control.

It became obvious that additional insight into the design and implementation deficiencies responsible for operating system security vulnerabilities was necessary. A much more comprehensive view was required of the number and form taken by such vulnerabilities. The system penetration work performed in the past did little to provide any such collective insight, however; the expertise resulting from such studies consisted of the individual insights of a few individuals rather than communicable ideas and knowledge.

2. PROJECT DESCRIPTION AND ASPIRATIONS

In September of 1973, the Protection Analysis project was initiated at ISI by ARPA IPTO to enhance our understanding of operating system vulnerabilities, expand the rather sparse knowledge base on this subject, and, if possible, identify automatable techniques for detecting vulnerabilities in existing system software. Near-term solutions to the problem of improving the security of existing and future systems were important if operating systems security research was to have much impact on the systems which would be in use over the next ten years. It was hoped that the effort would yield a more formalized knowledge base on operating system security, making it possible to decouple security and operating system expertise to some degree, i.e., to allow individuals having limited expertise in operating system security to effectively detect system vulnerabilities.

The approach adopted was a significant departure from the protection evaluation projects going on elsewhere at that time, such as those at Project RISOS and at System Development Corporation. These efforts to systematize penetration activities dealt primarily with the organization of the project staff itself rather than the discipline applied [Weis73]. They addressed the organizational and training aspects of teams of individuals tasked to analyze operating systems for security vulnerabilities--individuals who themselves would make good "penetrators" of a given target system, who had not only an intimate knowledge of that system but also a good understanding of and feel for protection error possibilities.

It was evident that the success of such groups would depend heavily on individual motivation as well as skill in finding protection errors--an apparent shortcoming when it came to making definitive statements about the validity of the evaluation effort in which such an approach was adopted. The primary goal of the ISI project was to make protection evaluation both more effective and more economical by decomposing it into more manageable and methodical subtasks so as to drastically reduce the requirement for protection expertise and make it as independent as possible of the skills and motivation of the actual individuals involved.

A general strategy was identified which promised to meet these objectives. It included the following five steps:

1. Collection of "raw" error descriptions.

2. Rerepresentation of raw error descriptions in a more formalized notation (producing "raw error patterns").

3. Elimination of superfluous features and abstraction of specific system elements into system-independent elements to develop generalized error patterns.

4. "Normalization" of the target system by extracting the information relevant to the evaluation and representing it in the form required by a "comparison" procedure.

5. Execution of the comparison procedure.

The specific approach adopted--subsequently referred to as "pattern-directed protection evaluation" [Car+75]--was tailored to the problem of evaluating existing systems. It differed from the more general approach principally in that specific features of interest were "extracted" from the operating system source code rather than the entire operating system being rerepresented in a "normalized" format (Figure 1). Thus, steps 4 and 5 changed as follows:

4. "Feature extraction": instantiation of generalized features and searches for instances of these features in the target operating system, and the description of their relevant contexts.

5. Comparison of combinations of feature instances and their contexts with the features and relations expressed in the appropriate error patterns.

A major expectation was that adopting this approach would make it easier to identify previously undiagnosed errors in given operating systems. As superfluous

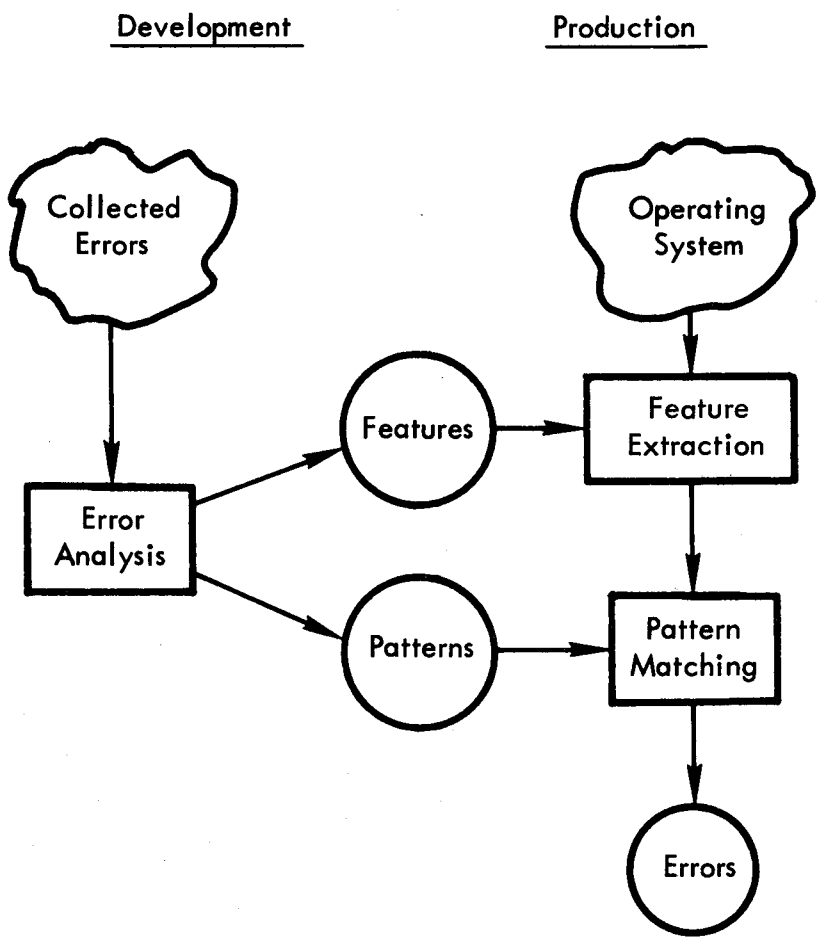

Figure 1. Error-driven evaluation process

features and qualifying details were eliminated and specific system features replaced by more generic or abstract features, a more generalized error representation would evolve. The process could conceivably result in a hierarchy of error patterns, with the most general and abstractly defined patterns at the upper levels and the most specialized and concrete ones at the lower levels. Subsequent instantiation of the generalized patterns by replacing the more general features with their more specific counterparts in particular classes of operating systems or particular functional areas might be expected to reveal previously undiscovered operating system errors (Figure 2).

