Towards a Property-based Testing Environment with Applications to Security-Critical Software [*]

George Fink Calvin Ko Myla Archer
Karl Levitt
Department of Computer Science, University of California, Davis
gfink@cs.ucdavis.edu

Abstract

We consider an approach to testing that combines white-box and black-box techniques. Black-box testing is used for testing a program's effects against its specification. White-box testing is essential if subtle implementation errors are to be identified, e.g., errors due to race conditions. Full white-box testing is a large task. However, for many properties, only a small portion of the program is relevant — hence property-based testing has the potential to substantially simplify much of the testing work. The portion of a program that relates to a given property can be identified through slicing. We describe the ongoing development of a Tester's Assistant, which in the long term, will include a specification-driven slicer for C programs, a test data generator, a coverage analyzer, and an execution monitor. The slicer and execution monitor are described in this paper, and applications to Unix security are indicated. Security is an important application of property-based testing because of the subtle undetected security errors in delivered operating systems. It is also a promising application because of the (unexpectedly) concise specifications that capture most security requirements, and because of the operating system support for execution monitoring.

1 Introduction

White-box testing involves the generation of test data based on program code. Black-box testing, traditionally, has involved the generation of test data based on program specifications. White-box testing is essential if subtle errors are suspected, especially those based on race conditions, internal program state, or "windows" where the program is exposed/vulnerable to actions that could cause failure. Block-box testing, on the other hand, helps focus testing activities on the expected behavior of the program. The general opinion of the testing community is that the judicious combinations of these techniques is needed. This paper explores the combinations of black-box and white-box testing, but employs specifications, the basis for black-box testing, to slice a program to yield the subset of the program relevant to the properties required by the specifications. We call this technique property-based testing.

Not surprisingly, the program properties which are to be to the focus of the testing effort depends on the applications; furthermore, a given application might involve numerous properties.

For software intended to implement a secure operating system, for example, the critical properties are those that related to preservation of the security state of the system to the unauthorized release of information, or to the preservation of the integrity of files. The portion of the program which addresses these critical properties is often significantly smaller than the whole pro-

[*]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.

Report Documentation Page		*Form Approved OMB No. 0704-0188*

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**
4. TITLE AND SUBTITLE **Towards a Property-based Testing Environment with Applications to Security-Critical Software**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California (Davis),Department of Computer Science,1 Shields Avenue /2063 Kemper Hall,Davis,CA,95616**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**10**	

