SANDIA REPORT

SAND2004-5610
Unlimited Release
Printed November 2004

Some Attributes of a Language for Property-Based Testing

Matt Bishop and Vicentiu Neagoe
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-5610
Unlimited Release
Printed November 2004

Some Attributes of a Language for Property-Based Testing

Matt Bishop and Vicentiu Neagoe
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562

Abstract

Property-based testing is a testing technique that evaluates executions of a program. The method checks that specifications, called properties, hold throughout the execution of the program. TASpec is a language used to specify these properties. This paper compares some attributes of the language with the specification patterns used for model-checking languages, and then presents some descriptions of properties that can be used to detect common security flaws in programs. This report describes the results of a one year research project at the University of California, Davis, which was funded by a University Collaboration LDRD entitled “Property-based Testing for Cyber Security Assurance”.

This page intentionally left blank

1 Introduction

Property-based testing is a technique for testing the security of programs. Recall that *secure* means conforming to a security policy. The analyst doing the testing first specifies the properties she wishes the program to conform to. The program is then instrumented based on the specification of the properties. Consider the execution of the program as a state machine, where as each instruction is executed, the state of the program changes. Relevant changes affect only those portions of the state used in the properties. The instrumentation produces output whenever such a change occurs. The program is then executed under control of another program called the test execution monitor (TEM). The TEM is given the properties, and whenever the execution enters a state forbidden by the specified by the properties, it reports a security error.

These properties are written in a little language (called TASPEC) with constructs designed to aid testing. These include location specifiers (which specify specific places in the program where the properties hold, or where the state of the executing program will change), assertion statements (which assert that a certain property now holds), retraction statements (which assert that a certain property no longer holds), and temporal relationship operators (which specify whether something occurs before or after something else, or should occur and hold for the rest of the execution).

This report first compares TASpec to other specification languages. We then examine some of the temporal operators used in TASpec in considerable detail. We analyze their precise meaning and show how to capture some of the notions in model checking and temporal logic languages using TASpec constructs. We conclude by presenting specifications of properties of programs that attackers commonly exploit in order to compromise the program (and, usually, the system on which the program runs).

A word about motivation will clarify the goals of this report. The software life cycle, in terms of assurance, is usually described using the Waterfall Life Cycle Model. The relevant stages of that model are:

1. Requirements definition and analysis, in which the specifications of the program or system are created and validated;
2. Design, during which the program or system is architected, and the design is validated;
3. Implementation, in which the program or system is created and tested;
4. Integration and system testing, in which a set of programs are brought together and their union is tested and validated; and
5. Operation and maintenance, in which the program or system is deployed and used.

TASpec fits into the life cycle at steps 3 and 4, because it is a language tied to the implementation of the program or system. However, the properties it must validate are often the same as, or derived from, properties that the design must meet. The design properties may be stated using a model checking language such as LTL. This naturally leads to the question of whether TASpec can describe the properties that LTL can

describe, although at an implementation level. This is the reason for the analysis of the operators in TASpec.

That said, there are vulnerabilities specific to an implementation that may have no counterpart to design flaws. For example, buffer overflows arise from a failure to check bounds. Models of systems and software usually do not have states in which the failure to check bounds causes a transition, unless that failure is necessary for some reason. The model is at a level of abstraction in which this detail of checking is not relevant, and so is omitted. But it cannot be ignored at the implementation level. Hence, TASpec must be able to express these properties.

The next section reviews TASpec very briefly to provide background. The third section discusses the temporal operators of TASpec with an emphasis on aspects found in LTL, a model checking language. The fourth presents some common implementation vulnerabilities and the TASpec properties that detect them

2 TASpec and Property-Based Testing

The goal of property-based testing is to validate that a program satisfies a set of specifications, called "properties". The program to be validated is called the "target program."

As with formal methods, we do not discuss the derivation of requirements or, from them, specifications. We simply assume that the specifications are known. We also assume they are written in a low-level specification language (called "TASpec") that describes the specifications in terms of the program being validated. We distinguish this type of specification from the higher-level specifications of formal methods by calling the former "properties."