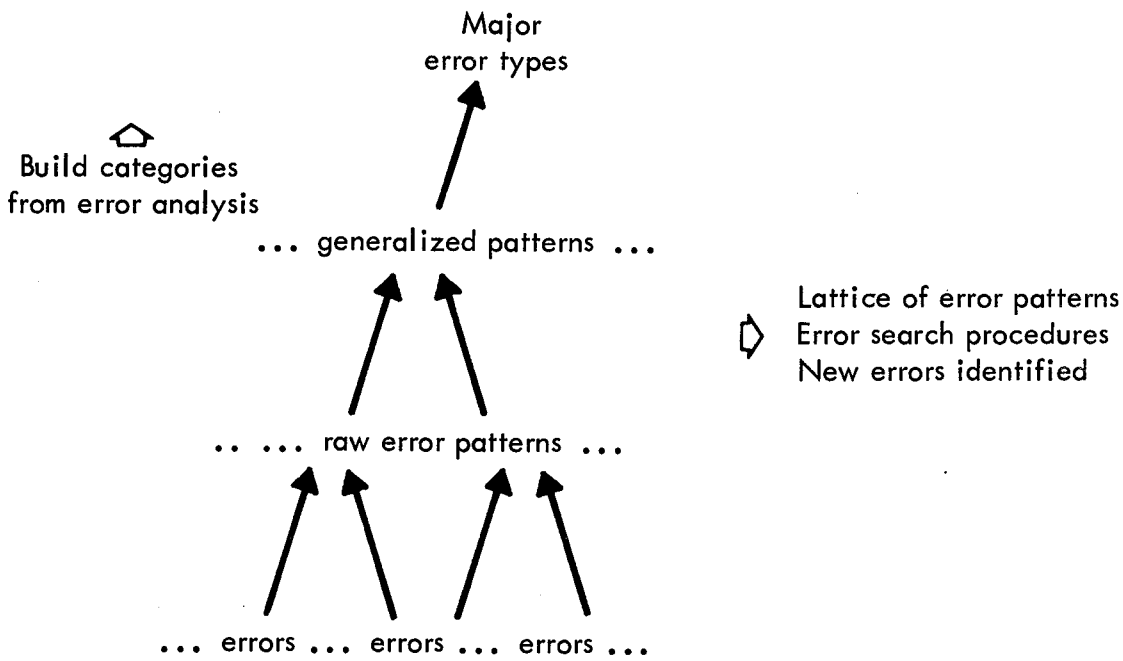

Figure 2

A second expectation was that this approach might result in an empirically sound taxonomy of operating system vulnerabilities and their causes, which would be particularly useful for system designers and implementers. The derivation of raw patterns, their generalization, and the instantiation of generalized patterns toward other systems and functional areas would all add new elements to the lattice of patterns formed by the relation "generalization of" and its converse, "instance of," with the more abstract patterns at the top and the more concrete ones at the bottom. As this structure was enriched with additional patterns, major substructures might emerge, at least below some level of abstractness. If, as was also expected, the search techniques determined to be appropriate for the patterns of each such substructure were also similar, then a reasonable basis would be provided to define major "error types."

The approach was tested with regard to a particular error type frequently found in operating systems, and it proved successful at uncovering previously undiagnosed errors in the MULTICS operating system [Bis+75, Bis+76]. The specific details of the approach and the results and problems which ensued are discussed in the sections which follow.

COLLECTION OF RAW ERROR DATA

Prior to this project, little data on known protection error vulnerabilities had actually been assembled as such in one place. Thus, the first phase of the project involved developing a sufficiently rich collection of data on operating system errors from as many operating systems as possible to provide a good sampling of the types of errors which existed.

Ultimately more than 100 errors that could be employed directly to penetrate existing operating systems were recorded in an error data base; numerous minor variations on these errors were also possible. These errors came from six systems: TENEX, MULTICS, EXEC-8, GCOS, UNIX, and OS/360.

The project staff itself was familiar in varying degrees with five of the six operating systems. They had been directly involved in penetration work on only three of these operating systems, however, and then in projects which examined the systems at widely differing levels of detail. Consequently, the project had to rely to some extent upon information it could gather from outside sources, namely other individuals involved in operating system penetration studies.

Unfortunately, it was difficult to acquire useful data on errors for systems which had not been directly reviewed by the staff. Perhaps the major difficulty was the unavailability of any overall information about operating system vulnerabilities, principally because most installations were reluctant to air weaknesses that might subsequently be exploited by individuals inside as well as outside their organizations. Another significant difficulty also arose whose principal impact was felt in the development of raw error patterns; it is discussed in the following section.

DEVELOPMENT OF RAW ERROR PATTERNS

Given a raw error description, the next step was to formulate an appropriate raw error pattern, a redescription of the error in terms specific to its source operating system but in the form of predicates that express "conditions," properties of or relations among distinct objects or features of that system. During this process those aspects of the initial description superfluous to the actual error itself were eliminated. The "condition set" of a raw pattern was a minimal set of conditions in the sense that if any were removed the raw pattern would no longer represent a potential error.

However, from a particular raw error description, it was often extremely difficult to write down a pattern that satisfactorily captured the essence of the error. First, of course, the error description had to be thoroughly comprehended, e.g., in terms of how the error could be exploited by a knowledgeable penetrator. This required substantial familiarity with and sufficient information on the operating system context in which it occurred. Unfortunately, even where such information was available, the errors were sometimes described in a rather incomplete fashion or in a fashion which presumed substantial knowledge about specific low-level details of the system implementation. This was further complicated by the lack of a common vocabulary for describing both functional elements of the system as well as the particulars of a given security deficiency, requiring some conjecture on the part of the staff as to the exact circumstances of the problem.