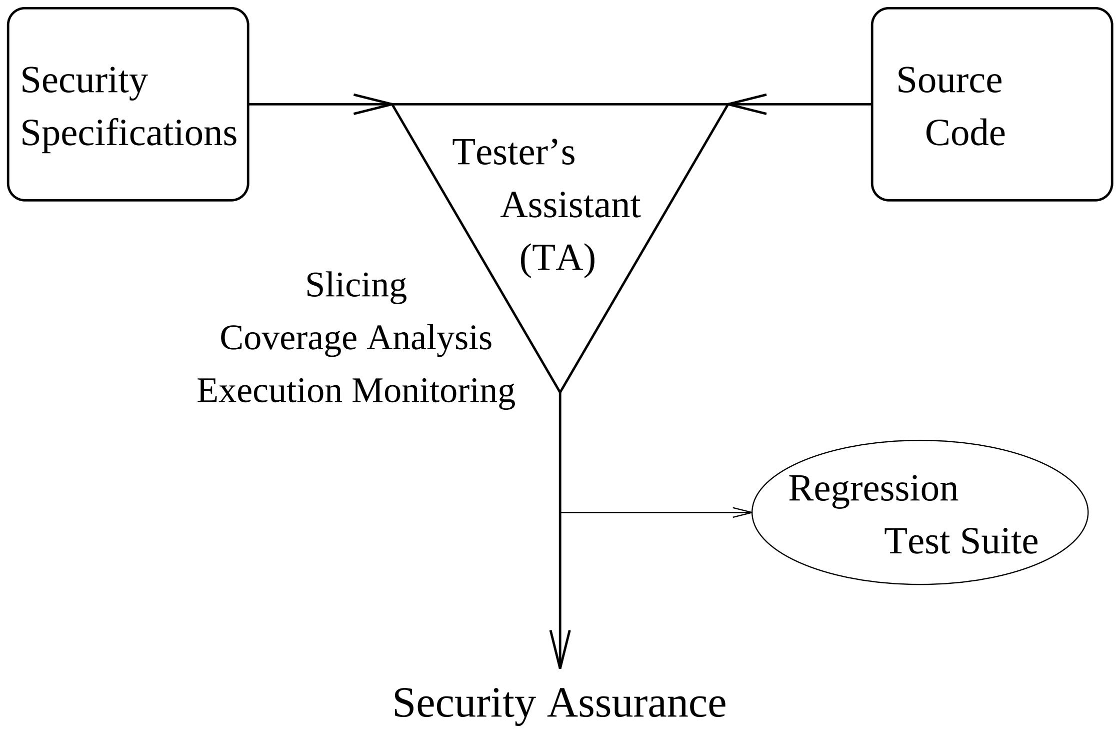

Figure 1: Overview of the Tester's Assistant

gram; therefore property-based testing is a more efficient testing scheme than classical white-box or black-box methods.

Specifications are necessary in order to describe the properties which drive the testing process. We have found that it is relatively easy to specify most security-related properties. It is not necessary, in general, to provide a full specification of the program's behavior, as in the work described in [OT89], and as we demonstrate below for security specifications. In some instances, no specification of the program itself is necessary for detecting security-related flaws. Certain generic flaws [Spa92] been shown to be the cause of many security problems. These generic flaws can be captured in specifications for use in property-based testing. By not requiring full specifications, property-based testing is relatively easy to apply to systems that do not typically come with full specifications.

There are three main uses of specifications in testing. The first, slicing, identifies the subset of the program that governs behavior relating to a specification. The second, execution monitoring, uses a specification to construct oracles which detect when an execution exhibits behavior violating the specification. The third, the use of specifications for test data generation, the traditional use of specifications in testing, is not emphasized in property-based testing, as the role of a specification in this case is mostly subsumed by its role in slicing.

Specifications are useful for all these purposes, whether or not the specifications are written in a formal language. However, requiring the partial specifications to be written in a formal language enables the use of automatic tools to process the specification.

We are developing an environment, the Tester's Assistant (See Figure 1) for property-based testing, to demonstrate the usefulness of the methodology. The Testers Assistant has modules for slicing, dataflow testing, test data generation, and execution monitoring[1]. The primary examples to which the Tester's Assistant is

[1]A side effect of the use of the Tester's Assistant is a test suite that can later be used for regression testing. Through coverage monitoring, test data can explicitly be associated with paths, which can also aid in regression testing

being applied are UNIX utility programs, which are all written in C (e.g., login and rdist), all of which are security critical. Our techniques rediscovered the known security flaws in several utility programs, including Rdist; to date we have discovered no new security flaws in UNIX although the testing of other programs is ongoing. In addition to testing sequential programs, the Tester's Assistant is being designed to address concurrent and distributed systems as well. Many of the known security flaws appear in concurrent or distributed programs on multi-tasking UNIX machines.

Section 2 outlines the security model used in property specifications. Section 3 defines slicing and discusses the issues associated with it. Section 4 describes how specifications can be used to monitor program execution. Section 5 applies the Tester's Assistant methodology to several example programs. Section 6 sketches out the Tester's Assistant implementation and reports on current progress. Finally, Section 7 spells out some conclusions and future plans.

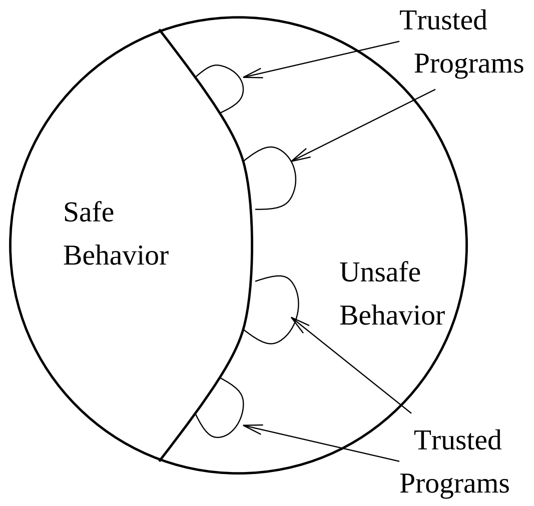

Figure 2: Security requirements distinguish programs which behave safely from those who do not. For some tasks, it is necessary to breach a security requirement; these tasks are assigned to "trusted" programs which can transcend the security requirements arbitrarily, but are expected not to exhibit unsafe behavior.

