Property-Based Testing; A New Approach to Testing for Assurance

George Fink & Matt Bishop
Department of Computer Science
University of California, Davis
email: gfink,bishop@cs.ucdavis.edu

Abstract

The goal of software testing analysis is to validate that an implementation satisfies its specifications. Many errors in software are caused by generalizable flaws in the source code. Property-based testing assures that a given program is free of specified generic flaws. Property-based testing uses property specifications and a data-flow analysis of the program to guide evaluation of test executions for correctness and completeness.

Introduction

Analysts test computer programs to determine if they meet reliability and assurance goals. In other words, testing validates semantic properties of a program's behavior. In order to do this, the actual program must be tested at the source code level, not at some higher-level description of the program. However, to validate high-level properties, the properties must be formalized, and the results of the testing related formally to the properties.

Property-based testing [FL94, FKAL94, FHBL95, Fin95] is a testing methodology that addresses this need. The specification of one or more properties drives the testing process, which assures that the given program meets the stated property. For example, if an analyst wants to validate that a specific program correctly authenticates a user, a property-basted testing procedure tests the implementation of the authentication mechanisms in the source code to determine if the code meets the specification of "correctly authenticating the user."

This paper introduces an approach to property-based testing and an implementation of that approach. First, the analyst specifies the target property in a low-level specification language called TASPEC (Tester's Assistant SPECification language). The program is sliced [Wei84] and code irrelevant to the property disregarded. The Tester's Assistant automatically translates the TASPEC specification into a test oracle that will check the correctness of program executions with respect to the desired property. A new path-based code coverage metric called "iterative contexts" [Fin95, Fin96] efficiently captures the slice-based computations in the program.

Property-based testing speaks to the following questions:

1. What is to be accomplished or established via testing?
2. What test data should be used?
3. When has enough testing been carried out?
4. How is it determined if a test is a success or a failure?

This paper presents an overview of property-based testing, its goals, and techniques used to accomplish these goals. The next section defines the problem, and discusses previous work. The next section describes property-based testing in general and its components in particular, illustrating property-based testing through an example. The final section concludes with future directions for work on this methodology.

Problem Statement

Trust that software programs work correctly and precisely is based upon the belief that authors of the programs have detected and fixed flaws in the design and implementation. Many potential flaws can be detected and avoided; however, systematic and formal analysis (both static and dynamic) of the finished program increases the assurance that the software is without critical flaws.

Most errors in programs result from programming and design mistakes. Many well-known mistakes are still common. For example, errors in bounds checking, race conditions, and authentication, continue to be the bane of privileged Unix programs.

Specifying well-known mistakes formally presents a clear picture of testing goals. Then, techniques are needed to map these formal descriptions to tests of actual code. The tests need to provide formalizable results that relate to the flaw descriptions. The whole process should be as automatic as possible, with reusable generic specifications.

Related Work

Property-based testing is complementary to software engineering life cycle methodologies. Analysis and inspection of design, requirements, and code help to prevent flaws from being introduced into source code. Property-based testing *validates* that the final product is free of specific flaws. Because property-based testing concentrates on generic flaws, it is ideal for focusing analysis late in the development cycle after program functionality has been established.

Specifications state what a system should or should not do. Many specification languages support precise expression of requirements, such as Z [Dil90] and VDM [AI91]. Treating specifications as bounds of program behavior suggests that test oracles can be derived from specifications; some specification languages like Larch [GH93] and TAOS [Ric94] allow this to be done automatically. Further, specifications can guide the generation of test data; ADL [CRS96], TAOS [Ric94], and VDM [DF93] allow this as does the TASPEC language presented here. The advantage of using specifications is the formalism they establish for verifying proper (or improper) program behavior.

Specifications are the basis of formal analytical tehcniques. Determining which assumptions (axioms) are correct is substantial, and failing to do so correctly would invalidate the analysis. For example, if an operation has an unanticipated side-effect during execution in some situations, formal analysis cannot determine the impact of the side-effect upon cor-

rectness. While testing has similar problems, it does test the actual execution of the program, and can determine the precise output corresponding to a given input. For example, thorough testing can determine unanticipated side effects.

