ABSTRACT

Title: ON THE USE OF FAULT INJECTION TO DISCOVER SECURITY VULNERABILITIES IN APPLICATIONS

Hariharan Sivaramakrishnan

M.S., 2006

Directed By: Prof. Michel Cukier

Center for Risk and Reliability

Department of Mechanical Engineering

The advent of the Internet has enabled developers to write and share software components with each other more easily. Developers have become increasingly reliant on code other than their own for application development; code that is often not well tested, and lacking any kind of security review, thus exposing its consumers to security vulnerabilities. The goal of this thesis is to adapt existing techniques, and discover new approaches that can be used to discover security vulnerabilities in applications. We use fault injection in each of our techniques and define a set of

criteria to evaluate these approaches. The hierarchy of approaches, starting from a black box and ending in a full white box approach, allows a security reviewer to choose a technique depending on the amount of information available about the application under review, time constraints, and extent of security analysis and confidence desired in the program.

ON THE USE OF FAULT INJECTION TO DISCOVER SECURITY VULNERABILITIES IN APPLICATIONS

By

Hariharan Sivaramakrishnan

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Advisory Committee:
Assistant Professor Michel Cukier, Chair
Assistant Professor Atif Memon
Dr. Ioana Rus

© Copyright by
Hariharan Sivaramakrishnan
2006

Dedication

To my parents, Bhavani and Sivaramakrishnan, who, through their emotional support, love and enthusiasm, have been the force constantly pushing me towards scaling greater heights during my college career here at UMD. This thesis dedication is my way of thanking them for their endless sacrifices and efforts to ensure that I receive a fantastic education.

Acknowledgements

I would like to take this opportunity to express my heartfelt since thanks and gratitude to Prof. Cukier for his encouragement, constant guidance and excellent support through the thesis process. I shall forever be indebted to him for the insightful conversations and discussions that we had over the last few months.

I would also like to extend my gratitude to the members of the thesis committee, Profs. Memon and Cukier, and Dr. Rus for taking time off their busy schedule and reviewing my research work.

Special thanks to my parents, and all the members of my family for their eternal love and affection.

Last but not the least, I would like to express my appreciation of the academic process. The last five years in university have been a fantastic learning opportunity and a life-changing experience. Successfully completing projects, exams, courses, research and papers gives encouragement and confidence unlike any other and makes education that much more enriching and rewarding.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

Software today is incredibly complex, making software engineering one of the most, if not the most challenging among all engineering disciplines. Programs will almost always have bugs in them; however, the most worrisome of these bugs are (security) vulnerabilities that could be exploited by an attacker to compromise an application. With developers increasingly adopting a componentized model for creating software, wherein they utilize code written by others in their applications, a vulnerability in one critical component has the ability to compromise all applications that depend on this module.

In programming today, security is often an after-thought; it is secondary to other more important considerations such as performance and usability. However, attackers are smarter, more energized, and more motivated than ever to compromise applications. In a recent paper, Panjwani et al showed that in 48 days, 760 different attackers attacked two computers that had 25 open vulnerabilities (each attacker was assumed to be associated with a source IP address) [1]. The invention of the Internet has made hacking all that much easier; a computer connected to the Internet is now susceptible to attacks from hackers all over the world. Security is a significant challenge, and it is bound to become more critical in the future, with applications increasingly using the network for communication, and growing in size and complexity.

Software testing and security research has produced various static and dynamic techniques to find application vulnerabilities. There are two approaches to finding vulnerabilities in applications, static and dynamic. Static approaches look at source code to identify potential security vulnerabilities, such as buffer overflows, null pointer references, dynamic memory allocation and memory corruption [4, 5, 6]. In software testing, which is a dynamic approach, strategies such as penetration testing are used, where a tester assumes the role of an attacker and tries to exploit an application by finding vulnerabilities in its specification and architecture. Another dynamic approach, fault injection, which has been used extensively in hardware verification in the past, is slowly catching steam as a possible complement to static analysis and testing methodologies. In this thesis, we consider different fault injection based approaches to make software more secure.