2 Security Specifications

The Tester's Assistant is heavily reliant upon specifications and, in particular, an effective model for security specifications. In this section, a model for UNIX security is introduced, and examples of the various kinds of specifications are given.

In UNIX, some programs have privileges beyond those needed to complete their job. (Figure 2) This happens because the granularity of the access control mechanism may not be fine enough. To determine that these programs do not indeed exceed their intended privilege, we test them with respect to their security specifications, which we provide. For example, in UNIX, the "`/bin/passwd`" program is a setuid root program, which means the program will run with superuser privileges no matter who executes the program. Therefore, `/bin/passwd` can potentially do anything in a UNIX system (e.g., plant a trojan horse, kill any process, shut down the system). We want to assure that the program is appropriately restricted in what it can do in the system. Conversely, we want to ensure that crucial activities such as user authentications take place. Without specifications, there is no assurance that the absence of authentication routines will be noticed during the testing process.

A motivating factor behind the model is the desire to require only minimal specifications to be provided. There are several reasons for this: specifications are hard to write; if large, they are hard to understand; it is difficult to retrofit specifications onto pre-existing code; finally, it is desirable to test programs written by others, for which you do not possess full specifications.

Our model divides security specifications into three categories, plus an additional category of safety specifications; these categories are not immutable, but they provide a convenient framework from which to define the specifications. The primitives of the specifications are users, objects, and access rights and restrictions.

The first category describes general system requirements. These 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[2]. However, it is obvious that in some instances this requirement must be violated; when some 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 strong barrier between programs and the system.

The second category of specifications defines the interface between programs and the system. Within the context of the interface, programs are allowed to violate first category specifications. Each system call that is related to security has a security-related specification. Some system calls have pre-conditions: 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 shown below:

- pre-condition: `authenticated(uid)`
- `setuid(uid)`
- post-condition: `permission-granted(uid)`

This specification says that the user should be authenticated before the setuid system call is made, and that after the system call is made, permissions have been granted for that user. The post-condition is information that is added to the security state after execution, and can help validate (or invalidate) other requirements and pre-conditions.

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 are specified. The correctness of the libraries can be established in an independent test run. An important set of library routines to define for the security model is the password library, a set of routines which define a standard interface to the password file.

[2]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 UNIX security policy allows read access.

The third category is program-specific specifications. Such a specification details when a program is allowed to circumvent the restrictions of the first and second category of specifications. For example, to allow users to change their passwords or new accounts to be created, special permission must be given in order 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, that instance program can write only to the password file and only the entry corresponding to the password of that user.

The final category of specification describes safety properties, which have been enumerated many times [Lut93] [Spa92]. Safety properties cover common programming mistakes which can cause flaws. These properties are necessary to handle the **well-behavedness** property, described in the next section.

3 Slicing for Security Properties

Slicing is an abstraction mechanism in which code that might influence the value of a given variable or set of variables is extracted from the full source code of a program. Weiser [Wei84] originally implemented slicing for FORTRAN programs. A criterion for the slice is selected. In the simplest scenario, a criterion is a variable and a location in the program. There are two essential characteristics that are required of a slice with respect to such a criterion: it must be executable, and for the same input values, the variable must have identical values at the corresponding location in the slice and the original program. More complex criteria can involve multiple variables, multiple locations, and include the behavior of the program after arriving at the location.

A slice of a program is produced by generating control and data flow graphs of the program, then finding a closure with respect to the "depends upon" property, starting with the nodes that correspond to the slicing criteria. For simple imperative languages, generation of the flow graphs and the resultant slice is relatively easy. More complicated language constructs such as procedures, gotos, and pointers require alterations in the slicing algorithm. Interprocedural slicing [LC92] and slicing for programs with arbitrary control flow [BH92] have been studied. In order to slice C code, the problems of pointers and pointer aliasing need to be addressed.

Weiser noted that slices are implicitly used in debugging [Wei82]. In the Tester's Assistant, the uses of slices are made explicit. Slicing impacts the other testing algorithms (e.g., coverage) by enabling an algorithm to be utilized on a subset of the original program (the slice), giving an increase in efficiency. For example, given that a slice possesses the two essential characteristics, code instrumentation in order to measure coverage with respect to a specification need only be added to the subprogram designated by the slice. Alternatively, the slice could be compiled and executed in isolation from the rest of the program.