The first step in property-based testing is *instrumentation*. The property file, which contains the properties, is analyzed to determine which variables and functions will affect the properties. A program called the *slicer* eliminates all code in the target program that does not affect the properties. After slicing, the only paths of control and of data flow in the program are those that could affect the program.[1] The resulting program satisfies the properties if, and only if, the original program satisfies the properties. Next, a second program called the *instrumenter* adds instructions to the target program to emit special directives describing changes of program state whenever a change of state occurs that could affect the desired properties. For example, suppose the property were "x > 5". The following fragment

```
if (y > 6) then x := 3; else x := 6
```

would be instrumented to output the value of x after each part of the statement:

[1] This step can be omitted if the resulting reduction in program size is too small. For example, if the target program is in C, programs can be reduced in size by 80%., so slicing is worthwhile. If the target program is in C++ or Java, the reduction in size is typically much smaller.

```
if (y > 6) then begin
     x = 3; print "assert x = 3";
end else begin
     x = 6; print "assert x = 6";
end
```

The directives will be saved in a file called the "state file" for later analysis.

The next step is *execution*. The program is executed with appropriate test data. Ideally, all paths of execution should be tested. In practice, this number is too large, so some fraction of those paths are executed. Failure to test all paths means that some flows of control have not been validated. This illustrates the difference between validation and verification (in which *all* paths would be shown to be correct) as well as the need for careful test data generation. The inputs to each run are saved with the corresponding state file.

The final step is *analysis*. A third program, called the *test execution monitor* or TEM, takes the property file and the state file, and tests whether the directives show that the properties have been violated. If so, the precise property violated, and the location at which the property was violated, will be printed.

These steps differ from those for formal verification in two key ways. First, formal verification is primarily an *a priori* technique for developing correct code, although it can be used to prove an existing program correct. Property-based testing is strictly an *a posteriori* testing technique. It requires an existing program to use. Secondly, the focus of property-based testing is on the implementation rather than the higher-level layers of abstraction such as design. The properties are written in a little language similar to that of the target program. In formal verification, design is to be verified as well as implementation. So the language of specification is more abstract.

2.1 How TASpec Works

A brief description of the mechanics of TASpec and the TEM will be helpful in what follows. When the instrumented target program emits directives describing changes of program state, those changes have the form of assertions or retractions of particular predicates (called *facts*). The desired properties are expressed in terms of these predicates. As each assertion is emitted, the predicate being asserted is added to a database; as each retraction is emitted, the predicate being retracted is deleted from that database. At each step, the TEM ensures that the facts in the database do not violate any of the properties.

Basically, the state of the program execution is encapsulated in the set of facts in the database.

3 TASpec and Model Checking Languages

Our first question is the relationship between the language constructs in TASpec and the logic languages used to do model checking, as discussed earlier. For our purposes, we focus on the temporal operators.

3.1 Taxonomy

We use the taxonomy described in [3] to categorize TASpec as a hybrid approach between a history-based and a state-based specification language.

State-based specification has been traditionally used for sequential programs while history-based specification with temporal logic has been traditionally used for concurrent programs.

In history-based specification, time can be either linear or branching. Linear time makes assertions about paths and at each moment, there is only one possible future. Branching time has a treelike nature. Assertions are made about states and at each moment time may be split into alternate courses representing different possible futures [2]. Branching time views events as concurrent "alternative universes" [1, p. 954].

TASpec implements a form of "history" that describes preconditions such as "predicate A must be true before predicate B becomes true". It does not keep track of events that happened in the past.

TASpec is a state-based specification language that focuses on functional requirements describing what the software is expected to do. Aside from the state-based paradigm, it borrows temporal logic from history-based specification languages. While the temporal operators do not increase its capability to express properties of programs, the temporal logic does allow some security properties to be expressed easily and clearly. History-based specification languages were designed for dealing with concurrent systems and systems that simulate real-time. TASpec is unique in using temporal logic because it deals with single threaded programs, and therefore linear time. TASpec fits into the discrete time paradigm because each state represents a discrete point in time.

If not designed carefully, specification languages can be cumbersome for expressing certain properties. Temporal logic does not enhance ease of expression when dealing with single threaded programs. It may make coding specifications easier and clearer, therefore reducing greatly the chances of introducing error in a specification. But even with temporal operators, simple specifications can require complex expressions. For example, consider a simple ordering property for moving an elevator. Between the time an elevator is summoned to a floor, and the time it opens its doors at that floor, the elevator can arrive at that floor at most twice [4]. This simple property requires "six levels of operator nesting in linear temporal logic" [3].

But without temporal operators, expressing temporal relationships between states and events becomes unnatural. Extra variables need to be added to indicate whether certain events happened, thereby making writing the specification more complex. Z is an example of a commonly used state-based specification language that lacks temporal operators.