Despite these complications, the staff generally was fairly successful in ascertaining what appeared to be the significant characteristics of the error from the available documentation. Even with that, however, it was not always clear precisely what policy was being violated and thus what conditions should constitute the pattern. In some cases, in which equally valid policies could be postulated, the same raw error appeared to lead to more than one pattern.

This process did not appear to be inordinately difficult in the case of the first pattern processed, "*Inconsistency of a Single Data Value over Time.*" The relevant characteristics of such errors were readily apparent, as manifested in the various examples in the error data base. Thus, the textual description of a given instance of the error type was successfully rerepresented in a raw pattern for which superfluous details had been eliminated. This is illustrated by the following raw error description and derived raw error pattern taken from an early version of MULTICS [Bis+75].

Raw Error Description: STOP-PROCESS-ERROR

STOP-PROCESS is a supervisor procedure for halting processes. The user can call the procedure with the process-id of the process to be stopped. The user entry to this procedure checks that the ID is that of the caller, then calls the traffic controller termination routine. The user can modify the value of the process-id between the time it is checked and the time it is passed to the traffic controller.

TOCTOU

Raw Error Pattern:

1. Procedure "STOP-PROCESS" is invoked by a user process to halt a specified process as indicated by a user-supplied parameter.

2. The "STOP-PROCESS" interface checks that the user-supplied process-id parameter is valid.

3. The traffic-controller termination routine uses the process-id to identify the appropriate process.

4. The user process may modify the checked parameter between the times of (2) and (3).

DEVELOPMENT OF GENERALIZED PATTERNS

As an error search criterion, a raw pattern is directly applicable only to operating systems that share the policy violated by that error and in which the features of that pattern are known by the same names. Even then, it may apply only to a particular functional area such as input/output control, and miss similar errors in other areas such as interprocess communication. To broaden the applicability of a pattern, its expression must be generalized by substituting more generic names or more abstract features for more specific ones or by deleting qualifying details without affecting the essence of the conditions themselves. The same concept, such as the call on a privileged system procedure by an unprivileged user procedure, may be known by different names (such as "MME," "JSYS," and "SVC") in different systems. Classes of similar objects, such as bytes or blocks of physical storage, pages, segments, variables, structured variables,

and files (to give an extreme example), can be regarded as instances of a more abstract object, in this case the "abstract cell," something that has a name and holds information (its value). The benefit of generalizing is that the generalized pattern applies to a correspondingly wider class of errors in a wider class of systems.

Generalization of the raw pattern for the inconsistency error examples yielded the following error pattern and corresponding security policy statement:

Generalized Error Pattern:

B:M(X) and for some operation L occurring before M,
[for operation L which does not modify Value(X),
Value(X) before L NOT = Value(X) before M], and
Value(X) after L NOT = Value(X) before M.

Informally stated, process B performs operation M on variable X and the value of X at the time operation M is performed is not equal to the value of X either before or after some operation L which occurs before M.

Corresponding Operating System Security Policy Statement:

(B,M,X) => for some operation L occurring before M, either
[for operation L which does not modify Value(X),
Value(X) before L = Value(X) before M], or
Value(X) after L = Value(X) before M.

Intuitively stated, process B (which presumably performs some critical function) can perform operation M on variable X only if the value of X at the time operation M is performed is equal to the value of X either before or after some operation L which occurs before M.

FEATURE EXTRACTION

Detecting errors in a set of target information implies some kind of comparison process between the target and the correctness or error criteria. The comparison need not be direct; various transformations may be applied, as practical, to either the criteria or the target to bring them into a suitable form, as long as essential properties are preserved. In the case of pattern-directed protection evaluation, the target is a set of operating system source programs and specifications; the criteria are the error patterns; and the comparison process is essentially one of "pattern recognition," in the sense of an ability to detect instances of errors embedded or camouflaged in a system.

Conceptually, the ideal tool is a general-purpose "protection evaluator," a computer program that not only could be applied to a wide class of operating systems but could also reliably detect a wide class of errors. The inputs to such a program would be representations of the patterns for the error types covered, together with a representation of the target operating system. The program would compare the target representation with the given patterns by searching it for all combinations of features related in one of the ways specified in some pattern, and would report every such combination found. In this concept, protection evaluation would seem to consist of two subtasks:

1. "Normalizing" the target system by extracting the information relevant to the evaluation and representing it in the form required by a comparison procedure.

2. Executing the comparison procedure.

Such an ideal is clearly out of reach, however. There exists no model into which the protection-relevant features of an existing system can be mapped and in which they can be related for comparison with given patterns, general enough to apply to wide classes of errors and systems. It is even difficult to determine with precision which elements of existing systems are relevant to protection and which are not.

Nevertheless, the goal of developing pattern-directed techniques and tools to systematize and automate protection evaluation might be achieved with a somewhat altered approach. This becomes evident when one investigates what the two major requirements for protection evaluation techniques imply about their form, application, and development.

The first requirement, that of general-purposeness with respect to operating systems, carries an obvious implication: there must exist some generalized set of terminology--a "comparison language"--in which the techniques are specified and in which the error patterns are expressed. To apply these techniques to a given system, it is necessary that a correspondence be established between the objects and terminology of the comparison language, i.e., between the features of the given patterns and their instantiations in the target system. Either the features of the patterns must be instantiated to the concepts, objects, and terminology of the target system or the target system must be represented in terms of the comparison language, or an intermediate comparison framework must be established and transformations performed in both directions. If no error possibilities are to be overlooked, then all the instances of a given pattern feature in the target system must be identified.