Coverage metrics measure testing completeness; how much of the program has been tested? For property-based testing, a coverage metric must be strong enough to provide formal assurance, but also be feasible to implement and utilize. Property-based testing uses a new metric called Iterative Contexts, which strikes a balance between simple definition-use (def-use) pair metrics [Las90, Nta84, CPRZ89] and stronger but impractical path coverage metrics [RW85].

Testing to Validate Programs

A test consists of a set of executions of a given program using different input data for each execution; its purpose is to determine if the program functions correctly. A test has a *negative* result if an error is detected during the test (i.e., the program crashes or a property is violated). A test has a *positive* result if a series of tests produces no error, and the series of tests is "complete" under some coverage metric. A test has an "incomplete" result if a series of tests produces no errors but the series is not complete under the coverage metric.

It is impossible to execute a program on all possible data. So, testing must approximate this, which may lead to an incorrect validation. However, for a testing process to be valuable, it must validate a program with respect to a property with a high degree of certainty. Property-based testing addresses this conflict with iterative contexts, a new data-flow coverage metric.

It is important to understand the relationship between testing and formal verification so that the two can be compared. The purpose of property-based testing is to establish formal validation results through testing. To validate that a program satisfies a property, the property must hold whenever the program is executed. Property-based testing assumes that the specified property captures everything of interest in the program, because the testing only validates that property. Additionally, property-based testing assumes that the completeness of testing can be measured structurally in terms of source code.

The property specification guides dynamic testing of the program. Information derived from the specification determines what points in the program need to be tested and if a test execution is correct. The iterative contexts coverage metric, based upon these points, determines when testing is complete.

Therefore, in property-based testing, checking the correctness of each execution together with a description of all the relevant executions of the program validates a program with respect to a given property.

Tester's Assistant

Figure 1 shows an overview of the implementation of property-based testing by the Tester's Assistant. To test the source code of a program, TASPEC specifications from a variety of sources are used. Program-independent specifications include system call, security, and generic flaw specifications. If necessary, program-specific specifications can also be used. The Tester's Assistant analyzes and tests the code with respect to the specifications. Three results of the property-based testing process are: the test suite, the coverage results, and/or flaws discovered during the test.

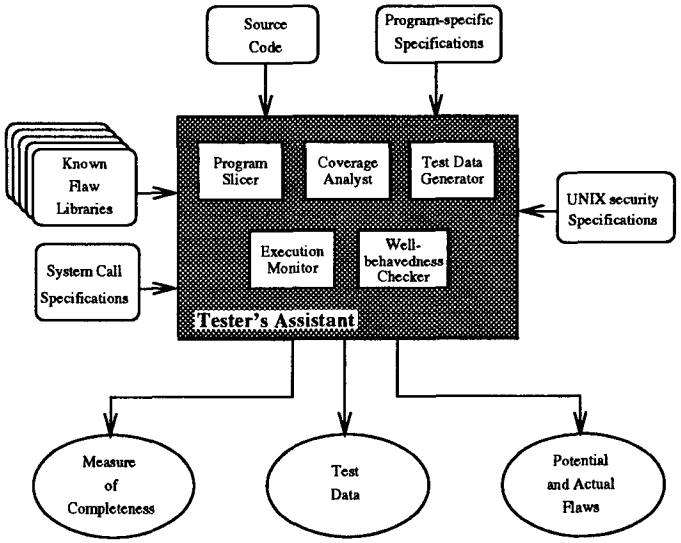

Figure 1: Property-based testing and the Tester's Assistant.

Many properties are defined independently of specific programs (for example, array bounds, race conditions, authentication), and so can be grouped together in libraries of properties. These libraries form models of system behavior, which are significant analytical objects in their own right. They can be reused and also analyzed by independent means to assess their completeness[4].

Iterative Contexts

The iterative contexts coverage metric is an ideal metric for satisfying property validation requirements. Iterative contexts are more powerful than other data-flow metrics [Las90, Nta84, CPRZ89], but are small enough so they can be satisfied by a reasonable test suite. Given a set of variables at a point in the program that are of interest, the optimal metric requires all possible results for that set of variables; for most sets this requires an infinite number of data values. Metrics based upon sequences of assignments within the slice approximate this optimum for given programs.

