PATTERN-DIRECTED PROTECTION VALUATION

Jim Carlstedt, et al

University of Southern California

Prepared for:

Defense Advanced Research Projects Agency

June 1975

ARPA ORDER NO. 2223

ISI/RR-75-31
June 1975

Jim Carlstedt
Richard Bisbey II
Gerald Popek

Pattern-Directed Protection Evaluation

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/RR-75-31	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Pattern-Directed Protection Evaluation		5. TYPE OF REPORT & PERIOD COVERED Research 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jim Carlstedt, Richard Bisbey II, Gerald Popek		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 Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE June 1975 13. NUMBER OF PAGES 24
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 approved for public release and sale; distribution 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)

computer security, debugging, error patterns, operating systems, protection, protection evaluation, software security.

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

Because of the urgent security requirements in many existing general-purpose operating systems, the large investment committed to such systems, and the large number of protection errors embedded in them, the problem of finding such errors is one of major importance. This report presents an approach to this task, based on the premise that the effectiveness of error searches can be greatly increased by techniques that utilize "patterns", i.e., formalized descriptions of error types. It gives a conceptual overview of the pattern-directed evaluation process and reports the authors' initial experience in formulating patterns from the analysis of protection errors previously detected in various systems, as well as in applying the pattern-directed technique.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

PRICES SUBJECT TO CHANGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ARPA ORDER NO. 2223

ISI/RR-75-31
June 1975

Jim Carlstedt
Richard Bisbey II
Gerald Popek

Pattern-Directed Protection Evaluation

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, PROGRAM CODE NO. 3D30 AND 3P10

CONTENTS

Preface v

1. Introduction 1

2. Basic Considerations 3

3. Pattern Development 5

4. Development and Application of Pattern-Directed Techniques 9

5. An Example 13

6. Summary 16

Acknowledgments 17

References 19

PREFACE

Because of the urgent security requirements in many existing general-purpose operating systems, the large investment committed to such systems, and the large number of protection errors embedded in them, the problem of finding such errors is one of major importance. This report presents an approach to this task, based on the premise that the effectiveness of error searches can be greatly increased by techniques that utilize "patterns," i.e., formalized descriptions of error types. It gives a conceptual overview of the pattern-directed evaluation process and reports the authors' initial experience in formulating patterns from the analysis of protection errors previously detected in various systems, as well as in applying the pattern-directed technique. This study is part of a larger effort to provide securable operating systems in DoD environments.

I. INTRODUCTION

This report deals with the problem of improving the security of existing generalized resource-sharing operating systems by finding errors in the protection mechanisms of those systems. This task has come to be called "protection evaluation" (PE). The PE problem is of obvious importance in view of the investment existing systems represent, their expected lifetime, and their insecurity. It is well known that current general-purpose operating systems, which are large and complex, usually contain a large number and variety of errors even after having been in service for years. That these include security errors is indicated by the fact that skillful penetration efforts directed against these systems invariably succeed. The task of improving such systems is urgent, since many of them are installed in environments--governmental, commercial, and military--in which the requirement for security (in terms of the magnitude of losses from accidental or intentional violations) is strong and immediate. These losses will be reduced in proportion to the cost-effectiveness of the available error-finding tools.

In economic terms, debugging is and always has been one of the most important problems in the computer field, and considerable effort has been spent in attempts to reduce it. The results have been less than spectacular; there remains a wide gap between the effectiveness of the most sophisticated currently available debugging tools and the eventual ability to prove the correctness of properly structured operating systems. The approach presented here is directed at narrowing that gap. The goal is to provide a basis for the development of tools (for at least the PE application area) that are significantly more effective than currently existing tools but that can also be made available in the relatively short-term future. The proposed approach is more formal than traditional debugging methods but less formal than logical/mathematical verification. It also restricts itself to static representations of operating systems--listings, accompanying documentation, and information derived from them (including the output of compilers and loaders)--and is intended to complement dynamic methods such as testing and monitoring in the common situation where on-line access to the target system by evaluators is not readily available. (It should be noted that in some cases flaws detected by static methods must be further analyzed with respect to dynamic operations to determine whether they represent actual errors.)