If one uses the term "features" to refer to objects that have concrete and typically localized representations in the target system description (e.g., variables, procedure calls, critical parameters), then identifying the relevant features in the target system is only part of the problem. The other part is to determine whether any of the relations among these features are those indicated by the conditions of an error pattern. The requirement that evaluators need not have a talent for recognizing protection errors and that difficult pattern-recognition processes must not be involved, makes it essential that the search for an error be decomposed. The search through the target system code (or some representation of it) for a single dispersed collection of instances of features in some given relation must be replaced. Instead we must require only independent searches for individual instances of features in the target system. This implies, of course, that the output of these searches must include simple specifications of the contexts in which the feature instances were found. The needed feature context is determined from the relations expressed in the patterns and is used to determine whether the features found actually satisfy these relations. Thus, the single integrated search step is replaced by a two-step procedure, the first of which is more amenable to automation, while the second is probably best performed manually. While the analysis of the relations among features is not avoided, it is deferred to a more convenient point in the process where the feature-set to be considered is greatly reduced in size.

In the case of the inconsistency error, the feature extraction process was applied to a particular instantiation of the error type involving the consistency of user-supplied parameters in the MULTICS operating system. To find instances of the error in code, a pattern was formed using the Error Statement above, which was then instantiated for identifying inconsistent parameter usage. The Error Statement requires the existence of two operations, both of which refer to a common variable X. The first operation, L, either fetches the value of the variable or generates a new value. The second operation, M, fetches the value of the variable. Other information contained in the Error Statement includes the fact that L occurs before M and that M performs some critical function. These statements give rise to the following pattern elements:

1. An operation L which either fetches or stores into a cell X.

2. An operation M which fetches cell X.

3. Operation M is critical.

4. Operation L occurs before operation M.

For this particular error, X is instantiated to a parameter, and thus the following additional pattern element is derived:

5. A procedure B which is interdomain-callable by user procedures and which accepts a parameter X.

This pattern ultimately resulted in the following search procedure intended to recognize, for each parameter, executable sequences of store or fetch operations followed by a fetch operation:

1. Filter out everything except procedures which are interdomain-callable by users.

2. Of these, identify those with parameters.

3. For each parameter, identify and output all instructions or statements which involve store or fetch operations on the parameter.

4. Identify and output all instructions or statements which contain flow of control operators.

This procedure was subsequently automated and applied to MULTICS with significant success, resulting in the detection of a number of candidate errors [Bis+76].

COMPARISON PROCESS

The search output constitutes the input to a separate, methodical comparison process in which the properties of the feature instances found are examined to determine whether actual error conditions exist. Obviously, the comparison is still not direct, since a translation must be made between the generalized relations expressed in the patterns and the descriptions of feature instances provided as input. Again, in

general the choice must be made between expressing the search results in the comparison language and instantiating the reference properties. The former is required for a system- independent comparison algorithm.

In the case of the inconsistency error, that comparison was handled manually. The feature matches were examined manually to determine if the second operation was in fact critical. Forty-seven procedures were examined in the MULTICS system. Of these, seven were observed to have one or more errors; five other procedures had matches for which "criticality" of the second fetch could not be determined due to lack of system documentation.

3. REDIRECTION OF RESEARCH

In September 1975 the research direction was significantly modified to conform to revised schedule and resource considerations. The major problem with the pattern-directed approach (detailed analysis and relating of error characteristic from the bottom-up) was that the process was both time-consuming and extremely tedious; it consumed a substantial amount of the project's resources while yielding few demonstrable results. The sponsor questioned whether or not the protection analysis process was bounded--i.e., whether the number of error categories was both finite and small enough to warrant the expenditure of the resources required. The project was asked to postulate the highest level error categories directly from the existing error data base--to categorize the entries in the error data base in some appropriate fashion based upon the analysis performed to date. We were to subsequently work from the postulated error categories to develop automatable search strategies rather than pursue the pattern-directed approach of gradually building up a set of empirically based categories. It was thought that we might short-circuit some of the more time-consuming elements of the pattern-directed approach, directly identifying an appropriate set of error types without having to devote much effort to analyzing individual errors. The process was expected to be iterative, possibly leading to a set of nonoverlapping error categories which could be precisely defined and which covered the known protection vulnerabilities in existing operating systems and ultimately to viable search techniques for identifying instances of the error categories in target operating systems. Thus, the earlier approach as characterized by Figure 2 was supplanted by that represented in Figure 3 below.

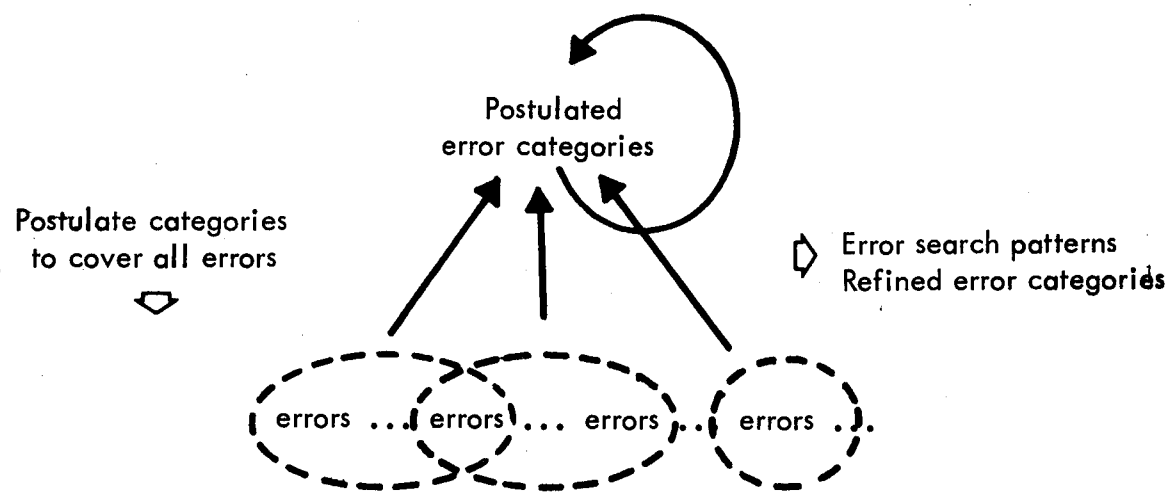

Figure 3

Various difficulties were encountered along the way--unexpected problems which further altered our approach and perspective as to the most appropriate strategy for achieving the original goals. They are mentioned below in the discussion of the specific steps in the revised process.