Implicitly, by using slices as the base unit for testing programs for security holes, we are assuming that testing a slice is equivalent to testing the whole program. The equivalency of a slice to a program with respect to some narrow criterion does not necessarily imply the equivalency of test results, although in the Tester's Assistant we have attempted to bridge the gap by relying on the completeness of the model defined by the security specifications to describe the properties of interest. 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.

An aspect of a program which potentially jeopardizes the correctness of slices is unexpected side-effects which make the flow graphs very difficult or impossible to calculate. This difficulty arises when slicing C programs because the language allows virtually unlimited pointer arithmetic and also allows many forms of pointer (and function) aliasing.

To address this, we make the powerful 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, try to refer to a different object in memory. A program is well-behaved if all of its pointers are well-behaved. Lo [Lo92] showed that static analysis can establish the well-behavedness property in many cases. The well-behavedness property can also be affirmed through testing with respect to appropriate safety specifications. Unfortunately, much of the power of C as a language derives from being able to do ill-behaved operations. However, in many cases, this ill-behavedness is used in very specific ways in 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.

Using a slice as an intermediary between a specification and coverage monitoring and test data generation changes the nature of the latter two tasks. Using a specification to generate a slice changes the way it is used to generate test data. Because slices tend to be small, it is feasible to use symbolic evaluation to create test data for most paths in a slice. This is an indirect use of a specification in test data generation. Specifications are also used in test data generation to identify partitions and interesting data values (e.g. `/etc/passwd` for filenames) in the input domains from which test data should be drawn. However, by definition, a slice contains all code relevant to a given property, so coverage measures are only necessary for paths contained in the slice.

Slicing typically 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 variables at different points in the program in a cumulative way, i.e., the union of the individual slices. In other cases, the desired slice is the intersection of slices, or is decided by a more complex boolean formula. For example, when attempting to slice with respect to the `setuid` specification given above, the slice will contain all data paths which traverse the `setuid` but do not traverse the authentication routine. The specification of `setuid` indicates that paths which traverse both are secure, and thus do not need further testing. Determining slices in this manner is called *dicing* [LC91].

The current version of the Tester's Assistant slicer does a data-flow breakdown of the program, and can find simple slices. Significantly, interprocedural and pointer analysis are not complete, so slices do not cross procedural boundaries and have to make limiting assumptions about pointer behavior. The primary cost in slicing is producing the data-flow representation. The largest example to which the slicer has been applied is the `rdist` server, which required over three thousand data-flow nodes and took between seven and eight seconds to execute.

4 Execution Monitoring

In this section we present our approach to monitoring the results of test runs to determine if the results are as expected. We show that for the testing of trusted programs in a UNIX environment (i.e., programs that typically run outside of the kernel, but use kernel services), the kernel, through its audit services, provides data that is useful in the analysis of test runs.

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. An executable oracle derived from the security requirements is desirable.

The use of specifications as oracles to test programs is not new: Richardson [OT89] and Sankar [SH94] [San89] 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. The logs of all system calls made by the program contains data pertinent to the results of the test. Therefore, system audit trails, which record all the system calls to the kernel, can be used as a basis for checking whether a program conforms to its security specifications.

A significant advantage of using auditing for testing programs is that auditing is a service available in most current operating systems (e.g., Sun Solaris). For many utility programs, we can directly check the results of a test without instrumenting the program, thus obviating the need to insert assertion-checking functions into the program source code. Moreover, auditing is done outside of the tested program, guaranteeing that neither the program nor its test data/results are tampered with. Auditing has been used for intrusion detection systems, where audit trails are analyzed in real time to detect attacks on computer systems.

In our scheme, the program in question is invoked by the test driver with appropriate test input. (See Figure 3.) With auditing enabled properly, actions of the program are recorded as

