# Property-based testing of privileged programs *

George Fink Karl Levitt
Department of Computer Science, University of California, Davis
Davis, CA 95616
email contact: gfink@cs.ucdavis.edu

## Abstract

*We address the problem of testing security-relevant software, especially privileged (typically setuid root) and daemon programs in UNIX. The problem is important, since it is these programs that are the source of most UNIX security flaws. For some programs, such as the UNIX* **sendmail** *program, new security flaws are still discovered, despite being in use for years. For special-purpose systems with fewer users, flaws are likely to remain undiscovered for even longer. Our testing process is driven by specifications we create for the privileged programs. These specifications simultaneously define the allowed behavior for these programs and identify problematic system calls, regions where the program is vulnerable, and generic security flaws. The specifications serve three roles in our testing methodology: as criteria against which a program is sliced, as oracles against which it is tested, and as a basis for generating useful tests. Slicing is employed to significantly reduce the size of the program to be tested. We show that a slice of a privileged program (rdist) with respect to its security specifications is quite small. We introduce the Tester's Assistant, a collection of tools to mechanize the process of testing security-related C programs.*

## 1 Introduction

Many of the security problems in UNIX are due to errors in privileged[1] programs, in network daemons, or in unprivileged programs run by the superuser. Examples of such flawed programs include **fingerd, ftp,** and **rdist**. Security-related flaws in these programs have been discovered years after their release. Errors might still lurk undetected in these and other privileged programs (or in network software) until discovered by an attacker. Clearly, an approach to screening these programs that is more effective than the current practice is needed; this paper considers systematic testing as its approach to screening.

As a step towards a systematic approach to the testing of security-relevant programs, we consider *property-based testing.* Property-based testing uses specifications of important properties to produce testing criteria and procedures which focus on these properties in a systematic manner. Portions of the program which do not deal with properties of interest can be ignored. Analyzing only part of the program results in significant performance gains in testing algorithms.

For this application of property-based testing, specifications are produced for security properties of privileged programs. Most uses of specifications (e.g., for verification or for documenting design decisions) indicate "what the program is supposed to do." However, many security properties are negative, describing what the program should not do (i.e., don't let a user log on without a password). Thus, our specifications can take different forms, such as: the objects a program is allowed to access during execution, the identification of system calls that if improperly invoked would cause security problems, and characterizations of generic security flaws [1].

We use specifications for multiple purposes in our methodology, one of which is to generate slicing criteria. A slice of a program with respect to a criterion is a sub-program which has the same behavior as the full program with respect to the criterion. As an example of a useful slicing criterion for security, a security flaw in the login program might mis-assign a user id to the shell created at login; by slicing the login program with respect to the criterion "setuid system call," code that creates an erroneous user id would be identified. We show that slicing with respect to security properties can produce slices that are significantly smaller than

*The work reported here is being supported in part by ARPA under contract USNN00014-94-1-0065 and by the Lawrence Livermore National Laboratory under work order LLNL-IUTB234584.

[1] A privileged program is defined as a program with privileges that cross protection domains.

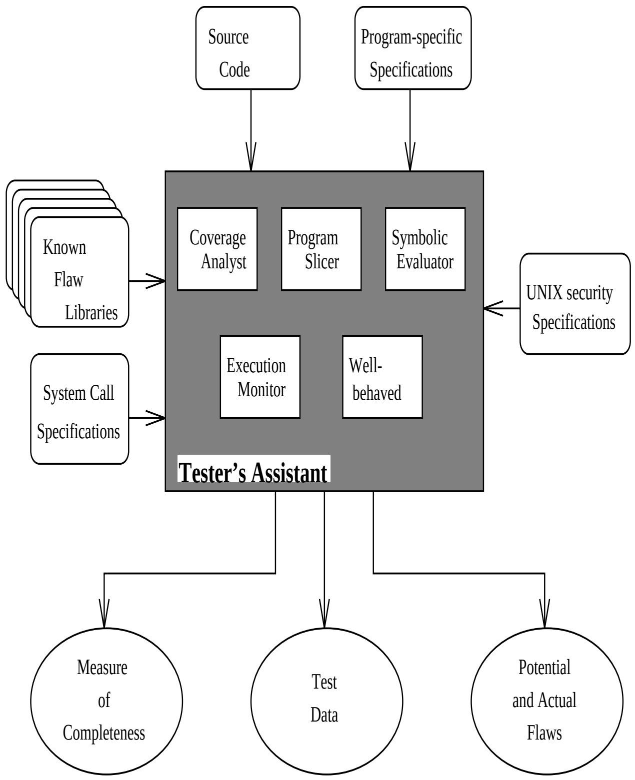


Figure 1: *Overview of the Tester's Assistant*

the original program. The testing effort can then be focussed on the slice.