1.2 Approach

Towards solving this problem, we start with an environmental perturbation approach, where we analyze the interactions between an application and its execution environment. An application runs on an Operating System (OS), and thus the execution environment consists of all resources that are outside the program, including OS services such as access to the file system, network interface, other processes executing on the machine and environment variables on the system. Every application receives and processes external input, be it from the user, or from a remote source over a network, or by reading files. These external interactions allow a fault injector to inject faults in environmental resources, and modify them at runtime.

When exposed to these faults, a program may behave differently than how it would under normal input, thus exposing a security vulnerability.

Environmental interactions are a significant source of program vulnerabilities, and we would like to explore the usefulness of such a technique in the absence of program source code. Often, software components are only available as compiled libraries or executables, and do not include any source code listing. We are interested in exploring the applicability and usefulness of environmental fault injection given these constraints.

An application interacts with its environment though OS system calls, and these system calls can be traced by monitoring the application at runtime. Through these traces, one can decipher the names of system calls, the parameters to the call, and even their return value. These three pieces of information provide a security reviewer with knowledge about the kind of resource being accessed by the program (name of the system call), the name of the resource (usually the first parameter), how the resource are being used (other parameters in the function), and the success or failure of the access attempt (return code). This information allows a fault injector to inject faults in resources as and when they are used in the program, so that the next time the program accesses the resource, the resource looks different or has properties that the developer did not consider when writing the program.

Looking at an application's behavior when a fault is injected sheds some light on the possibility of there being a security vulnerability in the program. We consider fault injection to be a hybrid of both fault and error injection. We use the term fault injection a little loosely because some of the faults that we inject are indeed errors.

Under an environmental approach, faults are injected in resources that are external to the program, making the process relatively easy. The difficulty is in choosing the number and type of faults to be injected. The fault injector maintains a list of candidate faults that it can inject every time it encounters a certain system call. A security reviewer usually develops such a list before-hand. The particular fault injected is chosen based on the parameters to the system call.

In addition to environmental interactions, there are others sources of security vulnerabilities in programs as well, such as the improper use of library functions, broken programming logic or even a careless oversight on the part of the developer. We clearly need a different approach to find such vulnerabilities, because while it is possible that such implementation gaffs appear when environmental resources are used, there is an equivalently large, if not bigger pool of problems that are independent of a program's interactions with the outside world.

A cursory look at the problem of finding general security vulnerabilities in applications leads to a rather simple conclusion. The extent to which a security reviewer will be able to analyze an application's security characteristics and check for vulnerabilities is entirely dependent on the kind of information he/she has about the program. If a reviewer has full access to source code, then he/she can understand the security properties and assumptions of the program in detail, while if he/she is only provided with a compiled executable, then the reviewer is dependent on what information can be inferred by running the program and tracking its behavior. Clearly, not every person has access to source code, and similarly, there are occasions when more then just a compiled executable is available. Such information may

include application specifications, architectural designs, code comments, etc. We therefore see the need to develop a hierarchy of approaches that a security reviewer will be able to use as a reference when choosing a technique to discover security vulnerabilities in programs. A hierarchy allows us to define a taxonomy of techniques, with each technique relaxing the constraints imposed on that above it in the hierarchy. The technique chosen depends on how much is known about the program, the amount of time that can be invested in the analysis and the confidence level desired in the security behavior of the program.

We start by assuming that the only information available about a program is a compiled executable, and slowly relax constraints and ultimately end up in a full white box approach where we assume that the reviewer has full access to source code. The hierarchy that we propose has five levels, a basic black box approach, an environmental approach, using a program's flow graph information, using a flow graph with parameter metadata, and ultimately using the program's source code. Each technique uses fault injection in its own unique way. While the black box approach injects faults in program input, the environmental approach does so in resources used by the program. The basic flow graph and flow graph with parameter metadata techniques modify program variables, while the full source code approach looks at modifying program input and source code to detect vulnerabilities.

