Vulnerability Testing of Software System Using Fault Injection

Wenliang Du[*]
Computer Operations, Audit, and Security Technology (COAST) Laboratory
1398 Department of Computer Sciences
Purdue University, W. Lafayette, IN 47907,USA
Aditya P. Mathur[†]
COAST Laboratory and Software Engineering Research Center (SERC)
1398 Department of Computer Sciences
Purdue University, W. Lafayette, IN 47907, USA

April 6, 1998

Abstract

We describe an approach for testing a software system for possible security flaws. Traditionally, security testing is done using penetration analysis and formal methods. Based on the observation that most security flaws are triggered due to a flawed interaction with the environment, we view the security testing problem as the problem of testing for the fault-tolerance properties of a software system. We consider each environment perturbation as a fault and the resulting security compromise a failure in the toleration of such faults. Our approach is based on the well known technique of fault-injection. Environment faults are injected into the system under test and system behavior observed. The failure to tolerate faults is an indicator of a potential security flaw in the system. An Environment-Application Interaction (EAI) fault model is proposed. EAI allows us to decide what faults to inject. Based on EAI, we present a security-flaw classification scheme. This scheme was used to classify 142 security flaws in a vulnerability database. This classification revealed that 91% of the security flaws in the database are covered by the EAI model.

Keywords: Environment fault, fault classification, fault injection, security, security flaw, testing, security testing.

1 Introduction

Security testing

Reports of security violations due to software errors are becoming increasingly common. We refer to such errors as "security errors" or "security flaws." This has resulted in security related concerns among software developers and users regarding the "robustness" of the software they use. All stages of software development are motivated by the desire to make the product secure and invulnerable to malicious intentions of some

[*]Portions of this work were supported by contract F30602-96-1-0334 from Rome Laboratory (USAF) and by sponsors of the COAST Laboratory.

[†]Portions of this work were supported by contract F30602-96-1-0334 from Rome Laboratory (USAF), by sponsors of the COAST Laboratory, and NSF award CCR-9102331.

users. Our work is concerned with the testing of software with the goal of detecting errors that might lead to security violations.

Traditional methods for detecting security flaws include penetration analysis and formal verification of security kernels [18, 20]. Penetration analysis relies on known security flaws in software systems other than the one being tested. A team of individuals is given the responsibility of penetrating the system using this knowledge. Formal methods use a mathematical description of the security requirements and that of the system that implements the requirements. The goal of these methods is to show formally that the requirements are indeed met by the system.

A weakness of penetration analysis is that it requires one either to know or be able to postulate the nature of flaws that might exist in a system. Further, the effectiveness of penetration analysis is believed to be as good as that of the team that performs the analysis. A lack of an objective criterion to measure the adequacy of penetration analysis leads to uncertainty in the reliability of the software system for which penetration analysis did not reveal any security flaws.

Attractive due to the precision they provide, formal methods suffer from the inherent difficulty in specifying the requirements, the system, and then applying the process of checking the requirements specification against system specification.

Recently, several specific security testing techniques have been developed [4, 7, 19, 23, 28]. As discussed in section 5, these techniques are either restricted to some specific security flaws or limited by the underlying testing techniques.

Another alternative for security testing is to use general testing techniques, such as path testing, data-flow testing, domain testing, and syntax testing [2]. However, the effectiveness of these techniques in revealing security flaws is still unknown and more studies are needed to justify their use in testing for security flaws.

Outline of our approach

Our approach for security testing employs a well known technique in the testing of fault-tolerant systems, namely fault injection. This approach has drawn upon years of research and experience in vulnerability analysis [1, 3, 6, 16, 21]. Our approach relies on an empirically supported belief that the environment plays a significant role in triggering security flaws that lead to security violations [9, 17].

The problem

For the purpose of our discussion, we assume that a “system” is composed of an “application” and its “environment.” Thus, potentially, all code that is not considered as belonging to the application belongs to the environment. However, we can reduce the size of the environment, by considering only those portions of the code that have a direct or indirect coupling with the application code. Such coupling might arise, for example, due to the application’s use of global variables declared in the environment or the use of common resources such as files and network elements.

For various reasons, programmers tend to make assumptions about the environment in which their application will function. When these assumptions hold, the application is likely to behave appropriately. But, because the environment, as a shared resource, can often be perturbed by other subjects, especially malicious users, these assumptions might not be true. A secure program is one that tolerates environment perturbations without any security violation.

If we consider environment perturbations to be faults, then a secure system can be regarded as a fault-tolerant system that is able to tolerate faults in the environment. Therefore, the goal of testing the security of a system is reduced to ensuring that the system is implemented to tolerate various environment faults; not leading to security violations is considered toleration of such faults. In the remainder of this paper, we will use the terms " environment perturbation" and "environment fault" interchangeably where there is no confusion.

Fault injection–the deliberate insertion of faults into an operational system to determine its response–offers an effective solution to validate the dependability of fault-tolerant computer and software systems [5]. In our approach, faults are injected into environment thereby perturbing it. In other words, we perturb the application environment during testing to see how the it responds and whether there will be a security violation under this perturbation. If not then the system is considered secure.