This paper presents a specification model for describing security properties of programs, and translations from properties into slicing criteria. The slicer is one component of our Tester's Assistant (see Figure 1), which mechanizes property-based testing; it slices C programs, and is implemented in the ELI [17] compiler construction toolset. Still under development are other components of the Tester's Assistant (see Figure 1), including:

- a coverage analyzer, which determines the effectiveness of a set of tests with respect to a coverage measure. For security-related programs, data-flow coverage (the percentage of definition-use paths executed by a test suite) appears to be an effective measure.

- an execution monitor which analyzes audit trails produced by the underlying operating system (e.g. the kernel) during the execution of a privileged program. The audit trails are checked against the program's specification. Audit trail analysis is augmented by automatically instrumenting the slice to provide further information on the program's state.

- a well-behavedness checker, which uses static analysis and testing techniques to identify pointer or array index overflow. Ill-behaved programs have been the source of several security problems.

In Section 2, we outline the security model used in property specifications. In Section 3, we define slicing and discuss the issues associated with it. In Section 4, we apply the property-based testing methodology to several example programs. In Section 5, we give an overview of the Tester's Assistant implementation and report on current progress and design. Finally, in Section 6, we give our conclusions and future plans.

## 2 Security Specifications

There are three main uses of specifications in property-based testing:

1. as criteria for slicing.
2. as oracles for evaluating test runs.
3. to generate test data.

With respect to slicing, specifications provide descriptions of the variables and locations in the program, the variables and locations being "interesting" with respect to security. For more details, see Section 3. An executable specification can be used to determine if a test run produces correct results. Specifications can also be the basis for identifying interesting data values on which the program must be tested. See Section 5 for how the Tester's Assistant uses specifications in the latter two ways.

Specifications are most useful when written in a formal language. This enables the use of automatic tools to process the specification. It is also possible to reason about the interaction of specifications for different programs and flaws, and to form an effective model for security specifications. In the remainder of this section, a model for UNIX security is introduced, and applied to specific UNIX objects and operations.

### UNIX security model

In UNIX, some programs have privileges beyond those needed to carry out their job. This happens because the granularity of UNIX's access control mechanism is not fine enough. A program which runs setuid to root[2] can make any change to the file system. It is necessary to determine that the actual changes these programs make do not indeed exceed their intended purpose. We provide security specifications for privileged programs, and then test the programs with respect to these specifications. For example, in UNIX, the "`/bin/passwd`" program is a setuid root program. Therefore, `/bin/passwd` can potentially do anything (e.g., plant a trojan horse, kill any process, shut down the system). We want to assure that the program is written so as to restrict what the user can make it do.

Not all security properties are negative. In the case of many authentication-related programs (such as login), we want to assure that the authentication is carried out before any privileged actions are taken.

To address these concerns, we create a security model for UNIX programs. A motivating factor behind the model is the desire to produce concise specifications for privileged programs, and to reason about the specifications.

Our model divides security specifications into three categories: general system requirements, the operating system interface specifications, and program-specific specifications. The basic primitives of the specifications are users, objects, and access rights and restrictions.

General system requirements are properties expressed as predicates which, in general, programs should not violate. An example of a system requirement is that the file that stores password information is not accessed, as this would be a breach of security[3]. However, it is obvious that in some instances this requirement must be violated; when a program needs to perform an authentication of a user, a check is necessary against the information in the password file. The system requirements form a stronger barrier between programs and the system.