3.2 Branching and Linear Time

Branching and linear time has more to do with how a person views reality. In the case of TASpec, we are only handling the case of single threaded programs, so the discussion about concurrent programs in [Lamport80] does not apply. We think the concept of linear and branching temporal logic can be applied in the following way. If a program has all information which it needs to run, before the program starts (such as a calculation intensive program), we consider that linear time logic is the more appropriate paradigm because we have all the information before the program starts to determine what the final result will be. If a program takes external input as it is running, we consider branching logic to be more appropriate because at any input state in the program, the next state depends upon the external input that may not be known when the program started.

3.3 Time and before and until Temporal Operators

In specification, there is a notion of "strong" and "weak" versions of the *before* and *until* temporal operators. Let **A** and **B** be states of the program. For the strong version of *before*, **A** *before* **B** means that if the execution enters state **A**, then the execution will enter state B at some point before the execution terminates. If the execution enters state **A** and thereafter never enters state **B**, then the expression is false. The expression becomes true only if state **B** is entered, and state **A** was entered before state **B** was entered. By comparison, for the weak version of *before*, **A** *before* **B** means that if the execution entered state **A**, it may or may not enter state **B**. As soon as state **A** is entered, the expression becomes true whether or not state **B** is entered subsequently. This expression is false if state **B** is entered before state **A** is entered.

However, the fundamental difference between checking specifications in TASpec and in logic creates a problem with mapping operators. Consider the *before* operator in (most) temporal logics. If **A** happened in the past, but is no longer true when **B**, then **A** *before* **B** would be true. Now consider the same operator in TASpec, and recall TASpec considers states **A** and **B** as facts in a database. These facts may be asserted (added to the database of current facts) or retracted (removed from the database of current facts). All checking is done over the set of current facts in the database. This means that if state **A** is entered, then left, then state **B** is entered, there is no indication that when state **B** is entered (and the fact corresponding to **B** is in the database), that state **A** held earlier (because as state **A** holds no longer, the fact corresponding to state **A** has been retracted from the database and is not there).

The TASpec interpretation of the *before* temporal operator actually states a precondition. So, **A** before **B** means "fact **A** is asserted before fact **B** is asserted, and must be asserted when **B** is asserted". Because multiple assertions are allowed, this definition needs to be made more precise. With multiple assertions, it is possible that **A** could be retracted in the same event that asserts **B**, in which case **A** *before* **B** would be false. So we should say that **A** must be asserted when **B** is first asserted. This means that the TASpec equivalent of "strong before" is:

```
A implies ((A before B) and eventually B)
```

But what if fact **A** needs to remain asserted throughout the whole time that **B** is asserted? For example, for the fact Authenticated(user) (meaning the user is authenticated) and the fact LoggedIn(user) (meaning the user has logged in), we want the property that Authenticated(user) is asserted before LoggedIn(user) is asserted, and Authenticated(user) remains asserted while LoggedIn(user) is asserted. How can we express that? We might use two properties to express this:

```
A before B
(A and B) until (not B)
```

In terms of strength, the TASpec *before* operator is weak, and the TASpec *until* operator is strong. The reason for this choice is simply that the security properties found so far were easier to express using this arrangement. An alternate arrangement would work equally well.

3.4 Expressibility

There is a tension between how easily a property can be expressed and how powerful the language is. It is the same tension that is found between high level and low level programming languages. Programming in a high level language like Modula makes high-level tasks easy, but the programmer loses the ability to access the machine architecture directly, as she can do when programming in assembly language. But the effort to write an accounting system (for example) in assembly language is much higher than in Modula, and the programmer is more prone to make a mistake.

Intuitively, we would like to be able to express every kind of property in one language. Someone may create a language in which this is possible. But, as [Lamport80] mentions, "this approach is based upon the misguided notion that the more expressive a system is, the better it is." We have high level languages to create an abstraction in order to hide the irrelevant details of the specific model we are working with.

TASpec has some limits that make expressing certain types of properties difficult. For example, consider the issue of "bounds", where one wants to say that between states P and R, the system is always in state Q. In many logics, this could be expressed as:

```
(P before Q) and (Q before R)
```

In TASpec, because of the representation of states as facts in the database, this is more complex:

```
(not P) implies (not Q)
      P implies Q
      Q and not R
```

Taking these three properties together, we have:

- If fact P has not been asserted, then if fact Q is asserted, the first property fails;
- If fact P has been asserted and fact Q has not been asserted, the second property fails;
- If fact R has been asserted and fact Q remains asserted, the third property fails

Combining these three statements, we have the desired result.