An iterative context is a sequence of assignments defining a sub-path of a possible program execution. The assignments are taken from the program slice and represent a possible computation of a value important to the target property. Taken together, all of the contexts represent many of the possible

[4]Through a previous iteration of property-based testing, perhaps.

computations of values relevant to the property. It is not possible to represent with a finite set of input data the infinite number of possible computations for some loops, so in those cases iterative contexts will not completely cover all behavior relevant to a property. In a complete test suite, every context must be represented by at least one test execution in the suite.

Static Analysis and Slicing

Program slicing [Wei84], the extraction of all code affecting conformance to a property, reduces the amount of code that a human tester must inspect manually. Applying automatic analysis tools to the slice rather than to the whole program also aids the analyst. Calculating a slice requires detailed global dependencies; this information is also used to generate iterative contexts.

TASPEC

TASPEC, the specification language used in the Tester's Assistant and developed specifically for property-based testing, has primitive constructs which enables it to be translated automatically into slicing criteria and test oracles. TASPEC includes basic logical and temporal operators as well as location specifiers, which associate events with code features. These events provide the primitive data for analyzing higher-level semantic features of the program. TASPEC is a flexible low-level specification language well suited for specifying a wide range of properties and deriving tests from the property specifications.

Using location specifiers, generic program-independent properties in TASPEC map automatically to source code. Therefore, test oracles can be generated independently of descriptions of specific modules or functions. With the emphasis on *properties* and not on full specifications, test oracles can handle a wider class of behavior than that rigidly defined by functional specifications. Translations between other specification languages and TASPEC can provide additional flexibility to the specification and testing phases of development. Helmke shows how translations from Z to TASPEC can assist in requirements traceability [Hel95].

Execution Monitors

Automatic high-level execution monitors derived automatically from property specifications in TASPEC become test oracles that assess the correctness of executions. Location specifiers produce primitive events for the specification state and the execution monitor processes these elements to raise higher-level events. The execution monitor checks for consistency between events and the property specification. Therefore, checking the adherence of a program execution to a complex property specification is automatic.

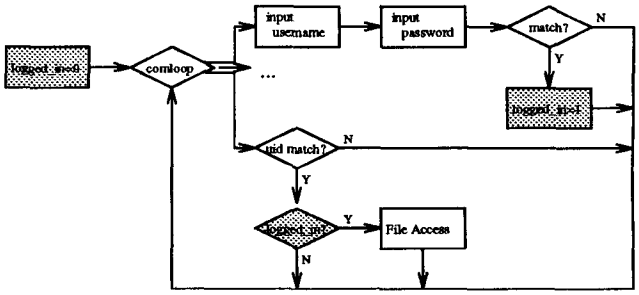

Figure 2: Ftpd flaw flowchart.

Example use of Property-based testing

This section describes testing a version the Unix ftpd (file transfer protocol [CER](FTP) daemon) program with property-based testing. Property-based testing has eight steps:

1. Selecting a property; the property is specified in TASPEC (currently implemented)
2. Static analysis and slicing (currently implemented)
3. Program instrumentation (currently implemented)
4. Initial test case selection and execution
5. Coverage evaluation (partially implemented)
6. Additional test case selection and execution
7. Correctness evaluation (partially implemented)
8. Repeat the last three steps as necessary

Testing ftpd with respect to an authentication property reveals a flaw in ftpd's authentication code.

Description of ftpd and its flaw

Ftp is a Unix program implementing the FTP protocol for transmitting files across a network. Ftpd, the program described here, is a server program that accepts file requests and processes authentication and other utility commands from remote client programs.

In the version of ftpd released with SunOS 3.2, a security flaw allows any user to gain permissions to read or write files owned by any user on the system (including root) [CER]. To do so, the user logs on with his or her normal user name and password. As a part of the correct authentication, a flag in the program is set. The flag records whether the user name has been authenticated. When a second user name is entered, the flag is never reset, so even if an incorrect password is entered for the second user name, the program thinks that the second user name has been authenticated. Therefore, the user has the access privilege of the second user name. Figure 2 is a simplified flow-chart that illustrates the flaw.

location func setuid(uid) result 1
{*assert permissions_granted(uid);* }

location func crypt(password, salt) result encryptpwd{
assert password_entered(encryptpwd); }

location func getpwnam(name) result pwent{
assert user_password(name,
pwent → *pw_passwd, pwent* → *pw_uid);*
}

location func strcmp(s1, s2) result 0{
assert equal(s1, s2); }

password_entered(pwd1) ∧
user_password(name, pwd2, uid) ∧
equal(pwd1, pwd2){*assert authenticated(uid);* }

Figure 3: Property specification for authentication.