The second category of specifications defines the interface between programs and the operating system. Within the context of the interface, programs are allowed to violate first category specifications. Each system call that has security implications has a security-related specification. Some system calls have preconditions: predicates that must be satisfied if the system call is to be used in a secure manner. For example, the `setuid` system call, which gives access permission to a process, requires that the user be authenticated. This assertion results in the specification in Figure 2. This specification says that the user should be authenticated before the current user is given permissions. After the system call `setuid` is executed, and if no error code is returned, permissions are granted to that user, and the corresponding assertion is made in the security state of the system.

```
authentication(uid) before
    current_user(uid);

funcall setuid(uid) if result = 0
    {assert current_user(uid);}
```

Figure 2: *Specification of the setuid system call in the Tester's Assistant specification language.*

Library routines are also dealt with in this manner. Although it may be possible to break down library routines into their component computations and system calls, it is more efficient to assume that the standard libraries operate correctly, and to specify the library functions in the same way as system calls. The correctness of the libraries can be established in an independent test run. An important example of library routines is the password library, a set of routines which define a standard interface to the password file.

A program-specific specification indicates when a program is allowed to violate the restrictions of the first and second category of specifications. For example, to allow users to change their passwords or for new accounts to be created, special permission must be given to write to the `/etc/passwd` file. The program to change passwords, `/bin/passwd`, is given a specification which exactly specifies the scope of the changes it can make:

```
passwd(U:uid)
      write(password_file.U.password)
```

Informally, the specification says that when a user executes the "passwd" program, the program can write only to the password file and only the field corresponding to the password of that user.

An additional category of specification describes safety properties and other generic flaws [12] [1]. Safety properties cover common programming mistakes such as lack of argument checking, and include the **well-behavedness** property, described in Section 3.

[2] A setuid root program runs with superuser privileges regardless of which user actually initiated the program. The programs need to be setuid root in order to access and change system objects.

[3] In UNIX, note that this file (`/etc/passwd`) is in fact globally readable; other information kept in the file needs to be generally available. This makes it more difficult to detect security breaches related to the password file, as the default UNIX security policy allows read access, and many programs utilize this policy.

### Specification Language

The syntax of the specification language includes constructs for:

- Specifying an abstract security model with assertions and predicates.
- Attaching assertions and predicates of the model to specific source code such as function calls and variable uses.
- Indicating how specification components can be related with constructors from temporal logic (e.g. before, until).

The UNIX security model expressed in this language includes the following primitives;

- the current user id (uid)
- special 'superuser' user id (root, or uid = 0)
- id's which have been authenticated
- authentication routines
- files/processes (an object in the file system)
- access permissions for files
- ownership of files
- special files - /dev/kmem /etc/passwd /bin/sh

We have determined that this list is sufficient for many programs, including `ftp`, `login`, `rdist`, and `ps`. Other programs which contained security flaws, including `sendmail` and `fingerd`, are not covered by this list of specification primitives. For these programs, some flaws are caused by a potential difference between code that is originally compiled and code that is actually executed, as code can be changed dynamically through overwriting variables on the stack and other ill-behaved operations. Such programs must also be tested against additional well-behavedness type specifications.

Section 4 contains a discussion of how this security model covers UNIX security flaws.

## 3 Program Slicing

Program slicing is an abstraction mechanism in which code that might influence the value of a given variable or set of variables at a location is extracted from the full source code of a program. Weiser [18] originally implemented slicing for FORTRAN programs. Slicing is carried out with respect to a slicing criterion. In its simplest form, a criterion is a variable and a location in the program. There are two essential properties of a slice:

1. the slice must be executable;
2. for the same input values, the variable must have identical values at the corresponding locations both in the slice and the original program.

More complex criteria can involve multiple variables, multiple locations, and include forward slices which capture behavior of the program dependent on a particular variable usage.

A basic *backward* slice of a program is produced by first generating a combined control and data flow graph of the program, based upon the program's parse tree. Nodes that correspond to the slicing criteria for the basis of a slice. A node is added to the slice if it is a definition of the value of a node in the slice, if it is used in a computation of a node in the slice, or if it is part of a control point which dominates a node in the slice. The final slice is the subset of nodes of the flow graph which have been identified by the above algorithm. The slice can then be formatted in syntax appropriate to the original programming language.