Now consider an "after" operator. This is more complex, because the TASpec *before* operator is weak. Expressing "**A** before **B** and ~**A** after **B** all the way to the end of the execution" would require converting the "after" operator to some expression using the *before* operator. A strong *before* operator would do this nicely, but the TASpec *before* operator is weak. So, this expression takes two steps:

1. Convert the "after" operator into a form using the strong "before" operator. This gives:

```
(A before B) and not ((B before not A) "strong before" A)
```

2. Express the strong "before" operator using the weak *before* operator, as described above.

This demonstrates the difficulty of mapping model checking specifications to TASpec. The reason for the difficulty is that the languages are fundamentally different, in that TASpec uses the model of facts in a database and logic languages use events. An event occurs. A fact is in the database or not in the database, but in TASpec's model one can record that a fact was in the database but is no longer there only by entering *another* fact in the database. This makes the underlying model of TASpec more like an assembly language, and the model-checking languages more like higher level languages. Intuitively, this makes sense, as TASpec deals with a much lower level of detail than is found in most models.

4 Common Vulnerabilities

Four common vulnerabilities are the escalation of privileges before authentication, buffer overflows, race conditions involving file accesses, and the ability to access files using a web server because the server does not adequately check the path name of the requested file. In this section, we present descriptions of how to write properties that will handle these problems. We assume a UNIX-like environment (this includes Linux), and present the first three properties for C programs, and the fourth for Java programs.

4.1 Privilege Escalation Without Authentication[2]

The problem of escalating privileges when a user has not properly authenticated herself arises because of programming flaws in most cases. In the UNIX world, consider the following program fragment:

[2] This is adapted from Fink and Bishop [6].

```
/* get user name */
if (fgets(stdin, uname, sizeof(uname)-1) == NULL)
     return(FAILED);
/* get user password */
typedpwd = getpass("Password: ");
/* now get information about user with that name */
if ((pw = getpwnam(uname)) != NULL){
     /* generate user's password hash */
     hashtp = crypt(pw->pw_passwd, typedpwd);
     /* compare this to stored hash;
          if match, grant access */
     if (strcmp(pw->pw_passwd, hashtp) == 0){
          /* match -- grant access */
          setuid(pw->pw_uid);
          return(SUCCESS);
     }
     /* didn't match -- fall through to deny access
*/
}
return(FAILED);
```

This fragment reads a user's name into a buffer, obtains the password from the user, and then validates it. If the password is correct, the user acquires privileges (the *setuid* system call). If not, an error code is returned.

Although this segment of code is straightforward, the escalation often occurs long after the authentication. If the programmer errs, the escalation may occur despite a user's having incorrectly authenticated herself. A bug in an FTP server illustrated this. The program authenticated the user, and set a "correct authentication" flag. If the user then tried to change to a new login (say, *root*), but entered an incorrect password, the authentication would fail but the flag would not be reset. As obtaining privileges was contingent on the flag being set, the user would promptly acquire *root* privileges without authorization.

We define a set of properties to capture the various states. The authentication process begins when a name is mapped to a set of privileges (*pwent->pw_uid*) and a password. This property is described as:

```
location func getpwnam(name) result pwent
          { assert user_password(name,
               pwent->pw_passwd, pwent->pw_uid); }
```

This instructs the instrumenter to add code to assert the predicate

`user_password(`*`name`*`, `*`password`*`, `*`UID`*`)`

with *name*, *password*, and *UID* the values stored in the variables *name*, *pwent->passwd*, and *pwent->pw_uid*, respectively.

The next state occurs when the cleartext password is hashed to produce the stored password. This property is described as:

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

This instructs the instrumenter to add code to assert the predicate

`password_entered(`*`hash`*`)`

where *hash* is the value returned by the function *crypt*, which is in fact a hash computed in the same way that a stored password is derived from the correct cleartext password.

The next state is entered when the computed hash is compared to the stored password:

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

At this point, the user would enter the *authenticated* state if the following holds:

```
password_entered(pwd1) and
     user_password(name, pwd2, uid) and
          equal(pwd1, pwd2)
     { assert authenticated(uid) ; }
```

Note this property is not tied to any function in the program. If, at any point, the three assertions joined by "and" in the above property hold, the predicate "authenticated(uid)" becomes asserted—and the "uid" corresponds to that of the predicate "user_password".