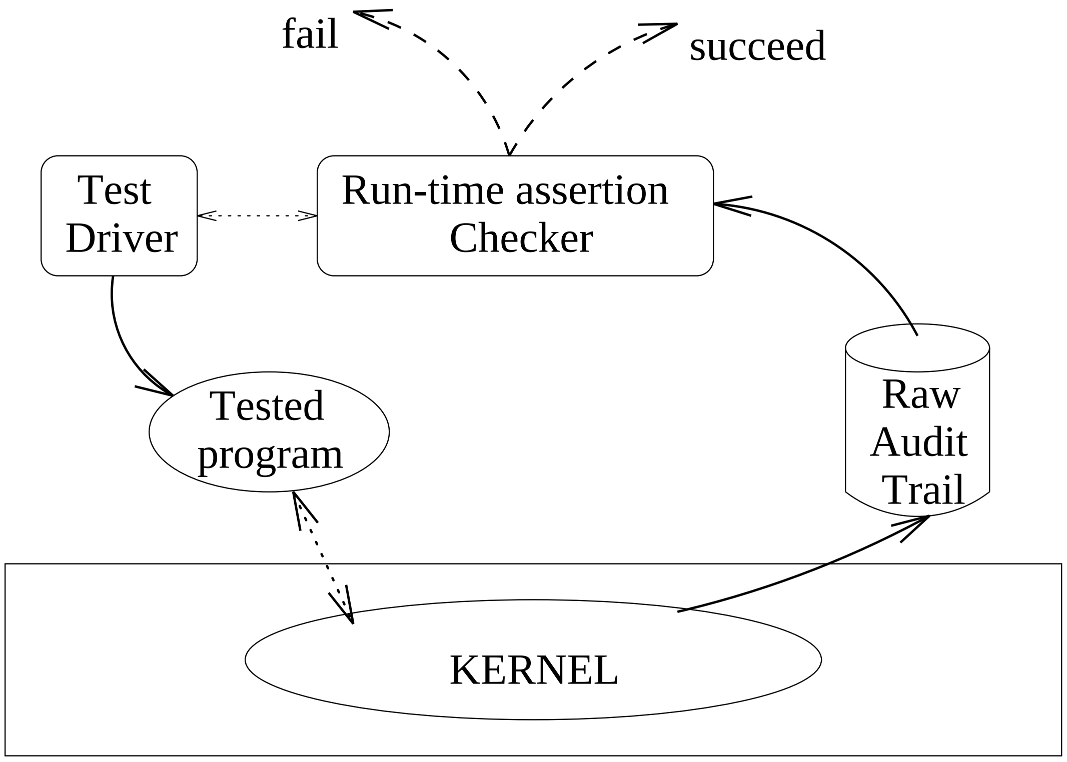

Figure 3: Architecture of the execution monitor.

audit trails and saved into files[3].

The audit trails, which consist of a sequence of audit records arranged in temporal order, are filtered and preprocessed by the run-time assertion checker to yield records associated with the execution of the program by the run-time assertion checker. The checker then checks the results of the test by matching the audit records against audit-trail rules, which are derived from the security specifications describe in Section 2.

In Sun BSM audit trails, an audit record signals an occurrence of an event, which is identified by the event ID in the record. In general, an action/operation referred to the security specification may correspond to one or more events. For instance, the following events all indicate a write to a file: `AUD_OPEN_W`, `AUD_OPEN_RW`, `AUD_OPEN_TW`, `AUD_OPEN_TRW`. Therefore, a regular expression of events is used to represent a high level operation in the security specification. If the acceptability of an action with respect to a security requirement depends on the current state of the program (e.g., authentication must precede the setuid call), state changes associated with program variables from which state changes of the program can be inferred are also included in the audit-trail results.

Even large programs have unexpectedly simple behavior with respect to security, and offer conformance to security properties which can be checked from audit trails. One example is the rdist program, which has a vulnerability that enables a user to change the permission bits of any file. However, from a security perspective, it is clear that `rdist` should not change the permissions of any files not owned by the process which calls `rdist`, a property that is easily stated and checked at runtime just by analysis of audit trails. Another example is the infamous sendmail program, which has been the source of many security problems[4]. The basic function of Send-

[3]In Sun OS, auditing is carried out by an audit daemon, which collects events defined by an audit specification and records then as audit trails.

[4]One of the vulnerabilities, employed by the internet worm, enables any users in a remote host to obtain a root shell in the host running sendmail. The most recently discovered vulnerability enables an attacker to cause root

mail is to deliver mail messages to users. In a networked environment, it also has to route mail to destination hosts. Sendmail should only have append access to a user's mailbox file, and the ability to execute a program on behalf of a user when a mail message arrives (e.g., vacation programs). It should not write to other files (e.g., the password file, system programs). Again, it is easy to check whether sendmail exceeds the expected behavior from the audit trails, thus enabling security testing.

On the other hand, 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 cannot be inferred from the audit trails. In addition, some specifications, like authentications, are hard to check in this manner.