Typically, PE is carried out manually, using only simple debugging tools and rudimentary aids such as symbol concordances. Usually PE projects, such as those of Project RISOS at Lawrence Livermore Laboratory* or at System Development Corporation [Wei73], are organized around teams of individuals, and their success depends heavily on the motivation of these individuals as well as on their skill in finding protection errors. In other words, they require individuals who would themselves make good "penetrators" of the target system. This means they must have not only an intimate knowledge of that system but also a good understanding of or feel for protection error possibilities. Efforts to systematize this work have dealt primarily with the organization of the project staff itself.

The goal of the approach presented here, in contrast, is to make PE both more effective and more economical by decomposing it into more manageable and more methodical subtasks, in such a way as to drastically reduce the requirement for protection expertise. The approach is called "pattern-directed" because it is based on an analysis of how formalized error patterns may be used to direct and systematize the PE task.

Section 2 states the basic observations from which the approach is derived and the requirements that shape the techniques and tools to be developed. Section 3 describes the formulation and classification of patterns together with some initial experience. Section 4 examines the application of pattern-directed techniques in light of the requirements stated earlier, and draws conclusions about the form such techniques must take. Section 5 gives an example of the application of a pattern-directed technique to a particular type of error and relates the results of a preliminary test of the feasibility of the pattern-directed approach.

---

*Private communication with Robert P. Abbott.

2. BASIC CONSIDERATIONS

The pattern-directed approach to PE is based on two observations:

1. Protection errors of the same or similar types appear not only in different functional areas of the same model of an operating system, but also in different systems. Furthermore, there is reason to believe that the number of types of basic vulnerabilities is fairly small. Some authors have speculated that the number is less than ten [And72, McP74], although the types they list do not entirely correspond. While the definition of "error type" is open to question (a reasonable interpretation is suggested in Section 3), we have already identified additional types. We believe, however, that the number of basic types is less than 25.

2. The effectiveness of a search depends, in part, on the degree to which the object or type of objects being searched for are well-described or well-defined. In the case of protection errors, we have experienced and witnessed in others the large difference in effectiveness between a "blind search" and an examination directed toward errors of a particular type described by a concise pattern. A typical example is that of the protection error continually overlooked--even though textually adjacent to an error found weeks or months earlier--until noticed as an instance of a given pattern.* We have found that even persons with no previous experience in protection evaluation can find errors when given a specific pattern to guide their search.

An approach suggested by these observations is to (1) identify the basic error types and formulate the patterns representing them, and (2) develop search techniques capitalizing on these patterns. These basic activities are described in the next two sections.

Two important requirements must guide the development of pattern-directed techniques:

---

*In a case with which one of us (Bisbey) is familiar, an error was discovered just three instructions away from one which had been previously corrected.

1. They must be widely applicable, which implies that to a large extent they must be general-purpose with respect to operating systems. Little is gained over current methods if completely separate techniques are necessary to evaluate each new system of different manufacturer or version.

2. If these techniques are to be significantly more effective, economical, and reliable than existing techniques, evaluators who use them must not be required to possess particular expertise in protection, nor to develop any deep understanding of protection errors, nor to be able to recognize them as such in the dispersed or camouflaged form in which they frequently exist. In the sections that follow, the word "evaluator" will be assumed to denote such a nonexpert. The use of these techniques must not require evaluators to perform pattern recognition activities nearly as difficult as those currently required to find protection errors. An evaluator will, of course, be assumed to be familiar with the internals of the system being evaluated.

The effects of these requirements are discussed in Section 4.

3. PATTERN DEVELOPMENT

There are two alternative strategies for deriving error patterns: either deduce them from theoretical considerations or infer them from an analysis of errors that have already been detected during PE of existing systems. The latter, empirical approach has been adopted because it appears to offer a greater assurance of success in less time, because a substantial number of such errors already awaits collection and analysis, and because we believe a methodical collection and analysis of such errors is a valuable undertaking in its own right (e.g., to develop a manual of "good design practices").

The material from which patterns are derived are "raw errors," descriptions of security errors found in various operating systems, usually expressed very informally and in terms specific to the particular systems in which they were found. Our collection currently contains raw errors from the OS/360, GCOS, Multics, TENEX, and Exec-8 systems. The following is an example of a raw error, exactly as collected:

> "Snap Dump is a supervisor routine for providing printed core dumps of memory. The routine consists of nine nonresident modules, each of which is separately fetched and executed, and one resident module which remains in main storage for the entire dump process. The resident module (IEAQAD0A) is loaded by the first segment of Snap Dump and contains several format and output subroutines used by the other modules. The error is that if a user names his program IEAQAD0A, his program will be given control in privileged mode, instead of the system program of the same name."