Selecting/identifying a property

The first step in property-based testing is to choose a property or properties from a selection of generic properties, and to write any specific program-specific properties to test. Property specifications are written in TASPEC. In the case of ftpd, a generic property is used.

A portion of the property library is a set of properties which describe a security model. One high-level property specification requires that authentication occur before any permissions are granted:

authenticated(uid) before permissions_granted(uid).

The library also contains low-level definitions of the predicates *authenticated* and *permissions_granted*, shown in Figure 3. In TASPEC actions within curly braces occur when the condition (either a program location or a logical predicate about the specification state) before the curly braces occurs. For example, the **setuid(uid)** location, when executed, causes the *permissions_granted* predicate to be true in the specification state.

The authentication property can be selected by hand. Optionally, an automatic tool could compare location specifiers (code templates) in the property specifications with the source code of ftpd to evaluate the relevance of properties in the library. The definition of *permissions_granted* involves the **setuid** system call[5]. The property, then, forms a pre-condition for the **setuid** system call. Since ftpd contains **setuid**, the authentication property can be automatically chosen as an important property for which to test.

[5] **Setuid is used here as an amalgam of the many different permissions-setting system calls (seteuid is actually used by ftpd).**

Static analysis and slicing

The Tester's Assistant statically analyzes the source code for ftpd. Ftpd contains about 3000 lines of C code, 1700 lines of which are machine-generated by lex and yacc. The static analysis produces a data-flow graph for ftpd. The ftpd data-flow graph has 6148 nodes and 31912 edges. The data-flow graph is used in other steps of the process: program instrumentation, coverage evaluation, additional test case generation, and correctness evaluation.

Next, slices of ftpd are derived from the data-flow graph. First the slicer generates a slice of ftpd with respect to the selected authentication property. The human tester inspects the slice manually, but even in the sliced code (represented in Figure 2) the flaw is subtle enough that it goes unnoticed. At this point the human tester can request additional slices based upon any other criteria that can aid in the tester's understanding of ftpd.

Program instrumentation

The Tester's Assistant produces an alternate version of ftpd to execute during testing. The alternate version has the same functionality as ftpd, but has additional data-gathering modules, so that coverage and correctness can be evaluated from test results. Every section of source code corresponding to a location specifier in the property has code added to record if and when the section of code is executed. The added code is used later in correctness evaluation. The assignments in the source code that are significant for coverage evaluation are also tagged to record when the assignments are executed. The Tester's Assistant instruments only the slice relative to the selected authentication property. The instrumented source is then compiled, at which point ftpd is ready to be executed.

Initial test executions

The instrumented ftpd is executed several times with various test data. There are three ways to generate test data for ftpd: First, use any available test data that was used in initial testing and debugging. Second, have the analyst generate simple test data from a description of ftpd's functionality. Finally, if there are any specifications of ftpd, the specifications can be used to generate test data. Generating test data from specifications is not specifically part of property-based testing, but other testing methodologies contain the necessary algorithms [CRS96, DF93].

The first method is simplest, because no extra work is required and the test suite is likely to be fairly complete. However, if these test cases aren't available, the analyst creates some test cases by reading the ftpd manual page. Figure 4 shows some sample test cases.

The test executions are then evaluated for coverage and correctness. None of the four executions result in a violation of the authentication property. However, coverage evaluation

Test Case 1

```
user <user name>
pass <incorrect password>
retr filename
```

Test Case 2

```
user <user name>
pass <correct password>
retr filename (no access permissions)
```

Test Case 3

```
user <user name>
pass <correct password>
cwd  directory
retr filename1 filename2
```

Test Case 4

```
user <user name>
pass <correct password>
list
```

"User" enters a user name, "pass" enters a password, "retr" retrieves a file, "cwd" changes directory, and "list" lists a directory.

Figure 4: Four initial test cases for ftpd.

reveals that ftpd has not been completely tested, so more test cases must be found and executed.