Another slicing technique works *forward* through the parse tree, tracing the fan-out effect of a flow

```
FB int example (int x, int y)
FB {
     int z;
     z = 0;
 B   if(x == 0)
       z = x + y;
 B   else
 B     x = y;
 B   if(y)
FB     setuid(x);
     else
       return z;
F    return -1;
F  }
```

Figure 3: *Example of forward and backward slicing with respect to* `setuid`. *Elements of the forward slice have an "F" at the beginning of the line. Elements of the backward slice have a "B" at the beginning of the line.*

node. Combinations of backward and forward slicing (possibly with different criteria) can produce slices which more exactly characterize a program's behavior in some circumstances. This technique is called *dicing* [9]. Different ways in which forward and backward slicing are applied in property-based testing are discussed in Section 4.

The current version of our Tester's Assistant slicer does a data-flow breakdown of C programs, and finds simple slices[4]. The primary cost in slicing is in producing the data-flow representation. The largest example to which the slicer has been applied is the `rdist` server, which required nearly four thousand data-flow nodes, and took between fifteen and twenty seconds to produce on a Sparc10[5].

By using slices as the basis of security testing, we are assuming that testing a slice is equivalent to testing the whole program. The equivalence of a slice and a program with respect to some narrow criterion does not necessarily imply the equivalence of test results. In the Tester's Assistant we rely on the completeness of the model defined by the security specifications to describe the properties of interest. For example, if there are other system calls that influence a property of interest but which are not specified accordingly, there is little the Tester's Assistant can accomplish. For properties that have not been defined (e.g., in our model, properties related to program correctness rather than security), testing program slices will give very little information. A test of a (correct) slice that was created with respect to a specification tests that specification, but not necessarily anything else.

### 3.1 Well-behavedness

Difficulty arises when slicing C programs because the C language allows virtually unlimited pointer arithmetic as well as many forms of pointer (and function) aliasing. This lack of well-behavedness affects the correctness of slices because unexpected side-effects make a correct data-flow graph very difficult or impossible to calculate.

To address this, we make the often unrealistic assumption that pointers are **well-behaved**. A pointer is well-behaved if, once assigned to an object in memory, it does not, through pointer addition or typecasting, refer to a different object in memory. A program is well-behaved if all of its pointers are well-behaved. Lo [11] showed that static analysis can establish the well-behavedness property in many cases without requiring a proof of correctness. The well-behavedness property can also be tested with respect to appropriate well-behavedness specifications. Unfortunately, much of the power of C as a language derives from its being able to support ill-behaved operations. In many cases, this ill-behavedness is used in very specific ways from which it is possible to deduce an underlying well-behavedness property. For example, the use of void pointers and type casting to implement generic data structures, and the use of pointer arithmetic for array indexing are both classic ways in which what appears to be ill-behaved code is actually well-behaved.

There are two reasons why we are able to make the well-behavedness assumption and still be able to make strong claims about the results of property-based testing. First, as mentioned above, it is possible in some cases to statically check the well-behavedness property, as well as to identify the classic ways in which ill-behavedness is used. However, if this is not the case, and it is not possible to make the determination whether or not the program is well-behaved, then the fact that a determination was not able to be made is enough to raise a certain level of suspicion as to the security of the program.

Second, most of the code that is potentially ill-behaved is intrinsic to the nature of C and C++ languages, i.e., in the low level interaction of the language with the underlying system. So, while our examples are written in C and our prototypes are written to analyze C code, the general techniques of specification, slicing, and dataflow testing are actually easier and more technologically feasible if not applied to such a permissive language as C.

As a final note, even though ill-behaved code causes incorrect data-flow graphs and thus renders most of the Tester's Assistant analysis unusable, it remains possible to test against well-behavedness conditions. A running program can be considered well-behaved up to the time in which it makes its first out-of-bounds reference. If this reference is described in a well-behavedness criteria, sliced, and tested, then the analysis of this reference (since it is the first ill-behaved reference) is not tainted by the lack of well-behavedness in a program. In this way, provided that all the ways in which well-behavedness can be violated are specified, it is possible to use the Tester's Assistant for well-behavedness checking.