A more precise representation is needed than the unconstrained narrative in which errors are first obtained. The formulation of patterns should facilitate both their classification and their application. This implies that patterns should be complete and concise representations of errors, cast in a standardized form and notation.

With this in mind, we regard a pattern first of all as a set of independent "conditions," predicates that express properties of or relations among distinct objects or "features" that can be identified or recognized in the system. The condition set of a pattern is minimal in the sense that if any were removed the pattern would no longer represent a potential error (i.e., an error can be corrected by changing any one of the conditions that imply it). The following are examples of conditions:

"The calling procedure has write-access to cell X."

"The value of parameter Y is critical to procedure P."

"The address of W is calculated as a function of Z."

"Procedure A calls procedure B."

"Control is passed to B in the environment of A."

Initially, to maintain a clear connection between a pattern and the error from which it was derived and to avoid overlooking possible areas of application of that pattern, it is important to express it in terms specific to its source operating system. For this reason the initial pattern is called a "raw pattern." The following is a raw pattern for the above error:

1. Load is called by Snap Dump to return the core address of IEAQAD0A.

2. It is critical to Snap Dump that the module loaded is the actual system module IEAQAD0A.

3. The identity of the module loaded is not verified by either Load or Snap Dump.

More formal and concise pattern notation and terminology are being developed; these will be reported in a subsequent document.

Given a raw error, it is often difficult to write down a pattern that satisfactorily captures the essence of the error. First, of course, the error description must be thoroughly comprehended, e.g., in terms of how the error could be exploited by a knowledgeable penetrator. This requires familiarity with the operating system context in which it occurred. Even then it may not be clear precisely what policy is being violated and thus what conditions should constitute the pattern. Consider the

"pass-through" problem, for example [McP74]. A supervisor procedure P may be programmed to omit the validity check for a critical input parameter X when called by other supervisor procedures, assuming that X is a properly maintained system data element in such cases. Under the assumption that P checks X, another supervisor procedure Q calls it with an argument for X that has been user-specified. The policies associated with P and Q are inconsistent. In such cases, in which different but equally valid policies can be postulated, the same raw error leads to more than one pattern. Conversely, of course, many raw errors can result in similar initial 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 another area 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, simple 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.

The following is a generalization of the raw pattern discussed previously:

1. Supervisor procedure A is called by supervisor procedure B to return the core address of a procedure or data element C having name N.

2. It is critical to B that C is the bona fide system element named N.

3. The identity of C is not verified by either A or B.

Here the names of the specific routines have been replaced parametrically.

Conversely, the more general the pattern and the broader its applicability, the less directly relevant it will be to particular functional areas of particular systems and the less immediate utility it will have as a search criterion, since its features must first be

identified with as many as possible of those of the target system. This is discussed in the next section. The opposite of generalizing a pattern is "instantiating" it by substituting examples or instances for one or more of its features. Just as the same pattern can have many generalizations, a given (non-raw) pattern potentially has many instances.

The derivation of raw patterns, their generalization, and the instantiation of generalized patterns toward other systems and functional areas 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 grows, major substructures may emerge, at least below some level of abstractness. If, as is also expected, the search techniques determined to be appropriate for the patterns of each such substructure are also similar, then a reasonable basis will have been provided to define distinct major "error types."

4. *DEVELOPMENT AND APPLICATION OF PATTERN-DIRECTED TECHNIQUES*

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 and the target to bring them into a suitable form, as long as essential properties are preserved. In the case of pattern-directed PE, 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. With this concept, PE is regarded as consisting of two subtasks:

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

2. Executing the comparison program.

Such an ideal is clearly out of reach. There exists no model into which the protection-relevant features of existing systems 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. Much research is now being done on the question of what actually should constitute a protection "kernel" [Pan74], including the effort to identify a kernel for Multics* and efforts to design new systems based on this notion, such as Hydra [Wul74] and the UCLA-VM system [Pop74].

---

*Private communication with Jerome Saltzer.

Nevertheless, the goal of developing pattern-directed techniques and tools to systematize and automate PE [illegible]ns valid. We must investigate what the requirements for these techniques stated in Section 2 imply about their form, application, and development.

First, the requirement for 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 then 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 second requirement, i.e., 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. Such searches can often be mechanized, as seen in the example given in the next section.