ERROR CATEGORIZATION

As a consequence of the error-pattern activities the errors collected in the error data-base had already been redescribed in a self-consistent fashion. Thus an attempt was made to directly identify a set of categories which covered the recorded set of protection errors. These categories were to serve the purpose of grouping like error types for in-depth study and analysis. The expectation was that the categories would be refined as the analysis process proceeded until a final set of highly representative, nonintersecting categories was identified.

Ten categories were identified which seemed to cover all the errors which were documented and which did not exclude any known error types. Unfortunately, the ten categories seemed to manifest themselves at differing levels of abstraction; thus, it was assumed that this would not be the final set of categories, that some would be absorbed by more abstract categories or possibly be a basis for new categories when additional analysis had been completed. The categories are briefly described in Appendix A.

ANALYSIS OF INDIVIDUAL CATEGORIES

After an initial set of categories had been identified, attention was directed toward analyzing individual categories to gain additional understanding into the associated operating system security vulnerabilities, allow refinement of the categories, and accommodate the identification of search techniques for given error types. The categories which first received attention were those which appeared to be the most tractable and manifested themselves at the less abstract levels of system object representation. The error type "*Inconsistency of a Single Data Value over Time*," pursued under the pattern-directed work, had been particularly tractable and facilitated identification and implementation of specific tools for identifying errors of this type in existing operating systems. The results of our efforts on that error type suggested that a quite comprehensive semi-automated search could be conducted for such errors in a given operating system. It was hoped that the same would hold true for other error types.

Analysis of the second error category led to a somewhat different result, however. In studying the error category "*Validation of Operands*" it became apparent that the objects under consideration were much less tangible than those dealt with in the "*Inconsistency...*" document. The definition of an operator or operand depended primarily on the level of abstraction on which the operating system was being represented, and the necessary validation was generally at a comparable level [Carl76].

A general strategy was devised for reviewing an operating system for errors of this type, and the requisite tools were identified. However, the analysis of this error type brought into sharp focus the requirement for research in the area of program verification, since the objectives of program verification and the requisite effort in diagnosing errors of this type were quite similar. With this error type it became apparent that the formalization and abstractions that were part and parcel of verifying an operating system were also important in identifying points where validation of critical conditions had not taken place or had been implemented improperly. Determination and analysis of the cumulative effect of conditions and results along relevant control paths as is addressed in the area of program verification is also required in identifying points where incomplete validation has occurred.

The third error type analyzed was that of residuals, i.e., information left over in an object when the object is deallocated from one process and allocated to another. Residuals represented the first error type which had a particularly concrete manifestation in terms of operating system objects (data left undestroyed in a deallocated cell) as well as being a highly intuitive error type. However, it was evident from the outset that the causes of residual errors might well result from other types of errors and that this category might eventually be absorbed by one or more categories handled later on [HolB76]. A strategy for identifying sources of residual errors amenable to partial automation was identified but once again it became apparent that successful identification of the causes of residual errors in operating systems would require sophisticated tools involving symbolic program execution and control flow analysis as well as possibly application of program verification techniques in order to determine the paths and condition sets that might result in bypassing of code intended to clear data cells on deallocation.

The fourth and final error type undertaken was that of serialization. Treatment of this error type launched the project into consideration of the fundamental notions of program structure, operator synchronization, principles of programming practice, etc., and it became quite difficult to identify a viable search strategy. As a side effect, it became immediately evident that the error type "Interrupted Atomic Operations" was a special manifestation of this error category and should be treated in the same context.

A major consequence of work on the aforementioned error types was that it became apparent that the original ten error categories might be reformulated in a more meaningful way in terms of the following four global error categories:

1. Domain Errors

2. Validation Errors

3. Naming Errors

4. Serialization Errors

The remainder of the ten error types (with the exception of the operator selection errors) presented earlier seem either to fall into or split across the four types shown in Table 1.

Of these four categories, two (serialization and validation) were addressed explicitly as a result of the work on the ten originally hypothesized error types; the other two (naming and domain errors) were partially covered through the analysis of one of the remaining error types (allocation/deallocation residual errors). However, the bulk of the examples associated with the latter two categories have not been addressed at any greater detail than was required to group them into their respective categories. Thus, while we believe that the four general categories and their respective subcategories identified represent a useful and representative grouping of example errors and a basis for more directed analysis, it is possible that further study and analysis would result in an even more insightful error classification set.

Appendix B summarizes the four documents produced by the project which address the aforementioned error types.

TABLE 1

Naming Errors	*Validation Errors*
Access Residual Errors	Queue Management/Boundary Errors
Originally Catalogued Naming Errors	Originally Catalogued Validation Errors
Serialization Errors	***Domain Errors***
Multiple Reference Errors	Exposed Representation Errors
Interrupted Atomic Operator Errors	Attribute Residual Errors
	Composition Residual Errors
Originally Catalogued Serialization Errors	Originally Catalogued Domain Errors

4. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In general, the technical community has continually underestimated the difficulty of the security problem; we feel that the PA effort was no exception. It has proved surprisingly difficult to diagnose protection error vulnerabilities, much less design techniques for detecting them. However, while the PA project is terminating at ISI we feel that work might be profitably continued in the original area of pattern-directed protection evaluation despite the inherent difficulties. This approach proved quite successful for the case in which it was taken to completion and we feel that it should prove equally successful in others. Progress occurs at its own rate, however; research of this type is painfully slow. Much thrashing about and some false starts must be allowed for if real progress is to be made in this difficult research area; the desire to produce useful results quickly can be counterproductive to the total effort.