While the black box and environmental approaches use external representation of a program (a compiled executable), the two flow graph based approaches use internal program representations to find possible vulnerabilities. In the latter case, an application is represented using a program dependence graph,

which describes the control flow and data flow dependencies in an application. The goal of such an approach is to force a program into states that the programmer did not envision, either because of an improper understanding of the application requirements, or implementation bugs. When such a state is discovered, the program could be vulnerable, because there is a very good chance the programmer's implementation would be unable to handle the wrong state and be susceptible to attack either as soon as the new state is discovered or somewhere else in the execution path of the program.

While some of this information can be inferred by just looking at a program's flow graph, there is a lot of insight that can be gained by having metadata that describes the programmer's view of his implementation. Such metadata may include the programmer's understanding of constraints on the values of input parameters to a function, and the function's return value. The fault injector then attempts to find possible variable values, and control flow paths that would violate these programmer provided constraints. If such variable values and an execution path is discovered, there is a clear disconnect between the programmer's view of his implementation and the actual implementation in source code. We consider this to be a vulnerability.

Finally, with all of the program's source code available, a security reviewer can use any of the static analysis techniques that is suggested in the literature, and use some of the dynamic techniques discussed earlier in this thesis, such as the black box and environmental approaches to find vulnerabilities in programs. In addition, the reviewer can modify source code to short circuit certain portions of the application, thus breaking up the entire program into smaller segments, either one function or a

collection of functions at a time. Doing so will allow for small components of the bigger program to be reviewed one at a time. Each component can be analyzed more easily, and its security characteristics better understood.

Each of the five approaches that make up the hierarchy sounds like a viable option for use by a security reviewer. However, in order to provide a meaningful comparison of the techniques, we propose four criteria to determine the usefulness, effectiveness and viability of each of our approaches. These four criteria are:

i. The ability to choose a fault injection point

ii. Ease of injecting a fault, i.e. the fault injection mechanism

iii. The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability

iv. The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered

The hierarchy of approaches when coupled with the relative merits and demerits of each as obtained by using the four criteria defined above gives a security professional enough hints to make a judicious decision on the approach(es) that would be best suited for his/her purposes. The choice of technique would be determined by the amount of time available for the review, the kind of knowledge and understanding that the reviewer has about the application, the level of confidence desired in the security characteristics of the program.

In all of our techniques, we consider application crashes are indicative of vulnerable behavior by the application. This could also be viewed as a testing problem, wherein one looks at the availability perspective of execution. However, in our opinion, the ability to crash an application by injecting a fault (as our approaches do) provides a means for an attacker to exploit this vulnerability and unleash a DoS attack. The effects of such an attack are exacerbated if the application is unable to perform its most basic functionalities. For example, if a web server can be crashed by having garbage data in a protocol string, then the server can be rendered useless very soon.

Similarly, application crashes also introduce a discussion about the distinction between finding vulnerabilities and correctness. Our black box approaches use crashes to find discover vulnerable behavior; the absence of source code or any other information about a program's security policies makes analysis using other criteria rather difficult. Towards improving this, we propose output comparisons as another possible approach, the success of which greatly depends on the kind of information made available to the security reviewer.

1.3 Thesis Contributions

This thesis greatly innovates around some of the work previously done by Melody Djam (under the guidance of Prof. Cukier) in the realm of environmental perturbation and proposes new techniques and a hierarchy that uses these techniques in conjunction with fault injection to discover application vulnerabilities.

The author inherited Pulad, a precursor to EFIVA, which is a tool described in this thesis. Significant amount of time was spent in understanding Pulad, fixing the source code to make it more robust and expand its functionalities. In particular, Pulad was a simple program that could track another application's system calls, and modify file properties, i.e. it could cause environmental perturbations relating to the file system. The author invested time in actually determining how Pulad and the information that it gathered could be used in finding application security vulnerabilities. Pulad's fault injection capabilities were enhanced so that it now had some intelligence. Instead of depending exclusively on human input to determine which faults it should inject at each interaction point, Pulad could now look at system calls, their parameters, and automatically determine the type and number of faults that can be injected.