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 (potential) error conditions exist. Obviously, the comparison is still not a direct one, 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.

The above considerations have led us to an evaluation process consisting of two steps that are similar but more straightforward than the two required for the ideal evaluator described above.

1. "Feature extraction," involving the instantiation of generalized features, the search for instances of these features in the target system, and the description of their relevant contexts.

2. Comparison of combinations of feature instances and their contexts with the features and relations expressed in the given patterns.

The nature of the techniques and tools to be developed has become more apparent. They consist, for a given set of error patterns, of a set of generalized directives for searching an operating system for instances of the features of those patterns and describing the instances found, as well as formal or informal procedures for evaluating the results of the search with respect to the given properties and relations.

In view of the inherent problems, an effort to develop such tools would still appear to be too ambitious were it not for the simple observation that it is not necessary to have an integrated package that (1) contains directives for a large number of error patterns, (2) includes a single general-purpose comparison algorithm, and (3) is based on a single comparison language. Instead, a set of relatively simple packages can be developed, each tailored to a particular error type of interest. This means that instead of requiring general solutions to the problems discussed above, the approach requires only solutions to problems local to each error type. The directives for instantiating and identifying the features of some patterns, and for describing their contexts, are relatively easy to formulate, and their comparison procedures relatively easy to specify. The directives and procedures of each search package can be designed to best suit that package alone. A set of techniques enabling protection evaluators to search an operating system economically and reliably for instances of even an incomplete set of types of potential errors would be extremely valuable. A reasonable approach is to start with those error types for which the payoff, in terms of the likely cost of such errors in existing operating systems, is greatest relative to the effort required to develop effective search packages for patterns of these types.

The concept of a single tool is therefore replaced by that of a set of packages of techniques, initially small but continually expanding in its coverage, and useful from the beginning. (Of course, certain packages may accommodate error types for which

feature extraction directives or comparison algorithms are similar.) The effect of this approach is that an enormous, monolithic manual PE process has been broken up into a set of smaller and much more manageable processes, each concerned with one or a few error types, and each consisting (conceptually) of two subprocesses: feature extraction and comparison. An example of the application of such a package is sketched in the next section.

5. AN EXAMPLE

The application of a pattern-directed search technique is illustrated by a simple and well-known error type that can be characterized as "the inconsistency of a data value between two references." The error is first represented below by an informal but highly abstract pattern; it is then instantiated into two familiar examples, and finally the second case is used to illustrate specific search techniques. This example is discussed in more detail in [Bis75].

The pattern is the following:

1. Operator G reads a value from or writes a value into cell X.

2. Operator F reads the value from cell X.

3. Operator G is applied before operator F.

4. It is critical (in a protection sense) that the value read by F is consistent with that read or written by G.

5. Between the applications of G and F, the value in X can be modified by a process other than that in which F is applied.

This pattern can be instantiated by substituting concrete examples for the abstract notions of "operator" and "cell" (see Section 3). One common subtype of this error is that which results when G and F are regarded as distinct supervisor procedures and X as a global variable. For example, G and F might be the "checkpoint" and "restart" procedures of an operating system, with X being the file used to store a checkpointed computation. If a checkpoint includes sensitive state information and this file is modifiable by a user between the checkpoint and restart times, then that user could cause his computation to be restarted with improper privileges.*

---

* This error has existed and has been exploited in the third-generation operating systems of more than one manufacturer.

A second common subtype results when G and F are low-level operators within a single user-callable supervisor procedure and X is a parameter of that procedure, passed by reference. This subtype can be expressed as follows:

1. Operator G (of supervisor procedure P) reads or writes parameter X (passed by reference).

2. Operator F (of P) reads parameter X.

3. Operator G is applied before operator F.

4. It is critical that the value read by F is consistent with that read or written by G.

5. Between the applications of G and F, the value of X can be modified by a user process.

The features of the pattern are:

a) Supervisor procedure (callable by user procedures).
b) Parameter passed by reference.
c) Operator that reads or writes the value of a given parameter.

To search for instances of these features, "operator that reads or writes" might be instantiated, for example, to "code that fetches or stores." The context of item C needed to determine relations of which item C is a participant, are (1) whether it reads or writes the parameter, (2) its location in the (uninterpreted) flow of control of the procedure, and (3) if it does read the parameter, whether or not the consistency of its value is critical. If these properties are obtained during the "feature extraction" step, then the subsequent "comparison" step need only determine whether the relation "before" holds between any operator reading or writing any parameter and another that reads the same parameter and is "critical." If this relation holds, a potential error is indicated.