The PA project has had its principal impact in extending the knowledge base and general understanding of operating system protection vulnerabilities, relating apparently unrelated example errors in terms of those common characteristics which result in a security vulnerability. In addition, it has identified some general procedures which will be valuable in detecting future security system vulnerabilities. Finally, the PA project has, along with other efforts, made the user community increasingly aware of the amount of effort and the extensive cost involved in producing a system which has even a remote chance of providing a reasonable degree of security in an open environment. Unfortunately, it has also become apparent that the commercial sector is unwilling to bear this cost at the present time - that there is no apparent commercial market for systems with the development costs, reduced performance and usage and environmental constraints that must be accepted if secure processing is to take place. Consequently, the procedures developed by this project will probably be of little benefit to the commercial sector and of only marginal benefit to the military sector at this time. They will find application only when we decide that the value of data security and personal privacy are greater than the price we must pay for secure data processing.

The analysis of identified error types was particularly useful in identifying some appropriate research and development activities in the area of data security, particularly with respect to the types of tools required if protection evaluation is to become automatable. Tools of the sort described in the "*Data Dependency Analysis*" document will be needed in much of the evaluation activity, but might be constructed so as to be generalizable across systems and programming languages.

During the research effort one thing that became evident was the role of program verification techniques in detecting operating system security vulnerabilities. It is hard to see how truly definitive statements about the security afforded by an operating system can ever be made until PV techniques have been applied. However, certain unsettled issues about the appropriate application of PV techniques to O.S. security analysis suggest that research in protection evaluation might be profitably continued in parallel with research in PV, principally to insure that PV is applied at appropriate levels of operating system representation, that mapping between levels is handled properly, and that the operating system is represented in sufficient detail to insure that security vulnerabilities do not go undetected.

As a final footnote to this research effort we offer the following comment for those who are optimistic about near-term improvement of the data security problem. Our insight into and awareness of security vulnerabilities has tended to vastly exceed our progress in detecting and correcting them. There are still difficult research problems to be attacked in the area of PE in particular and data security research in general. In the course of addressing these research problems there will undoubtedly be much floundering and some abortive starts. Progress can be expected to be painful and slow in final disposition of the security problem, particularly since such work seems to involve delving into the basic premises of programming theory and practice.

REFERENCES

Abb+76 Abbott, R. P. et al., *Security Analysis and Enhancements of Computer Operating Systems*, National Bureau of Standards Institute for Computer Sciences and Technology, NBSIR 76-1041, April 1976.

And+71 Anderson, J. P., R. L. Bisbey, D. Hollingworth, and K. W. Uncapher, *Computer Security Experiment* (U)., The Rand Corporation, WN-7275-ARPA, March 1971 (Secret).

Att+76 Attanasio, C. R., P. W. Markstein, and R. J Phillips,"Penetrating an Operating System: A Study of VM/370 Integrity," *IBM Systems Journal*, 15, January 1976, pp. 102-116

BelW74 Belady, L. A., and C. Weissman, "Experiments with Secure Resource Sharing for Virtual Machines," *Proceedings of the International Workshop on Protection in Operating Systems*, August 1974, pp. 27-33.

Bis+75 Bisbey, Richard, II, G. Popek, and J. Carlstedt, *Protection Errors in Operating Systems: Inconsistency of a Single Data Value Over Time*, Information Sciences Institute, ISI/SR-75-4, December 1975.

Bis+76 Bisbey, Richard, II et al., *Data Dependency Analysis*, Information Sciences Institute, ISI/RR-76-45, February 1976.

Bran73 Branstad, D., "Privacy and Protection in Operating Systems," *Computer*, January 1973.

Car+75 Carlstedt, J. et al., *Pattern Directed Protection Evaluation*, Information Sciences Institute, ISI/RR-75-31, June 1975.

Carl76 Carlstedt, J., *Protection Errors in Operating Systems: Validation of Critical Conditions*, Information Sciences Institute, ISI/SR-76-5 , May 1976.

Carl78a Carlstedt, J., *Protection Errors in Operating Systems: A Selected Annotated Bibliography and Index to Terminology*, Information Sciences Institute, ISI/SR-78-10, January 1978.

Carl78b Carlstedt, J., *Protection Errors in Operating Systems: Serialization*, Information Sciences Institute, ISI/SR-78-9, April 1978.

HolB76 Hollingworth, D. and R. Bisbey II, *Protection Errors in Operating Systems: Allocation/Deallocation Residuals*, Information Sciences Institute, ISI/SR-76-7, June 1976.

HolG74 Hollingworth, D. and S. Glasman, *WWMCCS/GCOS III: Security Analysis of Master Mode Entry Processing*, The Rand Corporation, WN(L)-8749-DCA, July 1974.

Mcph74 McPhee, W. S., "Operating System Integrity in OS/VS2," *IBM Systems Journal*, 13, 1974, pp. 230-252.

Weis73 Weissman, C., *System Security Analysis/Certification Methodology and Results*, System Development Corporation, SP-3728, October 1973.

APPENDIX A

1. Consistency of data over time

Operating systems continuously make protection-related decisions based on data values contained within the system data base as well as on values which have been submitted to and validated by the system.

In order for a correct protection decision to be made (in the absence of other types of protection errors), the data must be in a consistent state, and remain in a specific relationship with other data items during the interval in which the protection decision is made and the corresponding action taken.

2. Validation of operands

Within an operating system, numerous operators are responsible for maintaining the system's data base and for changing the protection state of processes or objects known to the system. Many of these operators are critical in the sense that if invalid or unconstrained data are presented to them, a protection error results.

3. Residuals

A generally accepted error type is that of the "residual," i.e., information which is "left over" in an object when the object is deallocated from one process and allocated to another. Several types of residual errors exist, including the following:

1. Access residuals: Incomplete revocation or deallocation of the access capabilities to the object or cell.

2. Composition residuals: Incomplete destruction of the cell's context with other cells or objects.

3. Data residuals: Incomplete destruction of old values within the cell.

4. Naming

Names are used within operating systems to distinguish objects from one another. There are many ways in which name binding errors can lead to protection errors. For example, often the naming scheme does not have enough resolution (or does not use that resolution) to distinguish properly between named objects. This results in those errors typified by a user creating an ambiguity by naming objects with the same name as a previously named (or about to be named) object with the system, as a result, referencing the wrong object.