Further, Pulad was given the capability of comparing an application's behavior before and after fault injection and determine if any security vulnerabilities were potentially exposed. Therefore, Pulad's scope broadened from being a tool that could track system calls and modify file attributes to one that could also choose faults intelligently, track application behavior and output, and analyze the behavior to determine if a vulnerability was discovered. These enhanced functionalities were put to test by running the application on real world test scenarios. Hence, the author's work transformed Pulad into an Environmental Fault Injector and Vulnerability Analyzer (EFIVA).

The author has submitted two conference papers:

- H. Sivaramakrishnan, M. Cukier and M. Djam. *Using Fault Injection and Environmental Petrurbation for Vulnerability Discovery*. Submitted to the Sixth European Dependable Computing Conference (EDCC-6). Coimbra, Portugal, Oct. 2006

- H. Sivaramakrishnan and M. Cukier. *A Hierarchy of Approaches to Find Security Vulnerabilities in Applications Using Fault Injection*. Submitted to The 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006). 6-10 November 2006 - Raleigh, North Carolina, USA

1.4 Thesis Organization

The rest of this thesis is organized as follows: *Chapter 2: Environmental Perturbation and Fault Injection* describes environmental interactions, and the fault injection process, and how when used together, these two techniques can be an effective mechanism for finding vulnerabilities in applications. *Chapter 3: EFIVA: A tool to discover program vulnerabilities* describes the architecture and implementation details of the tool that we developed to test some of the ideas and hypotheses proposed in Chapter 2. It also describes some of our case studies that helped validate that EFIVA can be used effectively in a real-world environment. *Chapter 4: Expanding on the Environmental Approach* considers some of the pros and cons of the environmental approach proposed in Chapters 2 and 3, and describes a more expansive approach to finding vulnerabilities in applications. It introduces the

hierarchy of approaches and the criteria that help evaluate each technique. *Chapter 5: The Hierarchy of Approaches in Detail* describes each of the approaches laid out in Chapter 4 in detail, and compares and contrasts of all these techniques in a tabular format. *Chapter 6: Conclusions and Future Work* discusses the conclusions of our research, and possible future directions that it might take.

Chapter 2: Environmental Perturbation and Fault Injection

2.1 Motivation

With the ever-increasing complexity of software, there has been an explosion in the number of faults (bugs) present in software. While one would hope that most of these faults are benign, malicious users and attackers are always on the lookout for *(security) vulnerabilities* that can be *exploited,* thus compromising applications.

Vulnerabilities are classified into three types [2,3], network, host and application. As with all software faults, there are two verification approaches to finding vulnerabilities in applications, static and dynamic. Static verification methods and tools (FlawFinder [4], RATS [5] and ITS4[6]) have been explored extensively and with good success to find certain classes of vulnerabilities such as buffer overflows, integer overflows, and race conditions. However, information gathered from the execution profile of an application may be very different from that inferred through static analysis, thus making dynamic techniques a worthy complement to static techniques. Several factors such as the environment, operating system scheduling and concurrent execution are often left unexplored by static approaches; these open up a completely new class of problems that may cause an application to fail. Even if static analysis tools were to look at these external factors, they would identify *potential* bugs without being able to prove conclusively that a bug is indeed a vulnerability.

The other approach, dynamic verification involves software testing and fault injection. Among the various software testing strategies, *penetration testing* is most suitable to find security vulnerabilities. Penetration testing is a methodology whereby a tester intentionally tries to breach the security properties of an application by understanding its features and design. In [7], McGraw explains that penetration testing often occurs very late in the software development cycle, thus compromising the extent to which an application can be modified to fix any vulnerability that this approach may discover. Not only are there scheduling concerns with fixing vulnerabilities late in the cycle, but one also runs the risk of introducing a new vulnerability while fixing another. Software testing can be a useful tool to review the security characteristics of an application and with the right criteria, is successful in finding vulnerabilities. Our work focuses on the use of fault injection, which is the other form of dynamic verification.