Finally, when privileges are escalated, we need to indicate this change of state. The relevant property is:

```
location func setuid(uid) result 1
     { assert access_acquired(uid); }
```

Note this includes the UID to which privileges are given.

To tie all this together, we require that one authenticate as a particular user before being given privileges of that user:

```
check authenticated(uid) before access_acquired(uid)
```

Now, let us consider two executions of this program. In the first, all proceeds as expected. The user "me" with UID 917 provides the correct password. During execution of the code fragment, after the *setuid* system call, the set of facts in the database is:

```
user_password("me", "xyz", 917)
password_entered("xyz")
equals("xyz", "xyz")
authenticated(917)
access_acquired(917)
```

At this point, the predicate

```
authenticated(917) before access_acquired(917)
```

holds, and no violations are found. But now the same user tries to become *root*, and supplies an incorrect password. After this (second) execution of the *setuid* system call, the database contains:

```
user_password("me", "xyz", 917)
password_entered("xyz")
equals("xyz", "xyz")
authenticated(917)
access_acquired(917)
user_password("root", "abc", 0)
password_entered("xyz")
```

```
equals("abc", "xyz")
access_acquired(0)
```

Now the property

```
authenticated(0) before access_acquired(0)
```

is false, and the TEM would report a violation.

4.2 File Creation Race Condition[3]

This flaw arises when two actions, in this example file creation and changing file ownership, are sequential and a third action can occur between the two. Consider a program executing as *root* that performs the following sequence of actions:

Create file
Read data from another program, adding it to file
Close file
Change ownership of file from *root* to *user*

If reading the data takes long enough for someone else to change the binding of the file name (used for the "create" and the "change ownership") to the file object, then the file whose ownership is changed will not be the one that was created. This can allow an unprivileged user to create a privileged program without authorization.

The relevant code fragment is:

```
/* create the file */
if ((fd = creat("xyz", O_WRITE) >= 0){
     /* read input and copy it to the
          created file */
     while(read(buf, 1000, finp) > 0)
          write(buf, n, fd);
     /* close it */
     close(fd);
     /* change ownership from root to
          user 917 group 10 */
     chown("xyz", 917, 10);
}
```

The race condition is exploited by a second program (one that does the rebinding). But as noted earlier, TASpec does not deal with concurrent programs. So, we look for system calls and functions that can block execution, such as the "read" function. Then a window of time during which the race condition can be exploited can be detected.

This involves two states. The first is *access*(*file*), in which the named *file* is being accessed. The second is *block*(*file*), in which the race condition exists. They work together as follows.

When the file is created, the process enters the *access*(*file*) state, and is not in the *block*(*file*) state. This is expressed as:

```
location func creat(file){
     assert access(file); retract block(file);
}
```

[3] This is adapted from Fink [7].

When the file is read, if the process is in the *access*(*file*) state, it then enters the *block*(*file*) state. The relevant property is:

```
location func read(){
     if access(file) { assert block(file) };
}
```

Note here the file name carries over from the last *creat* system call. When the file's ownership is changed, the process must not be in the *block*(*file*) state. That is expressed as:

```
location func chown(file, user, group){
     check not block(file);
}
```

Consider the database when the above code fragment is executed. After the *creat*, the database contains:

```
access("xyz")
```

After the while loop, the database contains:

```
access("xyz")
block("xyz")
...
```

where the `block("xyz")` predicate is repeated as many times as the loop is executed. When the *chown* system call is executed, the property

```
not block("xyz")
```

is tested, and fails. So this reports that a race condition exists.

4.3 Buffer Overflows

Buffer overflows are pernicious. A common source of buffer overflows is in the redaction of command line arguments. For example, in the fragment:

```
int main(int argc, char *argv[])
{
     char pname[1024];
     for(i = 0; argv[0]; i++)
          pname[i] = argv[0][i];
     pname[i] = '\0'
```

there is an implicit assumption that the name of the program (argv[0] will not be more than 1023 characters long. If it is, a buffer overflow will result.

First, we need to define a property that describes buffer overflow. It occurs when an array reference is out of bounds. Every array has an upper bound (for pname, 1023) and a lower bound (for pname, 0). Our property will say that when an array reference to element *i* occurs, and the array's bounds are *l* (lower) and *u* (upper), then *l* _ *i* and *i* _ *u* must both hold.