5. Domain

A domain is an authority specification over an object or set of objects (usually thought of in terms of an address space). Enforcement of domains is typically limited to the resolution of the hardware protection mechanism provided by the computer. Many

of the errors in operating systems are the direct result of one of two types of domain-related errors:

1. Information associated with the wrong domain.

2. Incorrect enforcement at domain crossing.

6. *Serialization*

Within any operating system, there are resources to which the operating system must not only control access, but also prevent concurrent use or otherwise enforce orderly use. This problem, known as "serialization," is of particular importance in multiprogramming systems where serialization errors often result in protection errors.

7. *Interrupted Atomic Operations*

Several protection errors have appeared in which the enforcement of a protection policy was based on the assumed uninterruptability of an operation. In each of the cases, the operation was in fact interruptable, resulting in a protection error.

8. *Exposed Representations*

To each user, an operating system presents an abstract machine consisting of the hardware user instruction set plus the pseudo-instructions provided through the supervisor call/invocation mechanism. The pseudo-instructions, in general, allow the user to manipulate abstract objects for which representations and operations are not provided in the basic hardware instruction set. Inadvertent exposure by the system of the representation of the abstract object, the primitive instructions which implement the pseudo-instructions or the data structures involved in the manipulation of the abstract object can sometimes result in protected information being made accessible to the user, thereby resulting in a protection error.

9. *Queue Management Dependencies*

This error type broadly includes those errors characterized by improper or incomplete handling of boundary conditions in manipulating data structures such as system queues or tables. The consequence is generally a system crash or lockup resulting in gross denial of service. We distinguish this from legitimate denial of service conditions when the system is merely overloaded, but still functioning according to the scheduling algorithm design specifications.

10. *Critical Operator Selection Errors*

This error type includes those errors in which the implementer invoked the wrong function, statement, or instruction resulting in the program performing the wrong function. In a sense, this is a catch-all category, since every programming error can ultimately be so classified.

APPENDIX B

The purpose of this appendix is to provide a context for reading the respective error detection papers.

Inconsistency of a single data value

A common error in contemporary operating systems is the assumed consistency of operands between multiple uses. If an operand can be modified between two uses by a program and the second use relies on an attribute referenced in or set by the first usage, an error results. Multiple usage of a single operand often occurs during validation/use sequences where an operand is first validated and subsequently used in a computation. Numerous variations exist that make locating instances of the error difficult. For example, the operand can be referred to by different names, or the uses may be contained in textually disjoint routines.

Two patterns for finding inconsistency errors are as follows:

1a. Find any sequence of REFERENCE ... REFERENCE to a common operand,
or
1b. Find any sequence of STORE ... REFERENCE to a common operand,

whenever

2. the operand can be modified between the pair of operators.

Detection of Inconsistency Errors. Outlined below is a set of search strategies for finding consistency errors based on detecting possible instances of condition 1a or 1b. Large portions can be automated.

Consider the possible storage classes that operand A can take with respect to the routine containing the two references. They are limited to one of the following three:

1. A local
2. A parameter
3. A global

Case 1: Local Operand

If the operand is local (in the sense that no other routine can access it), then the error cannot occur and, thus, no search technique is needed.

Case 2: Parameter Operand

If the operand is a value parameter, then, since it is copied at invocation time into a local variable within the routine in question, it can be treated as a local operand as in Case 1. If the operand is a name or reference parameter, the following search strategy applies:

1. For each parameter within a routine, find all reference and store instructions to the parameter.

2. For the routine, find all control flow operators.

3. For any REFERENCE ... REFERENCE or STORE ... REFERENCE on a control path (determined by the control flow operators found in 2), examine the pair to determine if the second reference operation relies on an attribute referenced or stored by the first operator.

4. For any control path that allows a single REFERENCE to be executed iteratively, determine if the second execution of the REFERENCE relies on an attribute referenced by the first execution.

The above procedure finds all possible occurrences of the error for parameter operands. Steps 1 and 2 can easily be implemented by computer program.

Case 3: Global Operand

If the operand is a global, then it can be accessed by multiple routines. The following search strategy applies:

1. For each global, find all reference and store instructions to the global.

2. Find all the control flow operators.

3. For any REFERENCE ... REFERENCE or STORE ... REFERENCE on a control path examine the pair to determine if the second reference operation relies on an attribute referenced or stored by the first.

4. For any control path that allows a single REFERENCE to be executed iteratively or recursively, determine if the second execution of the REFERENCE relies on an attribute referenced by the first execution.

Note that, with one exception, this is the same search strategy used for parameters. The difference is that, for globals, multiple execution of a single instruction can also result from recursion. Otherwise, the procedure is identical, and in fact the same code used to detect potential inconsistency errors for parameters can also be used to detect potential inconsistency errors for globals.

The above search strategies find all possible consistency errors. A more detailed description of Inconsistency Errors can be found in Bis+75.

Validation

Validation of operands is one of the more basic functions performed in operating systems; it constitutes one of the more basic error types. Validation can take a variety of forms, from checking that an integer subscript is within the bounds before allowing an array access operator to proceed, to checking that a set of properties such as the time-of-day and the caller's access rights hold for an operation to be performed. No single evaluation approach seems adequate to deal with the wide variety of validation found in contemporary systems and information a protection evaluator may have available for performing the evaluation task. As such, two approaches for finding validation errors have been identified. The protection evaluator may choose either or a combination of both.