### 3.2 Slicing Criteria

Anything that can be described in terms of nodes in a dataflow graph can serve as a slicing criterion.

[4] In our current version, pointer analysis is not complete.

[5] Considerable speedups are possible in the slicer, as it is currently an unoptimized prototype.

For example, the set of all function call nodes which correspond to `setuid` system calls is a valid set on which slicing can be performed. This assumes that the slicing algorithm is merely a backward arc closure algorithm, identifying all nodes which possibly influence the node(s) of interest. Slicing can also be accomplished with respect to two or more system calls or with respect to temporal factors, e.g., one of the system calls being required to be executed first (a permissions-checking call is executed before a file is read.)

The basic "unit" of a slicing criterion is a system or other function call, but there are different ways in which these units can be combined that could be of interest. So, when considering a specification language, these additional slice constructors can be used to map a specification to a slice.

From specifications we extract the system calls which are relevant, and slice with respect to them. Slicing of authentication routines poses some challenge, since in UNIX, authentication is not standardized in a single library or system call. It is important to identify specifically the code that is used in authentication and what that code does with correct and incorrect authentications.

Classically, slicing is performed to examine the behavior of a set of variables at a specific point in the program. When we slice for properties, the slicing criteria become more complex. In the simpler cases this involves the values of system call parameters at different points in the program in a cumulative way, i.e., the union of the individual slices. In other more complex cases, the desired slice is the intersection of slices, or is decided by more complex boolean formula. For example, when attempting to slice with respect to the `setuid` specification given in Section 2, the slice will contain all data paths which traverse `setuid` but do not traverse the authentication routine; in this case, a user would gain access without having provided a correct password. The specification of `setuid` indicates that paths which traverse both correct authentication and `setuid` are secure, and thus do not need further testing. Dicing methods can be used to construct this complex slice.

It is important to note that exact (minimal) slices for specifications often are not necessary. As long as a slice sufficiently reduces the size and complexity of the program, other static and dynamic analyses are made more efficient.

Some different security objects (like special files and passwords) might require forward slices. For example, if a password is typed into the system, we might want to see all the ways that the password is used within the program. If the use is not limited to an authentication routine, and for instance, the password is written (in plain text) to a file, this might arouse suspicion.

## 4 Examples

To apply the methodology of property-based testing, we take as examples UNIX privileged programs, and consider security properties of these programs. Security properties are easily isolatable from other program properties. In general, that portion of the program which deals with security-related objects (e.g., filenames and user ids) is a significantly small subset of the original program, so slicing is particularly useful in this case. Privileged programs provide us with a large example suite, many of which have widely known flaws. We can both evaluate the ability of the Tester's Assistant in detecting these known flaws, and also determine if our system might find previously unknown flaws. Finally, these examples provide a fairly representative sample of the range of specifications and programs relevant to computer security.

### rdist

Rdist is a program to update identical file systems on distributed hosts. It is executed on a machine which contains the master copy of the files, and it checks the other machines for outdated copies, replacing them as necessary.

Rdist was found to have a flaw that would allow a user to alter the permissions on any file in the system. A user can utilize a race condition in the program to apply the `chmod`[6] operation to a different file then the one rdist is copying the data into.

A truncated slice of rdist with respect to the chmod system call is shown in Figure 4; the behavior of the chmod system call, because it changes permissions in the file system, is a focus of a property-based test. The destination file (`new`) is first created, then its path is checked for correct permissions. Data is read into the file, and then the file is closed. Only at this point are the files permission bits changed. The race condition arises when some other process makes a change to the file before the chmod system call is made.

Although rdist is a very large program, this flaw can be detected with a simple security requirement; no file's permission bits can be changed unless the user owns the file. Standard specifications of the