Fault injection, which has historically been used extensively in hardware manufacturing processes, can also be used as a method to review the security characteristics of an application. More recently, fault injection has been used to verify that applications behave as expected for different input combinations and to find security vulnerabilities, the latter to very limited extents. Fault injection as a technique can be used in three different ways; either before the application starts executing (pre-execution Fault Injection), or when it is actually run (execution Fault Injection), or a combination of the two. When an application is analyzed, either by looking at source code, or its design documents and description, or in our case looking at its interactions with the execution environment, one can think of conditions

that possibly violate some of the application's assumptions. These conditions are immediately potential candidates for *faults* that can be injected either before the program starts executing, or during its execution phase. For example, if a program assumes the presence of an environmental variable, then deleting the variable before it is executed is an example of a pre-execution injected fault.

While pre-execution fault injection has the potential to expose vulnerabilities, execution fault injection has a greater potential to expose vulnerabilities that may not be readily obvious from a static review of an application's characteristics and behavior. In the previous example, let us assume that the environmental variable is deleted at runtime and not before the application starts execution. In such a situation, it is possible that in its first attempt, the application is able to read this variable, but fails when it tries to do so for a second time. If it fails on the second read, does the application use what it read before, or does it use some garbage value? Does the program crash or does it continue executing, albeit incorrectly? These interesting questions may be difficult to answer by looking at an application's source code (for example, in a multi-threaded environment), yet become readily apparent at runtime. Further, execution fault injection allows the fault injector to automatically adapt to actions of the program under test. The faults injected can be modified depending on the application's runtime behavior. On the other hand, pre-execution fault injection forces a re-run of the program every time a test parameter needs to be changed. This is because the fault injection system cannot automatically adapt itself to information that it collects by tracking the application's runtime behavior. This adaptation can

manifest itself in terms of perturbations to the internal state (variables) of an application [8], or the execution environment, or a combination of both.

While the jury is still out on the relative effectiveness of static vs. dynamic techniques, there are certain scenarios where one technique is naturally more applicable and effective than the other. As an example, static techniques could be extremely useful if an application's source code is available, but would be unable to make a good security review with the availability of only a compiled EXE. However, in such a situation, a black box approach would be the best first step to finding faults and security vulnerabilities.

2.1.1 Software Components without source code

Software today is highly componentized and often makes extensive use of libraries and Dynamic Link Libraries (DLLs) written by different vendors. Such Commercial Off The Shelf (COTS) components may have been exposed to very simple security reviews or sometimes none at all, making them potential targets for attackers, and a source of vulnerabilities for consuming applications. The majority of such components ship without source code, making the security analysis process more complicated for application developers and testers. Developers use libraries to ensure that the software development process is easier, but it is certainly not comforting when one's application is compromised by code that is acquired outside the organization. This problem is even more pronounced for libraries that are freely available on the internet and are the product of a developer's pet project.

Thus, there is the need to develop a security-testing framework that will allow for an intelligent black-box approach for finding security vulnerabilities. The

increasing number of security vulnerabilities due to environmental interactions, and the conduciveness of environmental perturbation to a black-box approach make interactions with the execution environment an ideal foundation on which to build our security platform. Further, the need to change environmental characteristics as an application executes makes fault injection the obvious choice for our research.

One of the most recent works by Neves et al [9] uses fault injection to discover vulnerabilities in implementations of the IMAP protocol. Their tool, AJECT, checks for buffer overflow and other vulnerabilities by detecting email server crashes on various input combinations. They adopt black box approach and inject faults in the information packets that are sent to the email server, i.e. its input. While we also adopt a black box approach, in addition to injecting faults in application input, we perturb the execution environment by injecting faults in resources that a program uses, notably the file system. Further, we implement output matching to determine if a vulnerability has been discovered in the application.

2.2 Environmental Interactions

Environmental interaction [10] refers to an application's use of resources or information from the environment where it is executed. These include files, communication over the network, communication with other processes, and requesting information (such as environment variables) from the operating system. When writing software, most programmers work with the implicit assumption that their application is the only one running on the system, and overlook the fact that their execution environment is constantly changing. This often manifests itself in how

file permissions and locations are handled, or how the program sends and receives messages over the network. Programmers make mistaken assumptions about the environment and do not account for the possibility that between its two uses in an application, a resource may have been used or modified by someone else, possibly with malicious intent. We consider environmental interaction to be a significant source of security problems in applications written today.