Advantages of our approach

The use of environment fault injection technique leads to several advantages. First, in practice, it is hard to trigger certain anomalies in the environment, and knowing how to trigger them depends on the tester's knowledge of the environment. Therefore, testing software security under those environment anomalies becomes difficult. Fault injection technique provides a way of emulating the environment anomalies without having to be concerned with how they could occur in practice. Second, our approach provides a systematic way of deciding when to emulate environment faults. If we want to test whether a system will behave appropriately under certain environment anomalies, we need to set up those environments. However, the set up time is often difficult to control. If the set up is too early, it might change during the test and the environment state might not be we is expected when an interaction between the application and the environment takes place. If the environment is set up too late, the effect it has on the application's behavior might not serve the purpose for which it was set up. By exploiting static information in the application and the environment's source code, our approach can, however, decide deterministically when to trigger environment faults. Third, unlike penetration analysis, where the procedure is difficult to automate and quantify, fault injection technique provides a capability of automating the testing procedure. In addition, we adopt a two-dimensional metrics to quantify the quality of our testing procedure.

Research issues

Fault injection requires the selection of a fault model [5]. The choice of this model depends on the nature of faults. Software errors arising from hardware faults, for instance, are often modeled via bits of zeroes and ones written into a data structure or a portion of the memory [15, 25], while protocol implementation errors arising from communication are often modeled via message dropping, duplication, reordering, delaying etc. [14]. Understanding the nature of security faults provides a basis for the application of fault injection. Several studies have been concerned with the nature of security faults [1, 3, 6, 16, 21].) However, we are not aware of any study that classifies security flaws from the point of view of environment perturbation. Some general fault models have also been widely used [13, 26, 28]. The semantic gap between these models and the environment faults that lead to security violations is wide and the relationship between faults injected and faults leading to security violations is not known. We have developed an Environment-Application Interaction (EAI) fault model which serves as the basis the fault injection technique described here. The advantage of the EAI model is in its capability of emulating environment faults that are likely to cause security violations.

Another issue in fault injection technique is the location, within the system under test, where faults are to be injected. In certain cases, the location is obvious. For example, in ORCHESTRA [14], the faults emulated are communication faults. Hence, the communication channels between communicating entities provide the obvious location for fault injection. In other cases, where the location is hard to decide, nondeterministic methods, such as random selection, selection according to distribution, are used to choose the locations. For example, FERRARI [15] and FINE [13] use such an approach. The selection of location is also a major issue for us. In the current stage of our research, we inject environment faults at the points where the environment and the application interact. In future work, we plan to exploit static analysis to further reduce the number of fault injection locations by finding the equivalence relationship among those locations. The motivation for using static analysis method is that we can reduce the testing efforts by utilizing static information from the program.

A general issue about software testing is "what is an acceptable test adequacy criterion?" [10]. We adopt a two-dimensional coverage metric (code coverage and fault coverage) to measure test adequacy.

The remainder of this paper is organized as follows: section 2 presents the fault model. A methodology for security testing is presented in section 3. In section 4 we present our evaluation of the EAI fault model by classifying known security flaws. Finally a brief overview of related studies is presented in section 5 followed by summary of this research and the potential for future work in section 6.

2 An Environment Fault Model

In order to determine system behavior under various environment conditions, an engineer must be able to determine the effects of environment perturbation on a given system. Therefore, it is useful to inject faults that manifest themselves as errors in systems at the environment-application interaction level. To maintain confidence in the validity of the errors, the model used for these injections should be drawn from actual environment faults, while faults injected into the system should be able to emulate those environment faults appropriately. One assumption behind this requirement is that a security violation resulting due to the injected fault is similar to one that results due to an environment fault that arises during the intended use of the system.

2.1 Terminology

Definition 2.1 *(Internal State and Internal Entity)* Any element in an application's code and data space is considered an internal entity. A state consisting of the status of these entities is called an internal state.

Variable i in a application, for example, is an internal entity. The value of i is part of an internal state. The size of a buffer used in the application is also part of its internal state. In general, all information in this application's data space, stack space, and heap space are part of its internal state.

Definition 2.2 *(Environment Entity and Environment State)* Any element that is external to an application's code and data space is called an environment entity. A state that consists of the status of these entities is called an environment state.

For instance, file and network are treated as environment entities. The permission of a file, existence of a file, ownership of a file, real user-id of a process, and the effective user-id of process are different parts of an environment state.

A key difference between an environment and an internal entity, which makes implementation of a secure system difficult and error-prone, is the shared nature of the environment entity. An application is not the only one that can access and change an environment entity. Other objects, such as other users, may access and change the environment entity as well. Internal entity, on the other hand, is private to an application in the sense that only the application can modify and access them, assuming that the underlying operating system provides protected process space.

In concurrent programming, shared resources are handled by using the mutual exclusion and the semaphore mechanism to guarantee assumptions about the state of shared resources. However, we believe that few programmers use a similar mechanism to guarantee their assumption about the state of the environment. There are several reasons for this. First, programmers might not have recognized that the environment entities are shared resources. When, for example, an application writes to a file, it checks that it has the permission to write to that file, and then assumes that right in subsequent operations to that file without noticing that a malicious attacker could have change the environment thereby rendering the assumption false. Most security flaws resulting from race conditions [4] are caused by such dubious assumptions. Second, although some mechanisms, such as file locking, guarantee that a programmer's assumption hold on some part of the environment state, there is no general mechanism to do the same as the environment entity has various attributes than what the mutual exclusion and semaphore mechanisms could handle. As a result, programmers often use *ad hoc* mechanisms to guarantee the correctness of their assumptions. This can lead to errors more readily than would be the case when a standard mechanism is used.

2.2 Developing a fault model

In order to provide high confidence in the validity of the security flaws caused by environment faults, the methodology described here models systems at a high level. We refer to this level as the Environment-Application Interaction (EAI) level. Fault injection at the interaction level attempts to emulate what a "real" attacker does. Since most of the vulnerability databases record the way attackers exploit a vulnerability, we transform these exploits to environment faults to be injected with little analysis on those records thereby narrowing the semantic gap between faults injected at the interaction level and faults that really occur during the intended use of the system. In contrast, other studies [28] inject faults at the program statement level thereby leaving a large semantic gap between faults injected and those that might arise during the intended use of the application.