[6] Chmod changes the file's permission flags in UNIX.

```
recvf(cmd, type)
{
  >>>calculation of the new filename
  >>>   and its protection mode
  if ((f = creat(new, mode)) < 0) {
    if (errno != ENOENT || chkparent(new) < 0 ||
     (f = creat(new, mode)) < 0)
      >>> Error condition
  }
  wrerr = 0;
  for (i = 0; i < size; i += BUFSIZ) {
    int amt = BUFSIZ;
    cp = buf;
    read(rem, cp, amt);
    >>> code to handle erroneous reads
    if (wrerr == 0 && write(f, buf, amt) != amt) {
      wrerr++;
    }
  }
  close(f);
  if (wrerr) {
    return;
  }
  chog(new, owner, group, mode);
}
```

```
chkparent(name)
{
  >>>recursive 20 line function that returns 0 if path
  >>>is valid and writable, -1 otherwise
}

chog(file, owner, group, mode)
{
  if (userid != 0)
    if ((mode & 04000) && strcmp(user, owner) != 0)
      mode &= ~04000;
  >>> calculation of group of the file
  if (userid && gid >= 0) {
    >>> if user not in group
      mode &= ~02000;
  }
  chmod(file, mode);
}
```

Figure 4: *Abbreviated slice of rdist with respect to* `chmod()`*.*

permission-checking system calls would determine when the state predicate `owns(user,file)` is asserted; this is related to system call which changes permission bits to form slicing criteria.

## fingerd

`Fingerd` was the cause of several security breaches. The flaw was caused by an overflow of the buffer which holds some of the input data. Portions of the stack were overwritten, which allowed the user of the program to execute code which had been entered as part of the input argument. Before this flaw was extensively exploited by the Internet Worm [16], fingerd was run setuid root, and so the flaw allowed system penetration. This flaw is a violation of a well-behavedness property, which is found by testing for well-behavedness requirements.

## ftpd

Some versions of `ftpd` contained a flaw which allowed any user to read or write any file on the system. The flaw was caused by an unanticipated path through the program which could bypass the authentication routines. Sufficiently sophisticated dataflow coverage metrics would require this path to be executed in a complete test relative to an authentication/access specification, and so the flaw would have been discovered through property-based testing.

## Scope of the domain

Property-based testing is designed to effectively test single programs for security flaws. It is important to realize where this capability fits into a complete picture of a secure system.

A program is tested against a given set of specifications that describe security flaws. Assuming the testing process does not reveal any flaws, a level of assurance is gained about the software in question. This level of assurance is dependent upon the quality of security specifications.

There are flaws for which it is difficult to write specifications. Consider a general scenario where program P1 provides input data for program P2. P2 is permitted to perform some privileged actions in some restricted way, but the input data that P1 provides is able to cause a security breach in P2. This flaw might have been found in a property-based test of P2, but what if the input language for P2 is a complex programming language in and of itself? It may not then be feasible to test P2 (which is in effect a compiler) for security flaws. Testing, validating, or verifying compilers is a hard research problem for which there are no easy solutions. One approach that property-based testing could offer would be to analyze each individual "program" which is passed from P1 to P2, but this would require building dataflow analysis tools for the new language.

A generalized version of this problem is where a flaw is manifest when programs are run sequentially, each one changing the security state of the system in some minor, harmless way. Thus, each individual step is seen to be free of security flaws (through analysis with property-based testing). However, if the n programs are run sequentially, a larger, harmful change in the security state is brought about. One example of this involving only two programs is as follows: one program changes the password of the user running the program to some fixed string, and the second program sends this string to some other site on a network. In this case the first program might be identified as insecure (although in some instances password-changing programs must be allowed); the second program in itself seems to be quite benign.

To catch this flaw in general, some global reasoning must be done on the whole set of security specifications for a system. This set must be analyzed for flaws which occur only when programs are used in conjunction with each other. A model of the whole system must be created, including the concept of secure and insecure states. Thus, our specification model implemented in the Tester's Assistant is adequate only for particular flaws in privileged programs.

## 5 Architecture of the Tester's Assistant

The five components of the Tester's Assistant are the specification language, the slicer, the dataflow coverage analyzer, the execution monitor, and the test data generator. A human tester would use the system by selecting and creating specifications and selecting some initial test data for the program to be tested. The execution monitor detects when an incorrect execution occurs. If no incorrect execution occurs, slices of the program are examined for their dataflow coverage information, and more test data is generated as a result of this analysis.

All of the components coordinate through a common dataflow program representation. To generate the dataflow representation and construct the instrumentation necessary for coverage analysis and monitoring, the ELI [17] system is used. ELI is a text processing and compiler construction toolkit. In ELI, high-level specifications are converted into high-performance executable translators and compilers. Using ELI, we are able to quickly prototype the Tester's Assistant.