In [10], Du and Mathur propose a white-box approach to identifying software vulnerabilities that result from environmental interaction. They enumerate potential faults that can be used to discover different types of bugs, such as those pertaining to IPC (Inter-Process Communication) or file system calls; the latter being responsible for 87% of faults injected directly through the environment. A direct environmental fault refers to a fault that stays within the environmental entity where it was injected. On the other hand, an indirect environmental fault is a fault that was injected in an environmental entity but propagates in a program though an internal program entity. Our efforts are thus focused on finding vulnerabilities related to improper use of the file system, as opposed to the network and IPC, which comprise the other 13% of all vulnerabilities.

We use those faults that Du and Mathur recommend in their paper and are summarized in Table 1 below.

Entity	Attribute	Fault Injection
	`file name`	Change length, use relative path, use absolute path, insert special characters such as "..", "/" in the name

User Input	Directory	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 new line in the command
Environment Variable	file name	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	Directory	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	execution path	Change length, rearrange order of path, insert a non trusted path, use incorrect path, use recursive path
	Library path	Change length, rearrange order of path, insert a non trusted 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	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	Directory	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	file extension	Change to other file extensions like ".exe" in Windows system; change length of file extension
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 contents of the file
	file name invariance	Change file name
	Working directory	Start application in different directory

Table 1 Candidate faults to be used in the vulnerability discovery process

2.3 Fault Injection

As with most fault injection methods, we use a three-step approach to find application vulnerabilities. The three steps are, discovering the fault injection point, injecting the fault, and determining if a vulnerability has been exposed.

2.3.1 Discovering the fault injection point

The OS system call layer is the abstraction that programs use to communicate with environmental resources, and other processes running on the machine. During execution, an application can be traced to determine the system calls that it executes, thus providing information about the parameters passed to a function, and the point in the application where the call is made.

The parameters passed to a system call can be used to find information about the resources being used by the application, for example, the name a file, the kind of access permissions requested (read, write, both), and the number of times it is used.

2.3.2 Injecting the fault

The process of injecting a fault in an application involves two steps:

i. Identifying candidate faults

Table 1 above shows the different kinds of faults that we consider for use in our approach. The particular faults to be injected are determined from the trapped system call information. The name of the system call allows a filtering of the kind of faults to be injected based on category, while the function parameters help determine the particular file that is being worked on.

ii. *Injecting the fault into the application, or in our case the environment.*

As described in Table 1 above, the faults to be injected involve basic operations on files such as modifying their locations or attributes. These changes are easily implemented by using operating system Application Programming Interfaces (APIs).

2.3.3 Identifying a vulnerability

We define two very simple criteria for discovering vulnerabilities in an application. The first is checking for application crashes. If an attacker knows how to crash an application, he can use the same technique repeatedly to create a DoS attack.

The second technique that we propose is output comparison. If a security reviewer has some knowledge of the output to be expected from the application for a given input and environmental state, then this output can be compared against that actually produced by the application at runtime. If there a mismatch between the expected and actual program output, we assume that a potential vulnerability has been discovered in the program.

While these two criteria are extremely useful, the vulnerability identification step of our approach still needs human help and intervention. Quite clearly, faults injected during program execution may open up several vulnerabilities that go undetected by just looking at the output produced, or expecting the application to crash. Similarly, output produced by a program when faults are injected may be correct, but since it differs from the template provided by the tester, a program execution trace may be flagged as one that exposed a vulnerability. Human

intervention is necessary to reduce false positives, and in some cases detect missed cases.

At the core of the problem is the lack of adequate knowledge about the operations performed by an application by just looking at its execution profile. This is a trade off that one encounters between black box and white box approaches. While the environmental fault injection approach that we propose is not as exhaustive as a brute-force testing technique or full state exploration using static methods, it has the potential to discover a significant number of vulnerabilities and report fewer false positives than the other two techniques.