Fortunately, since in the course of slicing we are parsing and regenerating the source of the program, it is simple to add code to the program for the purpose of monitoring the adherence to specifications. Additionally, information from the slice can be used to insert monitoring code at exactly those locations which can influence the specification assertion.

5 Examples

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

To illustrate, Figure 4 is a slice of the MINIX [Tan87] login program with respect to the `setuid` system call. The original program contains 337 lines, the slice only 20, demonstrating the effectiveness of slicing in this case[5].

The mapping of the abstract concept of authentication to source code in the MINIX login program is an example of the difficulty in general of mapping specifications to appropriate slicing criteria. While this slice can be produced by slicing the code with respect to the setuid system call, the additional information in the specification that deals with authentication is necessary in order to accurately produce an oracle that checks the adherence to the `setuid` specification.

Two other interesting examples of property-based testing are the UNIX utility `rdist`, and the UNIX network service `fingerd`. `Rdist` was found to have a flaw that would allow a user to alter the permissions on any file in the system. Although `rdist` is a very large program, its flaw can be detected with a simple security requirement: that no file be altered unless its file name is entered as input to the program.

`Fingerd` was the cause of many security breaches. The flaw that caused the security breach was found to be a string overflow error. This is a violation of a well-behavedness property, which will be found by testing for well-behavedness requirements.

6 Architecture of the Tester's Assistant

The five components of the Tester's Assistant are the specification language, the slicer, the dataflow coverage instrumenter, the execution monitor, and the test data generator. A human tester would use the system by selecting specifications to test against, and some initial test data for the program to be tested. The execution monitor detects when an incorrect execution

to execute any program he wants, perhaps a program with a Trojan Horse.

[5]Line counts generated using the UNIX `wc` command. The loop exit, a call to `exec()`, is not shown.

```
for (;;) {
      bad = 0;

      write(1, "login: ", 7);
      n = read(0, name, 30);

      if ((pwd = getpwnam(name)) == NULL) bad++;

      if (bad || strlen(pwd->pw_passwd) != 0) {
              write(1, "Password: ", 10);

              n = read(0, password, 30);

              if (bad && crypt(password, "aaaa") ||
                  strcmp(pwd->pw_passwd, crypt(password, pwd->pw_passwd))) {
                      write(1, "Login incorrect\n", 16);
                      continue;
              }
      }
setuid(pwd->pw_uid);
}
```

Figure 4: Slice of MINIX login with respect to `setuid()`.

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 data-flow program representation. To generate the dataflow representation and correct the instrumentation necessary for coverage analysis and monitoring, the ELI [W$^+$93] 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 single-procedure programs using most of C's operations and properties. Inter-procedural slices generated by algorithms from [LC92] will greatly increase the number of programs we can analyze. 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, technical tasks 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.

7 Discussion

We are investigating the technique of property-based testing, whereby specifications are used to slice a program to an executable subset relevant to the specification. Traditional methods are used to derive test data for the slice. The specifications are reused as an oracle, when the data is applied to the slice. A testing environment, the Tester's Assistant, is being developed to evaluate the effectiveness of property-based testing for C programs.

Our initial application has been UNIX utility programs, a 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 with respect to security properties. The reduction in program size due to slicing has proved to be substantial. With the use of system audit trails in execution monitoring, results of many test runs can be analyzed without instrumenting the tested program, which simplifies the analysis of test data.

At the current moment, work is continuing on the development of the Tester's Assistant. We are broadening the set of examples the Tester's Assistant can and will be applied to, including, in the farther future, extending the slicer to handle concurrent constructs. In the meantime, 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

[BH92] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. Department of Computer Science, University of Wisconsin-Madison, December 1992.

[LC91] Panas E. Livadas and Stephen Croll. The C-Ghinsu tool. Technical Report SERC-TR-55-F, University of Florida, December 1991.

[LC92] Panas E. Livadas and Stephen Croll. Program slicing. Technical Report SERC-TR-61-F, University of Florida, October 1992.

[Lo92] Raymond Waiman Lo. *Static Analysis of Programs with Application to Malicious Code Detection.* PhD thesis, University of California, Davis, 1992.

[Lut93] 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.

[OT89] 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.

[San89] 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.

[SH94] 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.

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

[Tan87] Andrew S. Tanenbaum. *Operating Systems – Design and Implementation.* Prentice–Hall, 1987.

[W$^+$93] William Waite et al. Eli system manuals. Unpublished Manuals, 1993.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, July 1982.

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