2.3 An EAI fault model

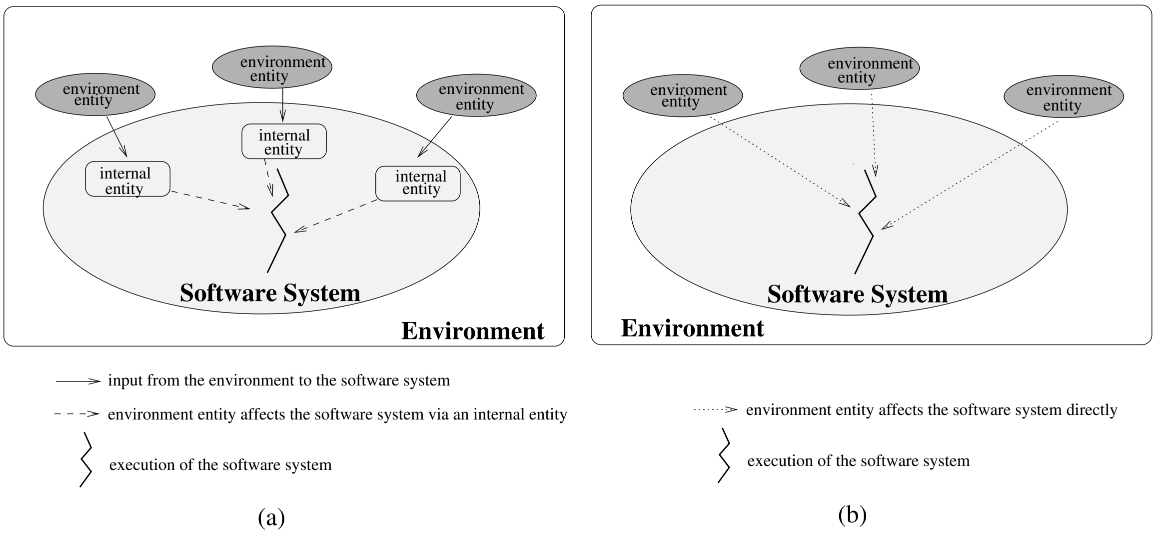

Figure 1: Interaction Model

In general, environment faults affect an application in two different ways. First, an application receives inputs from its environment. The environment faults now become faults in the input, which is then inherited by an internal entity of the application. From this point onwards the environment faults propagate through the application via the internal entities. If the application does not handle the faults correctly, a security violation might occur. The direct reason for this violation appear to be faults in the internal entity. However, this violation is due to the propagation of environment faults. Stated differently, the environment indirectly causes a security violation, through the medium of the internal entity. Figure 1(a) shows this indirect way in which the environment faults affect an application.

Consider the following example. Suppose that an application receives its input from the network. Any fault in the network message related to this input is inherited by an internal entity. When the application does a memory copy from this message to an internal buffer without checking the buffer's boundaries, the fault in the network message, the fault being "message too long," now triggers a violation of security policy.

A second way in which an environment fault affects the application is when the fault does not propagate via the internal entity. Instead, it stays within the environment entity and when the application interacts with the environment without correctly dealing with these faults, security policy is violated. In this case, the environment faults are the direct cause of security violation and the medium for environment faults is the environment entity itself. Figure 1(b) shows this direct way in which the environment faults affect an application.

Let us now a consider an example to illustrate this second kind of interaction. Suppose that an application needs to execute a file. There are two possibilities one being that the file belongs to the user who runs the application. Here the environment attribute is the file's ownership. In this case the execution is safe. The other possibility is that the file belongs to some malicious user. This is an environment fault created by the malicious user. Now the individual who runs the application assumes that the file belongs to the application. If the application does not deal with this environment fault, it might execute arbitrary commands in that file thereby resulting in a security violation.

The most error-prone interaction between an application and the environment is that involving files. Programmers tend to use an abstraction of a file that includes only a subset of the file attributes. A file name with a location or file content, for example, is a commonly used abstraction of a file. The environment faults, such as a long file name or a file name with special characters, associated with this abstraction will propagate via the internal entity. If the application does not place appropriate checks on these internal entities, such environment faults will cause security violations such as those due to buffer overflow and the execution of an unintended command. The environment faults associated with the remaining file attributes, such as whether the file is a symbolic link, the ownership of the file, existence of the file, and the permissions associated with the file, will not propagate via an internal entity. Although these attributes are extrinsic to the application, if not dealt correctly, they are likely to directly affect the interaction between application and environment.

In summary, we have categorized the environment faults according to the way they affect applications. Environment faults which affect programs via internal entities are called *indirect environment faults*. Environment faults which affect programs via environment entities are called *direct environment faults*.

2.3.1 Indirect environment faults

We categorize indirect environment faults according to the way they propagate in the internal space. The propagation includes initialization and use of an internal entity corresponding to an environment fault. Different ways of propagation are summarized in the following.

First, different kinds of environment faults are transferred to an internal entity, which has been initialized, in different ways. Most common initializations are through the interaction of the application with the environment, in which case, there must be a statement in the program that performs this initialization. However, for other initializations, there is no such statement in the application. The initialization of an environment variable, for example, is carried out by the operating system. The aspect of this kind of internal entity can easily cause mis-handling since programmers rarely notice the initialization or even their existence.