The dataflow representation has been implemented. The slicer is partially implemented: it is operational on multi-procedure programs using most of C's operations and properties, relying in part on algorithms from [10], [2], and [5]. Pointer anti-aliasing will be introduced to make slices more efficient. Currently, the slicer makes worst case assumptions about pointer aliasing, though the **well-behavedness** assumption limits the scope of the aliasing assumptions. Library and system calls are being specified as necessary. Slicing criteria are currently limited to a single variable and/or system call. Technical tasks for the short term are to improve the slicing algorithm and to provide a richer language for expressing slicing criteria. A harder problem to be addressed is to produce templates or other devices for automatically associating complex concepts in the specification (like authentication) to source code.

Below, we describe the issues involved with the other components of the Tester's Assistant, and the benefits gained from including them in the tool set.

### Dataflow testing

Ad hoc testing, the repeated application of arbitrarily selected test data to a program, can only confirm negative properties about the program. The only concrete information that can be gained is that an error has occurred; no positive generalities can be drawn from a successful test. Formal coverage analysis is an attempt to rectify this situation. Coverage analysis based on coverage metrics codifies the portions of the program which is used in an execution of a set of tests.

There is a wide variety of coverage metrics. The simplest coverage metric is statement coverage. In this measure, each program statement is executed at least once in complete code coverage. If the coverage is not complete, an indicator called coverage percentage is used to calculate the ratio of tested code to total code. Code that is not executed by such a test suite is referred to as a "gap" in the coverage.

As Hamlet demonstrates [6], simple coverage metrics can easily result in flawed code going undetected. Hamlet suggests that data-flow coverage approaches are superior. Data-flow coverage is a term that encompasses a variety of coverage schemes based on measuring coverage of sub-paths of a program from the definition of a variable to its uses. Covering def-use paths [8], allows complicated flaws to be exposed.

Computing complete dataflow coverage information becomes inefficient as the program size increases. Information needs to be kept on each individual path through the program; the number of paths grows exponentially with the size of the program. However, when testing for a specific security property encapsulated in a specification, it is only necessary to calculate the coverage metrics on paths which are contained in the slice. This reduction in the space of paths makes dataflow testing feasible in the Tester's Assistant.

### Execution Monitoring

A test run of a program needs an oracle that indicates whether or not the execution produced the correct results. Manual inspection of the output of an execution is infeasible in many situations, and at best error-prone, especially if correct behavior must satisfy numerous security requirements. We create executable oracles derived from the security requirements.[7].

The use of specifications as oracles to test programs is not new: Richardson [13] and Sankar [15] [14] used specifications to generate assertion-checking functions. What is unique about our approach is the relative (small) size of the specifications, and the ability to associate them with a slice of the program. For certain programs, in addition to adding predicate assertions, we track an abstract "state" of the program (for instance, the current uid).

Testing programs with respect to security specifications has a distinctive characteristic as compared with the testing with respect to general properties. Most security specifications capture "safety" properties – i.e., bad things should not happen. In current operating systems, the state can be changed only through invocation of system calls to the kernel, so state checking only need be done at these points in the program.

One method being explored to accomplish security-based execution monitoring is through system call audit trail analysis [7]; traces of system calls are collected by the operating system and analyzed in real time. For many security flaws, this form of execution monitoring is both effective and efficient.

But, using auditing as a basis for testing programs has limitations. Most current auditing systems do not record all parameters of system calls; hence, not all the information about an event is recorded. For instance, how a program modifies a file and a password which is typed in cannot be inferred from the audit trails. In addition, some specifications, like authentications, are hard to check in this manner.

To produce a more effective execution monitor, we add code to the program for the purpose of monitoring the adherence to specifications. Information from the slice can be used to insert monitoring code at exactly those locations which can influence the specification assertion.

[7] In the case of oracle specifications, a precise specification is needed; this is in contrast to the use of specifications in slicing, in which case only vague specifications are sufficient.

### Test data generation

So far, our discussion of property-based testing has assumed that a human generates test data. But, in part, the process for test data generation can be automated. Using a slice as an intermediary between a specification and test data generation is the key.