Chapter 3: EFIVA: A tool to discover program vulnerabilities

Chapter 2 introduced the usefulness of an environmental approach, and provided the list of candidate faults that a fault injector can use to perturb the environment and discover security vulnerabilities. It also briefly mentioned EFIVA, the fault injector that we developed to test our ideas and hypotheses. In this Chapter, we discuss the architecture of EFIVA, and provide some insight into the implementation of this tool.

EFIVA consists of three components, the collector, fault injector, and verifier. The collector is the component that runs the test application and records its interactions with the environment. EFIVA, as implemented now tracks interactions with the file system.

The tool runs the application under test twice; in the first execution, it collects information about all system calls including the time of their occurrence. In the second run, faults are injected and the application output verified.

3.1 Collector

The collector in turn has three primary components; they are the Application Executor, the Environmental Interaction Scanner (EIS), and the Persistor. The collector module as a whole is responsible for running the application and collecting information about all its interactions with the environment.

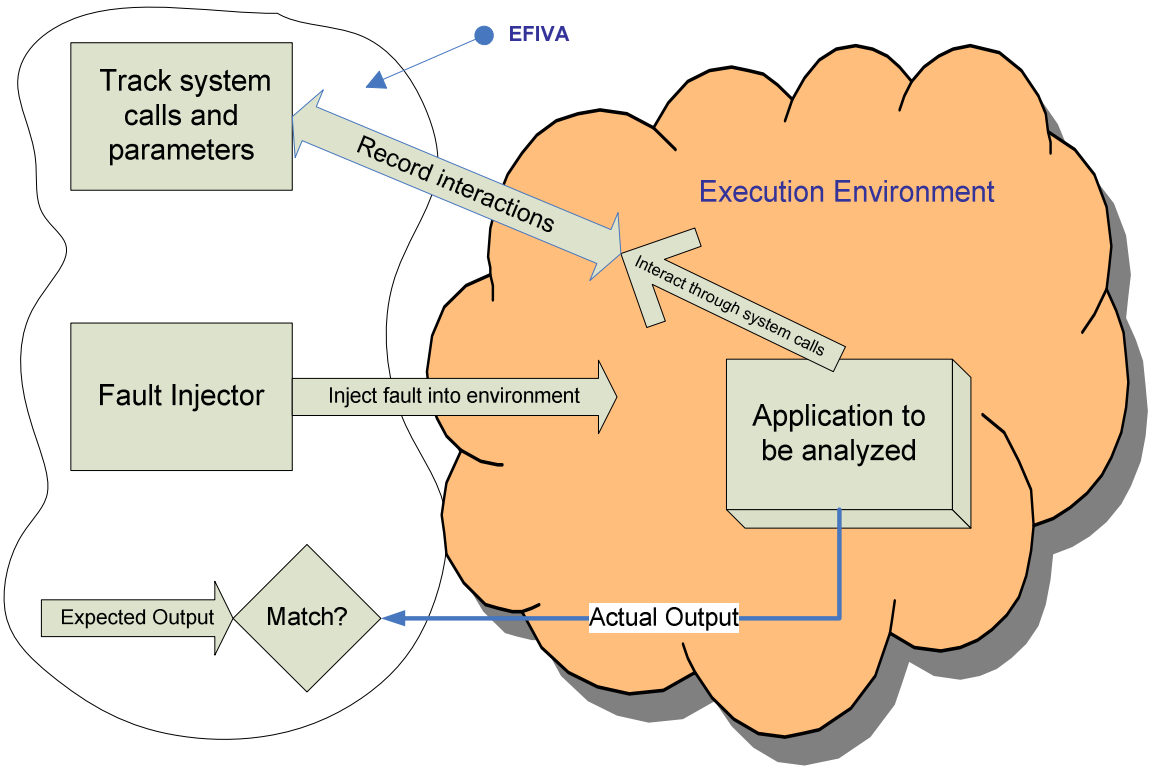

Figure 1 Conceptual model of our dynamic, black box approach

3.1.1 Application Executor

The application executor is the simplest of the three collector components; it executes the application under test by calling the environmental interaction scanner, and providing it appropriate command line arguments. This is the tool's interface to the outside world by accepting input about the application to be tested.