Second, environment faults inherited by internal entities propagate in different ways since internal entities come from different sources and are used differently. Some internal entities are used by the application directly in that there are explicit statements in the application that use the internal entities. Other internal entities are used by the application indirectly, meaning that there is no explicit statement in the application that uses the internal entities. Implicit usage might be caused by system calls as system calls use some internal entities without being noticed. When, for instance, a system call is made in `UNIX` to execute a command without using an absolute path, one might not notice from the application that this system call uses the `PATH` environment variable to find the location of that command. Without this knowledge on how the system call works, programmer is unaware of this invisible use of the internal entity and hence might make incorrect assumptions about it.

As per the above discussion, an understanding of security flaws is facilitated by dividing indirect environment faults into the following five sub-categories according to their origins.

- **user input**
- **environment variable**
- **file system input**
- **network input**
- **process input**

According to vulnerability analysis reported in [1, 3, 6, 16, 21] and our analysis of a vulnerability database, faults likely to cause security violations depend on the semantics of each entity. `PATH`, for example, is an environment variable, and comprises a list of paths used to search a command whenever an application needs to execute that command. In this case, the order of paths is important since the search will look for that command using the order specified in `PATH`, and the search will stop right after it has found it. The security could most likely be affected by changing the order of paths in the `PATH` variable or appending a new path to it. Certainly, an arbitrary modification of `PATH` will rarely cause a security breach.

Different semantics of each internal entity is summarized in Table 5.

2.3.2 Direct environment faults

Direct environment faults are perturbations of environment entities that affect an application's behavior directly. Unlike the internal entities, which consist only of variables, environment entities are more complex. For each type of entity, the attributes vary. There are three types of environment entities in a traditional operation system model. We categorize environment faults according to this model. These categories are enumerated as:

- **file system**
- **process**
- **network**

Studies of security violation reports, vulnerability databases, and vulnerability analyses suggest several security-related attributes corresponding to each environment entity. These are summarized in Table 6. This list is not exhaustive, nevertheless it provides the common attributes that appear in reports of security violations. Future vulnerability analyses, however, might add new entries to the list.

3 Environment Fault Injection Methodology

3.1 Fault injection

Like the EAI model, which models the environment faults at the interaction level, fault injections are also done at the interaction level. The previous section classifies the environment faults into direct and indirect environment faults. These faults are injected using the following mechanisms:

1. **Indirect Environment Fault Injections:** An indirect environment fault occurs at the interaction point where an application requests its environment for an input. The input that the environment provides to the application will most likely affect the application's behavior. A secure application should tolerate an unexpected anomaly in the environment input. One way to perturb the input is to use random input as in Fuzz [8, 23]. However, this approach dramatically increases the testing space, which and calls for a significantly large amount of testing effort. The Fuzz approach does not exploit the semantics of each input. Our vulnerability analysis, however, has shown that inputs most likely to cause security violations tend to have patterns according to their semantics. If, for instance, the input is a list of paths used to search for a command, then security failure will most likely occur when the order of these paths is altered, a new path is inserted or deleted, or the length of the list is increased. Other kinds of perturbations are less likely to cause security failure. Thus, by an examination of rare cases and by concentrating instead on fault patterns already observed, we reduce the testing space considerably.

   Faults injected into the application are based on patterns that are likely to cause security faults. These patterns come from our investigation of a vulnerability database and other studies reported in the literature. The faults are summarized in Table 7.

2. **Direct Environment Faults Injections:** A direct environment fault occurs at the interaction point where the application accesses an environment entity for creation, modification, reading or execution of an environment entity. Different status of environment entity attributes will affect the consequences of those interactions. Thus, the environment fault injections are used to perturb the attributes of an environment entity at points of interaction and to observe how the application responds to the perturbation. For example, before an application executes an `open` operation to a named `file`, several perturbations are performed on this file by changing its attributes such as its existence, permissions, ownership, and the type of the file since failure to handle these attributes is most likely to cause security violations. These attributes are listed in Table 6 while their perturbation is presented in Table 8.

3.2 Test adequacy criterion

An important issue in the management of software testing is to "ensure that prior to the start of testing the objectives of testing are known and agreed upon and that the objectives are set in terms that can be measured." Such objectives "should be quantified, reasonable, and achievable" [11].

We use *fault coverage* and *interaction coverage* measure test adequacy. Fault coverage is defined as the percentage of the number of faults tolerated with respect to that of the faults injected. Our conjecture is that the higher the fault coverage the more secure the application is. In addition to fault coverage, an additional measurement of the testing effort is the interaction coverage. Interaction coverage is defined as the percentage of the number of interaction points where we injected faults with respect to the total number of interaction points. Once again, we conjecture that the higher the interaction coverage, the more dependable the testing result are. Of course we assume that faults found during testing are removed. These two coverage criteria lead to a 2-dimensional metric for measuring test adequacy.

Figure 2 shows the 2-dimensional metric and four sample points of significance. The metric serves as a quantitative evaluation of a test set. Point 1 is representative of the region where testing resulted in low

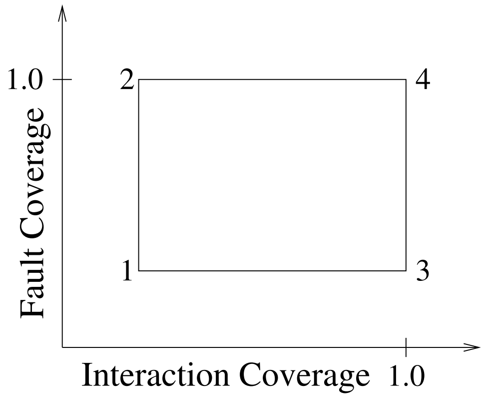

Figure 2: Test Adequacy Metric

interaction and fault coverage. In this case testing is considered inadequate. Point 2 is representative of the region where the fault coverage is high but interaction coverage is low. The test is considered inadequate since in this test, only a few interactions are perturbed, how the system behaves under perturbation of other interactions is still unknown.