In symbolic evaluation [3], input data is represented by variables, and execution is controlled by a meta-interpreter for the language. As computations are made, and the path of execution traverses conditional expressions and loops, the current values of program variables are kept as a set of constraint equations characterizing the relative values of the variables. These constraint equations can be solved, and some attempt made to provide closed form representations of output variables in terms of input variables; input data can then be selected which produces "interesting" output values. Often interval arithmetic is used in these computations. One limitation is that the constraint equations can easily become too difficult to solve automatically, requiring assistance from either a human or a mechanical theorem prover. Additionally, loop invariants and assertions often need to be provided in order to make simplifying inferences about the behavior of loops.

However, symbolic evaluation becomes feasible when conditional expressions and loops are eliminated [4]. If the results of the coverage analysis reveal unexecuted paths, each path can be given to the symbolic evaluator for test data generation. Since for a path through the program, the results of all conditional expressions and loop predicates are fixed, many of the computational difficulties are eliminated.

## 6 Discussion

We are investigating the technique of property-based testing as a way to address security testing of software. In property-based testing, specifications are used to slice a program to an executable subset relevant to the specification. Manual methods are used to derive test data for the slice. The specifications are also used as an oracle, when the data is applied to the slice. Dataflow coverage metrics measure the effectiveness of testing, in addition to focussing attention on suspicious sections of code that deserve a closer look. A testing environment, the Tester's Assistant, is being developed to evaluate the effectiveness of property-based testing for C programs.

Property-based testing is being applied to UNIX utility programs, which are the major source of security problems in UNIX. Although we have not captured all security-relevant behavior of UNIX in our specifications, the specifications we have written to represent integrity requirements have been surprisingly easy to produce and are very concise – typically just a few lines of predicate logic. Furthermore, the reduction in program size due to slicing is substantial. In some instances, analysis of system audit trails can be used in execution monitoring, reducing the amount of program instrumentation necessary to test.

Work is continuing on the development of the Tester's Assistant. We are extending the set of examples the Tester's Assistant can be applied to, including, extending the slicer to handle concurrent programs. We are also studying other applications (e.g., safety properties) where the specifications might be easy to write and the benefits of slicing are substantial.

## References


[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical report, Lawrence Livermore Laboratory, April 1976. available as NBSIR 76-1041 from National Technical Information Service.

[2] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. Technical Report 1128, Department of Computer Science, University of Wisconsin-Madison, December 1992.

[3] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, 1975.

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

[5] Michael Ernst. Program slicing using the value dependance graph. To appear in the *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[6] Richard Hamlet. Testing programs to detect malicious faults. In *Proceedings of the IFIP Working Conference on Dependable Computing*, pages 162–169, February 1991.

[7] Calvin Ko, Karl Levitt, and George Fink. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Tenth Annual Computer Security Applications Conference*, December 1994.

[8] Bogdan Korel and Janusz Laski. STAD - a system for testing and debugging: User perspective. In *Second Workshop on Software Testing, Verification, and Analysis*, 1888.

[9] Panas E. Livadas and Stephen Croll. The C-Ghinsu tool. Technical Report SERC-TR-55-F, University of Florida, December 1991.

[10] Panas E. Livadas and Stephen Croll. Program slicing. Technical Report SERC-TR-61-F, University of Florida, October 1992.

[11] Raymond Waiman Lo. *Static Analysis of Programs with Application to Malicious Code Detection.* PhD thesis, University of California, Davis, 1992.

[12] Robin R. Lutz. Targeting safety-related errors during software requirements analysis. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–105, December 1993.

[13] Debra J. Richardson Owen O'Malley and Cindy Tittle. Approaches to specification-based testing. In *Proceedings of the First ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification(TAV3)*, pages 86–96, December 1989.

[14] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[15] S. Sankar and R. Hayes. Adl — an interface definition language for specifying and testing software. Technical Report CMU-CS-94-WIDL-1, Carnegie-Mellon University, January 1994.

[16] Eugene H. Spafford. Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, 1989.

[17] William Waite et al. Eli system manuals. Unpublished Manuals, 1993.

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