We define the predicate *array*(*pname*, 0, 1023) to mean that the array *pname* was declared with upper bound 1023 and lower bound 0. We define the predicate *arrayref*(*pname*, *i*) to mean that element *i* of array *pname* was referenced. The property to be tested is then:

```
check array(aname,lower,upper) and arrayref(aname,index)
          implies (lower <= index and index <= upper)
```

Note that we need not explicitly name the array in the property. The TEM will validate all array references when checking the property.

Next, we must instruct the instrumenter to put these predicates, and the property, into the source file appropriately. We use location specifiers to do this. The property file is:

```
location decl pname[1024] {
     assert array(pname, 0, 1023); }
location variable pname[i] {
     assert arrayref(pname, i);
     check array(aname,lower,upper) and
          arrayref(aname,index)
             implies (lower <= index and index <= upper);
}
```

Unlike the property, the “pname” and “i” in the first two lines must match the variables in the program. The instrumenter will transform the code fragment above into the following (conceptually; several implementation-level details and error checking are omitted):

```
int main(int argc, char *argv[])
{
     char pname[1024];
     tf = open(“directivefile”, WR_ONLY);
     fprintf(tf, “assert array(pname, 0, 1023)\n”);
     for(i = 0; argv[0]; i++){
          fprintf(tf, “assert arrayref(pname, i)\n”);
          fprintf(tf, “check ...\n”);
          pname[i] = argv[0][i];
     }
     fprintf(tf, “assert arrayref(pname, i)\n”);
     fprintf(tf, “check ...\n”);
     pname[i] = ‘\0’
```

(The “check ...” refers to the entire check in the property file; it is elided for clarity.) The program containing this code fragment can now be compiled and executed.

Assume the program is given a name of 1025 characters. When the run is complete, the state file will contain:

```
assert array(pname, 0, 1023);
assert arrayref(pname, 0);
check array(aname,lower,upper) and arrayref(aname,index)
          implies (lower <= index and index <= upper);
...
assert arrayref(pname, 1024);
check array(aname,lower,upper) and arrayref(aname,index)
          implies (lower <= index and index <= upper);
assert arrayref(pname, 1025);
```

```
check array(aname,lower,upper) and arrayref(aname,index)
          implies (lower <= index and index <= upper);
```
(the ellipsis indicates 1023 lines elided). When the TEM is run over this state file, the property will be checked at every *arrayref*, because both predicates hold. At the next-to-last check above, the antecedents are true, but the consequent is false, as *index* is 1024. Hence the TEM will report a violation of the property.

4.4 Improper Web Server Restriction

The next bug is one common to older web servers. When a web browser connects to a web server, the web server gives the browser access to a hierarchy of files. The area accessible to the browser is to be restricted to the top-level directory of this hierarchy, and its descendents. Hence an attempt to access the file "../../../../../etc/passwd" should fail if made from the top-level directory. The ".." means to access the parent directory. That directory exists on the web server, but the software should block access to it. Unfortunately, many web servers do not restrict this type of access. In this case, attackers can read system and other files not in the web hierarchy.

As a test of the Java implementation of property-based testing, we obtained a Java web server and had a student delete the checks for this flaw. We then used property-based testing to determine whether the flaw existed.

The basic structure of the relevant parts of the web server are as follows. A method called *HttpWorker.doit()* is given the URL from the browser. If the URL is illegal in any way, the web server throws an exception with that URL passed as a parameter.

First, we define the property. For this example, we will assume there are no subdirectories of the hierarchy. This simplifies the statement of properties without changing the basic ideas. Again, we define two predicates. The first, *hasDotDot*(*url*), asserts that the URL *url* has the ".." in it. The second, *causedException*(*req*), asserts that the URL *req* caused an exception to be thrown. Then the property of interest is:

```
check hasDotDot(url) implies causedException(url);
```

because if *hasDotDot*(*url*) is true, and *causedException*(*url*) is true, the URL referred to the parent directory and was caught. But if *hasDotDot*(*url*) is true and *causedException*(*url*) is not asserted, then the invalid URL is not caught and the property was violated.