Point 3 is representative of an insecure region because the fault coverage is so low that we consider the application is likely to be vulnerable to the perturbation of the environment. The safest region is indicated by point 4 which corresponds to a high interaction and fault coverage.

3.3 Implementation

The proposed methodology is currently under implementation. Once implemented we propose to conduct an experimental evaluation. The implementation of this methodology consists of the following 10 steps.

1. Set **count** and **n** to 0.

2. For each test case, do step 2 to 9.

3. For each interaction point in the execution trace, decide if the application asks for an input. If there is no input, only inject direct environment faults; if there is an input, inject both direct and indirect environment faults.

4. Decide the object where faults will be injected.

5. Establish a fault list corresponding to this object using Table7 and Table 8.

6. For each fault in the list, inject it before the interaction point for the direct environment faults; inject each fault after the interaction point for the indirect environment faults since in this case, we want to change the value the internal entity receives from the input.

7. Increase **n** by 1.

8. Detect if security policy is violated. If so, increase **count** by 1.

9. Calculate interaction coverage. If the test adequacy criteria for interaction coverage is satisfied then stop else repeat steps 2-9 until the adequacy criteria for interaction coverage is achieved.

10. Divide **count** by **n** yielding α to obtain the vulnerability assessment score (fault coverage) for the application.

3.4 Example

To illustrate the steps shown above, we consider an example of fault injection. The following code is taken from BSD version of lpr.c. Notice that lpr is a privileged application. It is a set-UID application which means that it runs in the root's privilege even when it is invoked by a user who does not have the same privilege as the root.

```
f = create(n, 0660);
if (f<0) {
     printf(``%s: cannot create %s'', name, n);
     cleanup();
}
... (code skipped here)
if (write(f, buf, i)!=i) {
     printf(``%s: %s: temp file write error\n'', name, n);
     break;
}
```

Suppose that we have decided to perturb the environment at a place where the create system call is issued. This is an interaction point where lpr interacts with the file system. There is no input in this case and hence we simply carry out direct environment fault injections.

The next step is to identify the object. Here, n is a file name, and hence the object is the file referred to using this file name. Then we refer to Table 8 and get a list of attributes that need to be perturbed. This list includes 1) file existence, 2) file ownership, 3) file permission, 4) symbolic link, 5) file content invariance, 6) file name invariance and 7) working directory. A further analysis shows that attributes 5 and 6 are not applicable in this case as this is supposed to be the first time the file is encountered.

We then perturb the remaining four attributes of the file and inject the faults into the application. For example, the perturbation of the "existence" means that we make the file exist or not exist before the application creates it. The perturbation of "symbolic link" means that we make the file link to some other file, such as the password file, before the application creates it.

After fault injection, we execute the application and detect if there is any violation of the security policy. In this case the violation is detected when we perturb attributes 1, 2, 3 and 4. Doing so causes lpr to write to a file even when the user who runs it does not have the appropriate ownership and file permissions. Thus when the file is linked to the password file, the password file is be modified by lpr. The problem here is that the application assumes that the file does not exist before the creation or assumes that the file belongs to the user who runs the application. In a real environment, this assumption could easily be false and the fault injection test points out a security vulnerability.

4 Evaluation of the EAI fault model

4.1 Criteria

Two criteria are needed to evaluate the EAI fault model. First, a good EAI fault model should cover a significant portion of environment faults that have led to security violations in practice. Second, when injected into an application, environment faults from EAI fault model should have appropriate capability to reveal software vulnerability related to the environment.

To evaluate EAI using the first criterion, we classify software security flaws according to a classification scheme based on the EAI fault model. The percentage of security flaws that can be classified using this

Table 1: high-level classification (total 142)

Categories	Indirect Environment Fault	Direct Environment Fault	Others
number	81	48	13
percent	57%	34%	9%

scheme reflects the covered portion of real-world environment faults that lead to security violations. Later we summarize the result of our classification.

To evaluate EAI using the second criterion, we plan to design a set of experiments. In these experiments we will seed some known software security flaws into target applications and apply our fault injection testing approach. The portion of seeded vulnerability revealed by our approach will be considered as a reflection of how good the EAI fault model is. These experiments have not been designed at the time of writing this paper.

4.2 Data collection

A security vulnerability database [17] is maintained in the COAST Laboratory at Purdue University. Currently there are 195 entries in this database which include vulnerabilities of applications from different operating systems, such as `Windows NT`, `Solaris`, `HP-UX`, and `Linux`. A useful property of this database is that most of the vulnerabilities are analyzed in detail either using the first hand knowledge from actual penetration testing or using second hand knowledge.

Among the 195 entries in the database 26 entries do not provide sufficient information for our classification, 22 entries are caused by incorrect design, and 5 entries are caused by incorrect configuration. Both design and configuration errors excluded from the scope of our research. We therefore classify only those errors that manifest directly as incorrect code in the application. Hence the total number of entries used for our classification is 142.

4.3 Classification scheme

As each environment-related security flaw is caused by the failure to tolerate the environment faults, we classify the security flaws according to the environment faults they failed to tolerate. The EAI fault model can be considered as a classification of the environment faults. An environment-related security flaw belongs to category A if and only if the environment fault it failed to tolerate belongs to category A. The algorithm for our classification is given in Figure 3.

4.4 Results of classification

Table 1 shows the high-level classification of environment faults. 91% of the 142 security flaws are classified by using the EAI fault model; the remaining 9% are caused by software faults irrelevant to the environment. These include errors such as those due to mistyping of the code in the application.