In the above scheme, most of the work is done during feature extraction, while the comparison step is trivial. Actually, except for determining the criticality of an operator that reads a given parameter (which can sometimes require considerable analysis by a person familiar with the system), feature extraction in this case is a well-defined procedure and can be entirely automated (if reference parameters can be recognized). It requires a sophisticated program, of course, to evaluate flow of control. If final determination of flow location is also left for later, then feature extraction is

straightforward. In a program of moderate size, it is usually easy to determine by visual inspection whether one operator can occur before or after another. This illustrates the tradeoffs that can be made between the two steps of the PE process and the flexibility with which evaluation techniques can be designed for a given error type.

As an initial exercise to judge the feasibility of the general approach, the above pattern was applied in the manner just described to portions of the Multics operating system. Since Multics is written in a higher level language (PL/1), and since each of the pattern features has a concrete PL/1 representation, there were no difficulties in identifying and extracting instances from the original text. A TECO [Tec73] program was written for this purpose. Several instances of errors previously unknown were detected and verified.

6. SUMMARY

While important advances have been made in the design of protection mechanisms, they are not generally applicable to existing general-purpose operating systems, in which there is a huge investment. The protection aspects of such systems are notoriously unreliable and the security risks accompanying their use in certain environments are high. This paper has addressed itself to the problem of "protection evaluation" --searching for protection errors using informal static methods, i.e., methods that depend primarily on the use of system documentation and program listings. There is a severe shortage of anything but the most rudimentary tools for this task. Techniques are needed that can be applied to a wide class of operating systems and that do not depend on the evaluator's being an expert in the field of security and privacy.

An approach has been proposed in which formalized patterns are used to direct the protection evaluation task. The patterns are derived from the analysis of errors previously detected, possibly in quite different systems. The report discusses the principal components of a pattern-directed methodology--formulating and generalizing patterns, instantiating them to different systems and functional areas, identifying instances of the features of given patterns in a target system, and comparing the properties of the instances found with those indicated by the patterns. It concludes that the best approach is to develop techniques that are general-purpose with respect to operating systems but special-purpose with respect to error types. Among the advantages of this approach are that the techniques are simpler and can be optimized to particular error types, the approach is empirical rather than theoretical, its payoff begins sooner, and a set of such tools is expandable in coverage and applicability.

Examples are given of errors, corresponding patterns, and the application of a pattern-directed technique to the search for errors of a particular common type.

ACKNOWLEDGMENTS

Listings of portions of the Multics operating system, as well as technical assistance in evaluating potential errors found in Multics, were kindly provided by Jerry Saltzer and David Clark. Dennis Hollingworth supplied numerous error descriptions and helpful suggestions for patterns.

REFERENCES

1. [And72] Anderson, James P., ***Computer Security Technology Planning Study***, U.S. Air Force, ESD-TR-73-51, Vol. 2, October 1972.

2. [Bis75] Bisbey, Richard; Popek, Gerald J.; Carlstedt, Jim, ***Inconsistency of a Single Data Value Over Time***, USC/Information Sciences Institute (in preparation).

3. [McP74] McPhee, W.S., "Operating System Integrity in OS/VS2," ***IBM Systems Journal***, Vol. 13, No. 3, 1974, pp. 230-252.

4. [Pan74] Panel Session--Security Kernels, ***AFIPS Conference Proceedings***, National Computer Conference, Vol. 43, AFIPS Press, 1974, pp. 973-980.

5. [Pop74] Popek, Gerald J.; Kline, Charles S., "Verifiable Secure Operating System Software." ***AFIPS Conference Proceedings***, National Computer Conference, Vol. 43, AFIPS Press, 1974, pp. 145-151.

6. [Tec73] ***TENEX TECO***, Bolt Beranek and Newman, Inc., Cambridge, Mass., October 1973.

7. [Wei73] Weissman, Clark, ***System Security Analysis/Certification Methodology and Results***, System Development Corporation, SP-3728, October 8, 1973.

8. [Wul74] Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; Pollack, F., "HYDRA: The Kernel of a Multiprocessor Operating System." ***Communications of the ACM***, Vol. 17, No. 6, June 1974, pp. 337-345.