We next instrument the server. The assertion for a URL referring to the parent goes, as indicated, at the beginning of *HttpWorker.doit()*:

```
location assign HttpWorker.doit()::req result r
     if "r.getRequestURI.indexOf(\"../\") > -1" {
          assert hasDotDot(r);
}
```

This says to check whenever something is assigned to the variable *req* in the function *doit*() in the class *HttpWorker*. If the new value of *req* contains ".." (the "if" part) then the program will output the appropriate assertion. Similarly, the assertion for exceptions is output when the reply function in the exception part of the class is called:

```
location funcall HttpException::reply(HttpRequest req,
                                      HttpResponse res){
     assert causedexception(req);
}
```
This binds the assert to the call to the *reply* function in the class *HttpException*. Note the URL is output; the specific response sent is irrelevant to the property.

When the instrumented web server was run, and a browser requested an illegal URL, the state file contained:
```
check hasDotDot(Areq) implies causedException(Areq);
assert hasdoddot("/../../../../../etc/passwd");
check hasDotDot(Areq) implies causedException(Areq);
```
The first line is vacuously true, as *hasDotDot*(*Areq*) has not been asserted. The last line, however, is false, because *hasDotDot*("/../../../../../etc/passwd") is asserted, but the corresponding *causedException*("/../../../../../etc/passwd") has not been asserted.

As a control, the same sequence of URLs was requested from an instrumented but correct version of the same web server. The state file for that version of the program contained:
```
check hasDotDot(Areq) implies causedException(Areq);
assert hasdoddot("/../../../../../etc/passwd");
assert causedException("/../../../../../etc/passwd");
check hasDotDot(Areq) implies causedException(Areq);
```
Both *hasDotDot*("/../../../../../etc/passwd") and *causedException*("/../../../../../etc/passwd") have been asserted when the final check is made.

4.5 Summary

This section presented four examples of programs with common flaws. We showed the properties that described each, and the instrumenting necessary to enable the TEM to detect changes of state that affect the truth of the properties as the program runs. These examples demonstrate the usefulness of this testing methodology.

5 Conclusion

Although it is used to specify properties that programs must satisfy, TASPec is different from logic languages used to do model checking. The key difference comes from TASpec's model of events (state changes are represented as facts in a database), which differs from the notion of state changes used in model checking. TASpec's lack of history constructs increases the complexity of expressing some temporal events. Fortunately, as shown by the description of several common vulnerabilities above, these do not affect the use of TASpec to describe many common implementation flaws.

References

1. M. Bishop, *Computer Security – Art and Science*, Addison Wesley, Boston, MA (2003).

2. L. Lamport Leslie, "'Sometime' Is Sometimes 'Not Never': On the Temporal Logic of Programs", *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pp. 174-185 (Jan. 1980).

3. A. van Lamsweerde, "Formal Specification: a Roadmap", *Proceedings of the Conference on The Future of Software Engineering*, pp. 147-159 (June 2000).

4. M. Dwyer, G. Avrunin, and James C. Corbett, "Property Specification Patterns for Finite-State Verification", Proceedings of the Second Workshop on Formal Methods in Software Practice, pp. 7-15 (Mar. 1998).

5. M. Dwyer, "Spec Patterns", athttp://patterns.projects.cis.ksu.edu/documentation/patterns.shtml.

6. G. Fink and M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance," *ACM SIGSOFT Software Engineering Notes* **22** (4) pp. 74-80 (July 1997).

7. G. Fink, *Discovering Security and Safety Flaws using Property-Based Testing*, Ph.D. Thesis, Dept. of Computer Science, University of California, Davis, CA 95616-8562 (1996).

Distribution

2		Department of Computer Science Attn: Matt Bishop Vicentiu Neagoe University of California, Davis Davis, CA 95616-8562
1	MS 0151	T. Hunter, 9000
1	MS 0151	A. Ratzel, 9750
1	MS 0784	R. Trellue, 5610
1	MS 0785	R. Hutchinson, 5616
1	MS 0795	P. C. Jones, 9317
1	MS 9001	M. John, 8000
1	MS 9003	C. Hartwig, 8940
1	MS 9011	N. Durgin, 8941
1	MS 9011	B. Hess, 8941
1	MS 9011	J. Howard, 8941
1	MS 9151	J. Handrock, 8960
1	MS 9151	C. Oien, 8940
1	MS 9151	K. Washington, 8900
3	MS 9018	Central Technical Files, 8945-1
1	MS 0899	Technical Library, 9616
1	MS 9021	Classification Office, 8511/Technical Library, NS 0899, 9616 DOE/OSTI via URL
1	MS 0323	D.L. Chavez, LDRD Office, 1011