Table 2 shows the classification of indirect environment faults. Table 3 shows the classification of direct environment faults. Data in Table 3 indicates that a significant number of part of software vulnerabilities are caused by the interaction with the `file system` environment. Interaction with the `network` contributes only 10% of all software vulnerabilities in our database. The reason for the low percentage network-caused vulnerabilities is that most of the network vulnerabilities are introduced by a weak protocol design which

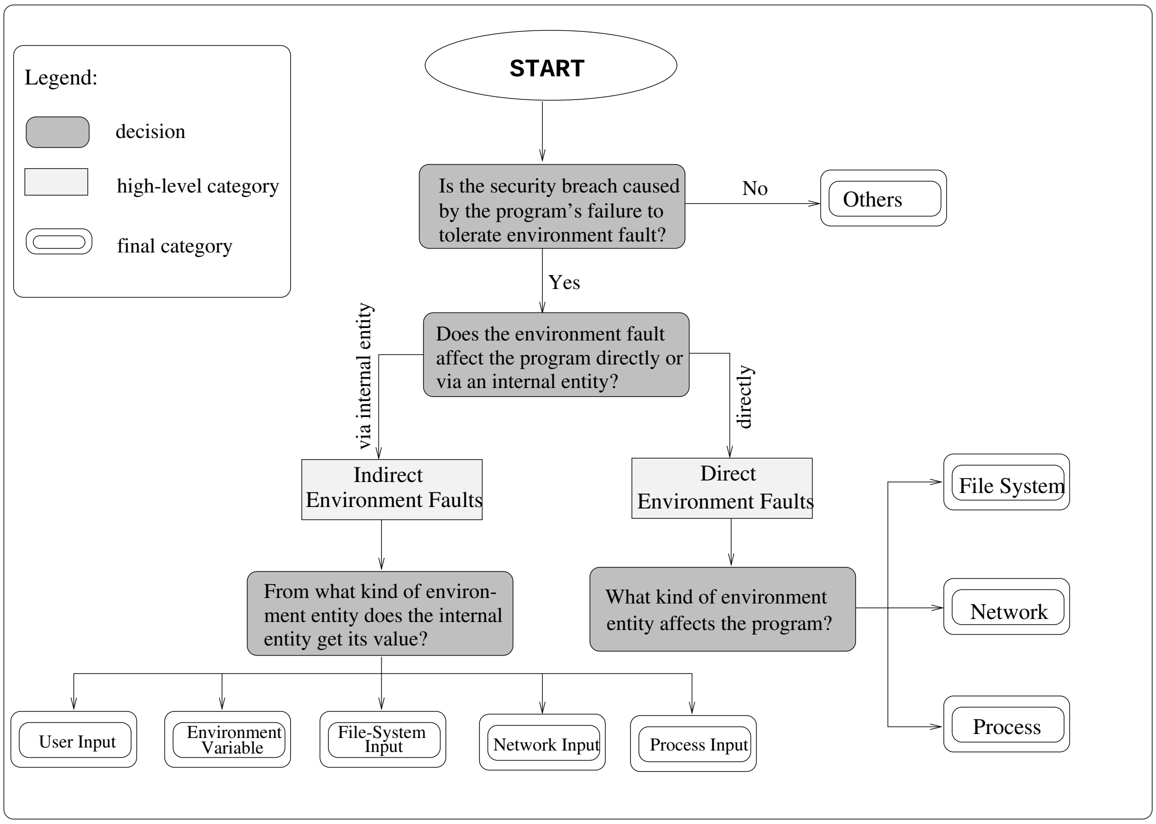

Figure 3: Classification Algorithm

does not fall into the scope of our classification. Table 4 provides further classification of `file system` environment faults according to Table 6.

5 Related Work

A significant amount of computer security testing is performed using penetration testing. Security is assessed by attempting to break into an installed system by exploiting well-known vulnerabilities. Several researchers, including Linde and Attanasio [18], Pfleeger [24], describes the process of penetration testing. Pfleeger points out that penetration testing is prone to several difficulties. First, there is usually no simple procedure to identify the appropriate cases to test. Error prediction depends on the skill, experience, and familiarity with the system of the creator of the hypotheses. Second, there is no well defined and tested criterion used to decide when to stop penetration testing. Statistical analysis is needed to show how much

Table 2: Indirect Environment Faults that Cause Security Violations (total 81)

Categories	`User Input`	`Environment Variable`	`File System Input`	`Network Input`	`Process Input`
Number	51	17	5	8	0
Percent	63%	21%	6%	10%	0%

Table 3: Direct Environmental Faults that Cause Security Violations (total 48)

Categories	File System	Network	Process
Number	42	5	1
Percent	87%	10%	2%

Table 4: File System Environmental Faults (total 42)

Categories	file existence	symbolic link	permission	ownership	file invariance	working directory
Number	20	6	6	3	6	1
Percent	48%	14%	14%	7%	14%	2%

confidence we can gain after a certain "quantity" of penetration testing has been done. Penetration testing does not provide such a metric. Third, it is difficult to develop a test plan as it not only needs familiarity with system but also needs skill and experience. It is also possible that testers do not know how to develop a test to investigate some hypotheses due to the limitation of their knowledge of the environment. This might lead to a decrease in our confidence in the test result as attackers might know what the testers do not know.

Our research attempts to overcome the above mentioned difficulties. It has a deterministic procedure to conduct and test, a criterion to decide when testing should stop. It overcomes the limitation of the lack of knowledge of the environment by emulating possible attacks using the faults injection technique. Finally, our approach overcomes the limitation of testers' knowledge by offering a set of concrete faults that should be injected into application.