Coverage evaluation

While ftpd executes with each given test data, the coverage instrumentation writes a file recording the execution history of the slice. The execution history indicates which path in ftpd was executed. The initial test executions yield several execution histories. The execution histories are compared with the coverage metric. Property-based testing uses iterative contexts. Each context is an ordered sequence of assignments, which defines a sub-path of the program. For a history to match a context, the assignments must be executed in order with no intervening and interfering assignments. The contexts are generated using static analysis and the data-flow graph,

For the (abstracted) fragment of ftpd source

```
(1) logged_in = 0;
(2) while(1)
(3)   switch(cmd) {
(4)     user:  name = read();
(5)            pass = read();
(6)            if(match(name,pass))
(7)              logged_in = 1;
(8)            break;
(9)     get:   if (logged_in)
(10)             setuid(name);
(11)    }
```

the contexts required include

$$\{\{4,5,6,10\},\{4,5,6,4,10\},\{4,10,4,5,6\}\}$$

.

The execution histories are compared with the set of contexts to see which histories match which contexts. The unmatched contexts are coverage gaps.

The execution histories from the four initial test cases are

$$\{\{4,10,4,5,6\},\{4,5,6,10\},\{4,5,6,10\},\{4,5,6\}.$$

The second and third execution histories are identical because their behavior relative to the property specification is identical. The context $\{4,5,6,4,10\}$ is a coverage gap in the initial test data, and corresponds to the flaw in ftpd.

Additional test cases

In order to complete the coverage metric, additional executions of ftpd are necessary, with different test data that addresses the coverage gaps. This paper does not present a method to produce this additional test data automatically, and the problem is not trivial.

A human tester produces additional test data by examining the contexts not covered and the code corresponding to the contexts. For the contexts and code in ftpd, there is a close correspondence between input statements and statement numbers in the uncovered context (Statements 4 and 5). The uncovered context $\{4,5,6,4,10\}$ is executed by the the following test script:

```
user <user 1's name>
pass <user 1's password>
user <user 2>
pass <random string>
retr  filename
```

Correctness evaluation of this execution detects that the flaw exists in ftpd.

Future versions of the Tester's Assistant may be able to automate some of the steps in generating test data for gaps in coverage using techniques based upon symbolic execution [DO91].

Correctness evaluation

During each test execution, a file records the activated TASPEC primitives. The TASPEC evaluation engine processes this data and compares it with the property specification. If the data violates the property specification, then

the human tester is informed that the test caused an error condition.

During processing of the correctness records for the additional test case given above, the correctness monitor registers that there is a correct authentication of user 1. No authentication of user 2 is registered, because the password match fails. When the file retrieve action occurs, the *permissions_granted* property is registered. However, the retrieve occurs with the permissions of user 2, for whom there is no authentication. Therefore, the additional test case causes an error condition, so ftpd fails the property-based testing with respect to the authentication property.

Applications to Computer Security

Assuring that computer programs and systems are secure is an important and difficult problem. Security flaws are still being discovered in computer programs that have been in use for many years. Many of the flaws are caused by the same basic recurring faults [Spa92]. For example, the Internet worm [Spa89] exploited errors in Unix network programs. Examination of the flaws which caused the errors revealed them to be of an elementary nature.

It is time for a concerted effort to try to prevent such flaws from occurring. Therefore, an appropriate initial application of property-based testing and the Tester's Assistant is Unix security, specifically for network programs. Security is a good application of property-based testing because the parts of programs that relate to security are small, and generic security properties can be precisely expressed program-independentally with TASPEC.

Security Issues

Networked systems cause special security problems because any communication or authentication between networked systems must be performed entirely through an exchange of information. The exchange of information is limited by the network structure as well; many networks in use today are asynchronous, and make no strict delivery guarantees for information packets. Problems with asynchrony are complicated by different implementations for the same service protocol, which may have different performance. Therefore, network services must be flexible in their implementation of communication and authentication services. This flexibility can sometimes be exploited and become a source of security problems, adding to security problems arising from bad design or implementation.

Network services with Unix involve the client/server model. The server runs on a host machine, and regulates access to information on the host by communication with client processes on other machines on the network. The server can do its task in one of two ways: by forking off a server-end client process to handle commands, or by doing all the work internally. In either case, the server will be interacting with the host system in a number of ways – reading/writing files, etc.