The first requires the protection evaluator to be able to recognize an invalid condition for an operand. It begins with the sources of data needing validation, finds the operators which use such data (i.e., those which are potential candidates for validation errors), and computes the validation condition holding for a given operator/operand. A protection evaluator must then judge the adequacy of the validity condition for the given operator. The second approach begins with operators and validation conditions which must hold and determines if the conditions are actually enforced by the code. It requires the evaluator to be able to identify all critical operators and specify their associated validation conditions before proceeding with the evaluation.

Outside-to-Inside Approach. A purpose of validation is to prevent privileged system operators from operating on incorrect/unvalidated operands. Externally-supplied user data constitutes such a source. They enter the system in a variety of ways. Direct or indirect parameters to supervisor subroutines constitute one large source. Others include mutually agreed upon mail boxes, communications areas, or files. The operating system is responsible for insuring that this data is properly checked before a system operator uses it.

One approach for determining the adequacy of validation is to begin at the user/system interface and calculate the validity conditions for all user-supplied data at various operators within the system. This can be done as follows:

1. Identify all data entry points into the system. (At all such points, data can enter the system that needs to be validated.)

2. For each data entry point, calculate data flow paths through the system. All operating system variables to which the entering data is directly or indirectly assigned must be recorded.

3. Examine all operators referencing a variable identified in (2) above. Verify that the validity condition enforced on each data path leading to that operator/operand is sufficient.

Step 2 can be automated using data dependency analysis or a modified form of symbolic execution. Steps 1 and 3 must be done manually. It is important to note that without detailed semantic information describing operations being performed, any procedure, such as the above, can only tell an evaluator where to look for errors, but not what to look for.

Inside-to-Outside Approach. Suppose a protection evaluator can identify all critical operators in the system and can specify for each operator the validity condition that must hold for the successful completion of that operator. The problem of finding validation errors then amounts to determining the sufficiency of validation code on all paths leading to that operator. A procedure for checking sufficiency would be as follows:

1. Identify the critical operations within the operating system and the necessary conditions associated with those operations. Record the condition with the associated operand.

2. If an operand is a local or a parameter, follow all possible control paths leading from the operation to determine the data paths leading to the critical operation. In passing in a reverse direction through code that enforces

portions of the validation condition, discard the enforced condition. Eventually, one of the following will occur:

a. All conditions are enforced for that control path.

b. All conditions are not enforced upon reaching a user/system interface, i.e., a validation error can be caused by supplying a value outside the range of the remaining unenforced condition.

c. The control path terminates at a global variable/parameter interface within the system. Go to 3.

3. If the operand is a global or formal parameter from 2c, all operators modifying the global/parameter must contain as an output condition the validity condition associated with the respective variables. They become critical operators to be evaluated by this same algorithm.

A more detailed description of validation errors can be found in Carl76.

Residuals

A common security problem is the residual--data or access capability left after the completion of a process and not intended for use outside the context of that process. If a residual becomes accessible to another process, a security error may result. A major source of such residuals is improper or incomplete allocation/deallocation processing.

Probably the most widely recognized type of residual is the data residual in which some property of the data associated with a cell is not disposed of upon reallocation. One typically thinks of content residuals, i.e., residuals where the cell content is retained after reallocation. Data residuals can, however, involve other cell attributes. Such attributes can include cell size, cell location, and the physical relationship of the cell to other cells. While not representing as high a communications bandwidth as the content residuals, these latter forms of data residual can also represent significant security errors.

The following procedure for finding data residuals is based on identifying the cell allocation/deallocation routine in which residual prevention code should be contained. It consists of four basic steps:

1. Identify all cell types found in the system. This can be done by manually listing various storage media and cells on that media and by examining system data declarations.

2. For each cell, identify its particular freepool, i.e., the buffers for cell resources between deallocation and allocation.

3. For each freepool, identify allocation/deallocation code by finding all symbolic references to the freepool.

4. For each allocation/deallocation routine, determine if a data residual can occur.

A second major type of residual is the access management residual, sometimes known as a "dangling reference." Unlike data residuals that deal with the various attributes of a cell, access management residuals deal with the access paths used to reference a cell, their creation and destruction.

Access paths are, at some level of representation, simply data stored in special cells (e.g., bounds registers, PSW's, segment/page tables, capability cells, etc.). Thus, techniques similar to those described above for finding content residuals will also find certain types of access residuals, i.e., those caused by incomplete deallocation of an access path created by an allocation routine. Access management residuals differ from content residuals in an important aspect. There may be multiple access paths to a given cell, all of which must be deallocated. Furthermore, access paths can be created by other than the formal allocation routines. For example, code that copies an existing access path produces an access path which must also be accounted for at deallocation. Similarly, special instructions may exist (e.g., the IBM 370 "LOAD-REAL-ADDRESS") that produce access paths as a result of invocation, or that can be interrupted causing an access path to be stored for use when the instruction is reinvoked. Thus, in addition to the above procedure, one must examine the system for these latter three sources of access paths and account for the paths at cell deallocation.

A more detailed description of Residual errors can be found in HolB76.

Serialization

Serialization errors represent one of the broader categories investigated. As such, the error has numerous manifestations and can be described in a variety of ways including ordering specifications; interoperation communication and insuring the proper use of communication channels; mutual exclusion for preserving object integrity; and mutual exclusion for the noninterference of non-atomic operations.

Three distinct approaches for detecting serialization errors are:

1. Analyze the target system macroscopically and informally for the adequacy of each of a list of serialization provisions. The problem with this approach is that no actual algorithm is suggested by the serialization provisions for deciding when serialization errors do or do not exist.

2. Determine potential concurrencies, and, given these, determine whether any of them (taken pairwise) represent access conflicts.

3. Assume all access sequences to sharable objects are critical and represent potentially conflicting concurrencies unless these are made impossible either by explicit invocations of serialization mechanisms or by other serializing program logic. The problem with this approach is that it detects a great many access intervals that are not serialized in an obvious manner, and one must then resort to deeper analysis such as that in (2).

Each approach is discussed in greater detail along with suggested ways for alleviating deficiencies in Carl 78.