Adaptive Vulnerability Analysis (AVA) is designed by Voas et al. to quantitatively assess information system security and survivability. This approach exercises software in source-code form by simulating incoming malicious and non-malicious attacks that fall under various threat classes [22, 27, 28]. In this respect, our own work parallels the AVA approach. A major divergence appears, however, with respect to how incoming attacks are simulated. AVA chooses to perturb the internal state of the executing application by corrupting the flow of data and the internal states assigned to application variables. Our approach chooses to perturb the environment state by changing the attributes of the environment entity and perturbing the input that an application receives from the environment. Our approach should be considered as complementary to Voas'.

For attacks that do not affect the internal states of an application, AVA appears incapable of simulating them by only perturbing the internal states. For vulnerabilities that are caused purely by incorrect internal states, our approach cannot simulate them by only perturbing the environment. One disadvantage of the AVA is the semantic gap between the attacks during the use of an application and the perturbation AVA makes during testing. In other words, knowing that the application fails under certain perturbation, it is difficult to derive what kind of attacks correspond to this failure. This makes it difficult to assess the validity of the perturbation. Our approach narrows the semantic gap by perturbing at the environment-application level since most attacks really occur due to intentional perturbation of the environment.

Fuzz is a black-box testing method designed by Miller et al. It feeds randomly generated input stream to several system utilities, including `login`, `ftp`, `telnet`. The results show that 40% of the basic applications and over 25% of the `X-Window` application can crash [23]. Different patterns of input could possibly cause more applications to fail. Inputs made under different environmental circumstances could also lead to abnormal behavior. Other testing methods could expose these problems where random testing, by its very nature, might not [9]. Rather than rely on random inputs, our approach exploits those input patterns that could possibly cause security violations.

Bishop and Dilger studied one class of the time-of-check-to-time-of-use (TOCTTOU) flaws [4]. A TOCTTOU flaw occurs when an application checks for a particular characteristic of an object and then takes some action that assumes the characteristic still holds when in fact it does not. This approach focuses on a source-code based technique for identifying patterns of code which could have this programming condition flaw. One of its limitations is that static analysis cannot always determine whether the environmental conditions necessary for this class of TOCTTOU flaws exist [4]. The authors conclude that dynamic analyzers could help test the environment during execution and warn when an exploitable TOCTTOU flaw occurs. Our approach is dynamic. Instead of detecting dangerous environment conditions, we inject dangerous environment conditions and see whether the application will fail.

Fink and Levitt employ application-slicing technique to test privileged applications. Specifications are used to slice an application to an executable subset relevant to the specification, and manual methods are used to derive test data for the slice. By using application slices as the basis of security testing, they assume that testing a slice is equivalent to testing the whole application [7]. The motivation behind the application-slicing technique is to focus on a reduced and less complex portion of the application such that other static and dynamic analyses are made more efficient. We believe this to be a significant step in security testing. However, what is missing in this approach is an efficient testing technique used to test the slices. This paper assumes general testing methods can be used to test the slices and the effectiveness of their approach depends on the effectiveness of general testing methods on revealing security flaws, which, as far as we know, is still unknown.

Gligor has proposed a security testing method. It eliminates redundant test cases by 1) using a variant of control synthesis graphs, 2) analyzing dependencies between descriptive kernel-call specifications, and 3) exploiting access check separability. The method is used to test the Secure Xenix kernel [19]. A key drawback of this approach is that it cannot detect the fact that entire sequences of functions, i.e. access check computations, may be missing [12] as many security flaws are caused by the missing of access checking and input validity checking.

6 Summary and Future Work

We have presented a white-box security testing methodology using a variant of the fault injection technique. A classification scheme is also proposed in this paper based on the underlying EAI fault model. The initial results of evaluation of EAI fault model shows that 91% of software vulnerability from a vulnerability database is modeled by the EAI fault model.

Future work will concentrate on applying this methodology to an application that is in common use. We hope to develop and conduct a set of experiments on this methodology to evaluate its effectiveness. We plan to use the categorization scheme proposed in the paper to classify software vulnerability in different databases from commercial organizations to further assess the effectiveness of the underlying EAI fault model. Based on the outcome of the planned study we hope to be able to develop a prototype tool for security testing.

References

[1] T. Aslam. A taxonomy of security faults in the unix operation system. Master's thesis, Purdue University, August 1995.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[3] M. Bishop. A taxonomy of unix system and network vulnerabilities. Technical Report CSE-95-10, Department of Computer Science, University of California at Davis, May 1995.

[4] M. Bishop and M. Dilger. Checking for race conditions in file acesses. *The USENIX Association Computing Systems*, 9(2):131–151, Spring 1996.

[5] J. Clark and D. Pradhan. Fault injection: A method for validating computer-system dependability. *IEEE Computer*, pages 47–56, June 1995.

[6] W. Du and A. Mathur. Categorization of software errors that led to security breaches. Technical Report COAST Technical Report 97-09, Purdue University, Department of Computer Sciences, 1997.

[7] G. Fink and K. Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference; Orlando, FL, USA; 1994 Dec 5-9*, 1994.

[8] B. Miller, L. Fredriksen and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[9] S. Garfinkel and G. Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, Inc., 1996.

[10] J. Goodenough and S. Gerhart. Toward a theory of testing: Data selection criteria. *current Trends in Programming Methodology*, 2:44–79, 1977.

[11] H. Zhu, P. Hall and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[12] W. Howden. The theory and practice of functional testing. *IEEE Software*, 2:18–23, September 1985.

[13] W. Kao, R. Iyer and D. Tang. FINE: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Envineering*, 19(11):1105–1118, November 1993.