Most network servers are privileged programs; they are run with root privileges on the host machine. Unix has a coarse-grained tri-level file protection scheme. If the access level for a process cannot fit into this scheme, the process must be given root level permissions, which override the scheme. Network services typically do not fit into the tri-level scheme and are given root permissions, even though root permissions are used in only one particular function of the program. Therefore, the server is given excess privileges, which become fertile ground for exploitable vulnerabilities.

Using property-based testing for security

Formal testing with property-based testing can validate security properties of software and thus produce secure systems. Security-related code is often only a small part of a program's functionality. Property-based testing focuses on code relevant to security functionality in great detail, and so efficiently validates the security-related part of the program without testing the whole program.

Property-based testing provides a methodology for testing narrow properties of source code. It produces a specific and absolute metric for successful testing with respect to those properties. A successful test validates that properties are not violated; if these properties form the security policy for the system, then the system is secure.

Property-based testing uses a security model of the system, as well as a library of generic flaws (such as [LBMC93, Spa92]) specified in TASPEC, to produce a test process, whereby the target program can be certified to be free of certain types of flaws.

Concluding remarks and future work

Property-based testing defines a formalized framework for testing. With property-based testing tools, a tester can produce a validation that a program satisfies given properties. Other aspects of the process have not yet been well defined. How are properties selected? How is it determined that the properties represent a complete model of a program's possible failures?

Ongoing research into property-based testing at University of California-Davis includes:

- Tool development: automating more property-based testing techniques and incorporating them into the Tester's Assistant, and distributing the tools to gain a wider evaluation base.
- Property specification: specifying generic flaws and features of protocol implementions of TCP and NFS, for example.
- Evaluation of iterative contexts: performing empirical comparisons between iterative contexts and other similar metrics such as L-contexts [Las90].

- Case studies: gaining more experience using the methodology of property-based testing and understanding how it can be applied to different problems.

Acknowledgments Part of this work has been supported by DARPA, under contract USNN00014-94-1-0065.

References

[AI91] Derek Andrews and Darrel Ince. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[CER] CERT advisory CA-88:01.ftpd.hole.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1331, November 1989.

[CRS96] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. Submitted to ISSTA 1996 as a Regular Paper, 1996.

[DF93] Jeremy Dick and Alain Faivre. *Automating the Generation and Sequencing of Test Cases from Model-Based Specifications*, chapter 4, pages 268–284. First International Symposium of Formal Methods Europe Proceedings. Springer-Verlag, 1993.

[Dil90] Antoni Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, 1990.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[FHBL95] George Fink, Michael Helmke, Matt Bishop, and Karl Levitt. An interface language between specifications and testing. Technical Report CSE-95-15, University of California, Davis, 1995.

[Fin95] George Fink. *Discovering security and safety flaws using property-based testing.* PhD thesis, UC Davis, 1995.

[Fin96] George Fink. Iterative contexts, a complete and practical data-flow coverage metric. In preparation, 1996.

[FKAL94] George Fink, Calvin Ko, Myla Archer, and Karl Levitt. Towards a property-based testing environment with applications to security-critical software. In *Proceedings of the 4th Irvine Software Symposium*, April 1994.

[FL94] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Tenth Annual Computer Security Applications Conference*, pages 154–163. IEEE Computer Society Press, December 1994.

[GH93] John V. Guttag and James J. Horning. *Larch: Langauges and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hel95] Michael Helmke. A semi-formal approach to the validation of requirements traceability from Z to C. Master's thesis, UC Davis, September 1995.

[Las90] Janusz Laski. Data flow testing in STAD. *Journal of Systems Software*, 12:3–14, 1990.

[LBMC93] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws, with examples. Technical Report NRL/FR/5542-93-9591, Naval Research Laboratory, November 1993.

[Nta84] Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[Ric94] Debra Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[Spa89] Eugene. H. Spafford. The internet worm: Crisis and aftermath. *Communications of the ACM*, pages 678–687, June 1989.

[Spa92] Eugene H. Spafford. Common system vulnerabilities. Workshop on Future Directions in Intrusion and Misuses Detection, 1992.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–375, July 1984.