[14] S. Dawson, F. Jahanian and T. Mitton. ORCHESTRA: A fault injection environment for distributed systems. In *26th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 404–414, Sendai, Japan, June 1996.

[15] G. Kanawati, N. Kanawati and J. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proceedings 22nd International Symposium Fault Tolerant Computing*, pages 336–344, July 1992.

[16] I. Krsul. Computer vulnerability analysis thesis proposal. Technical Report CSD-TR-97-026, Computer Science Department, Purdue University, 1997.

[17] I. Krsul. Coast vulnerability database reference guide - draft version. Technical report, Purdue University, Department of Computer Sciences, 1998.

[18] R. R. Linde. Operating system penetration. In *AFIPS National Computer Conference*, pages pp. 361–368, 1975.

[19] V. D. Gligor, C. S. Chandersekaran, W. Jiang, A. Johri, G. L. Luchenbaugh and L. E. Reich. A new security testing method and its application to the secure xenix kernel. *IEEE Transactions on Software Engineering*, SE-13(2):169–183, February 1987.

[20] E. J. McCauley and P. J. Drongowski. The design of a secure operating system. In *National Computer Conference*, 1979.

[21] C. E. Landwehr, A. R. Bull, J. P. McDermott and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3), September 1994.

[22] J. Voas, F. Charron, G. McGraw, K. Miller and M.Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, August 1997.

[23] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, Computer Sciences Department, University of Wisconsin, 1995.

[24] C. Pfleeger, S. Pfleeger and M. Theofanos. A methodology for penetration testing. *Computers and Security*, 8(7):613–620, 1989.

[25] S. Han, K. Shin and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. Technical report, University of Michigan, Department of Elect. Engr. and Computer Science, 1995.

[26] M. Hsueh, T. Tsai and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, pages 75–82, April 1997.

[27] J. Voas. Testing software for characteristics other than correctness: Safety, failure tolerance, and security. In *Proc. of the Int'l Conference on Testing Computer Software*, 1996.

[28] J. Voas and G. McGraw. *Software Fault Injection: Incoculating Programs Against Errors*. John Wiley & Sons, Inc., 1998.

Appendix:

Table 5: Semantic attributes of an internal entity

Indirect Faults	Semantic Attribute	Description
User Input	file name	name of a file
	directory	name of a directory
	Command	name of a command executed in the application
Environment Variable	file name	name of a file
	directory	name of a directory
	execution path	list of paths used to search executable files or commands
	library path	list of paths used to search libraries
	permission mask	a mask which decides default permission of a newly created file
File System Input	file content	content of a file
	file name	name of a file
	directory	name of a directory
	file extension	special string that represents the type of files
Network Input	IP address	representation of IP address
	packet	packet
	host name	string that represents the name of a host
	DNS reply	reply from DNS server
Process Input	message	message sent from one process to another

Table 6: Attributes of environment entity

Direct Fault	Attribute	Description
File System	file existence	file does or does not exist
	file ownership	owner of the file
	file permission	the access permission for different users
	symbolic link	the file is a symbolic link to another file
	file content invariance	file can or cannot be modified during the execution of the application
	file name invariance	file name can or cannot be modified during the execution of the application
	working directory	the directory where the application is invoked
Network	message authenticity	the message is genuine or is spoofed by other people
	protocol	the message from network does or does not comply with underlying protocol
	status of socket	socket is or is not shared with another process
	availability of service	network service is or is not available
	trustability of entity	entity in the other end of network is or is not trusted
Process	message authenticity	the message is genuine or is spoofed by other people
	trustability process	the process with which the application is communicating is or is not trusted
	availability of service	the service that the application requested is or is not available
	protocol	the message from another process does or does not comply with underlying protocol

Table 7: Indirect Environment Fault Injections

Internal Entity	Semantic Attribute	Fault Injections
User Input	file name + directory name	change length, use relative path, use absolute path, insert special characters such as "..", "/" in the name
	command	change length, use relative path, use absolute path, insert special characters such as "|", "&", ">" or newline in the command
Environment Variable	file name + directory name	change length, use relative path, use absolute path, use special characters, such as "|", "&" or ">" in the name
	execution path + library path	change length, rearrange order of path, insert a untrusted path, use incorrect path, use recursive path
	permission mask	change mask to 0 so it will not mask any permission bit
File System Input	file name + directory name	change length, use relative path, use absolute path, use special characters in the name such as "|", "&" or ">" in name
	file extension	change to other file extensions like ".exe" in Windows system; change length of file extension
Network Input	IP address	change length of the address, use bad-formatted address
	packet	change size of the packet, use bad-formatted packet
	host name	change length of host name, use bad-formatted host name
	DNS reply	change length of the DNS reply, use bad-formatted reply
Process Input	message	change length of the message, use bad-formatted message

Table 8: Direct Environment Fault Injections

Environment Entity	Attribute	Fault Injections
File System	file existence	delete an existing file or make a non-existing file exist
	file ownership	change ownership to the owner of the process, other normal users, or root
	file permission	flip the permission bit
	symbolic link	if the file is a symbolic link, change the target it links to; if the file is not a symbolic link, change it to a symbolic link
	file content invariance	modify file
	file name invariance	change file name
	working directory	start application in different directory
Network	message authenticity	make the message come from other network entity instead of where it is expected to come from
	protocol	purposely violates underlying protocol by omitting a protocol step, adding an extra step, reorderng steps
	socket	share the socket with another process
	service availability	deny the service that application is asking for
	entity trustability	change the entity with which the application interacts to a untrusted one
Process	message authenticity	make the message come from other process instead of where it is expected to come from
	process trustability	change the entity with which the application interacts to a untrusted one
	service availability	deny the service that application is asking for