3.1.2 Environmental Interaction Scanner

The EIS module captures all file related application-environment interactions. It is built on Strace [11], which is a system call trace and debugging tool. Strace intercepts system calls made by an application without requiring its source code, or a recompile with a special compiler switch.

Strace executes a user specified command, which is the name of the application under test and its corresponding command line arguments. All system calls made by the application and signals that it receives from the operating system are recorded in an output file. In particular, for each system call, Strace records the function name, its arguments and return value.

Files being the focus of our work in the environmental interaction approach, we use Strace to track file system calls. Information gathered by Strace allows us to be intelligent about the faults injected. For example, one of the arguments in file system calls is the file I/O mode requested (Read, Write, etc). Modifying the corresponding file permissions enables us to inject faults that may have otherwise been difficult to discern by having a black-box representation of the application.

While Strace as implemented provides us with a lot of useful information, we have included two enhancements that will open up a new class of faults that can be used for vulnerability detection. Two features we added were the ability to determine the owner of files used in the application, and its access time. The access time is measured relative to when the program started execution.

In addition to collecting useful tracking statistics, Strace records unnecessary system calls and signals. For example, EFIVA is written in Java, and the Java process APIs are used to invoke Strace. This adds extraneous JVM environmental interactions to the output; EFIVA prunes out such information when Strace exits. Thus, at the end of one run, the tester now has a full trace of all file system interactions between the application and the environment.

At this point, the EIS module, having completed its functionality transfers control to the persistor module.

3.1.3 Persistor

The persistor module takes all the information recorded by the EIS module and stores it in a database. EFIVA uses this information in its second run to inject faults. This database is a useful information store where the tester can try to analyze the interactions to effectively design test scenarios outside of vulnerability detection. For example, the tester may be able to create a functional test case based on the sequence of system calls executed, and their corresponding parameters. This may be useful for testing application functionality, and not just vulnerability discovery.

3.2 Fault injector

The fault injection process consists of two steps: first is identifying the type of faults to inject and the second is actually modifying resources or input, i.e. making the fault visible.

3.2.1 Identifying the fault type

EFIVA has the capability to inject more than one fault at an injection point. The user of the tool has full control to decide what faults should be injected, and where. In the absence of any specific input from the user, EFIVA tries to be as intelligent as possible with choosing how many and what kind of faults should be injected at each step. For example, when the tool detects a file open system call with the read only parameter (O_RDONLY), it modifies the file ownership, making it

inaccessible to the user. The file open system call would thus fail, but if such an error condition is not checked for in the application, there is the possibility of a null pointer dereference, which would ultimately end in a program crash. Table 2 below provides a sampling of some of the heuristics that the fault injector uses to choose faults based on the application's runtime behavior.

System Call Name	System Call Parameters	Faults injected	Why these faults?
Open () Opens a file	O_RDONLY O_WRONLY O_RDWR Name of file being opened	- File existence - File ownership - File permission - File content invariance - Directory permission	O_RDONLY, and O_WRONLY assume that the file being opened exists, and do not create one if it doesn't. Use File content invariance when the O_RDWR or O_RDONLY is observed. Change permissions of the directory where the file is created or read or written
Read () Reads file contents	Buffer size Name of file being read	- File content invariance	The application may crash if the kind of data included in the file is different from what it expects.
Access () Checks if the executing program has appropriate privileges	R_OK W_OK X_OK Name of file or path being accessed	- File existence - File ownership - File permissions - File content invariance	The various security properties may be checked by the application, but the application may be vulnerable if these properties change between when they were checked, and when the file is

		- Change executing target to a symbolic link	actually opened. For X_OK, having a symbolic link point to any arbitrary location may have the program execute malicious code.

Table 2 Automatic fault injection decisions made by EFIVA

Similarly, environment variables and user inputs are modified using techniques illustrated earlier in Table 1.

Figure 2 below provides a graphical representation of the algorithm used by the fault injector to choose the faults to be injected in the environment.

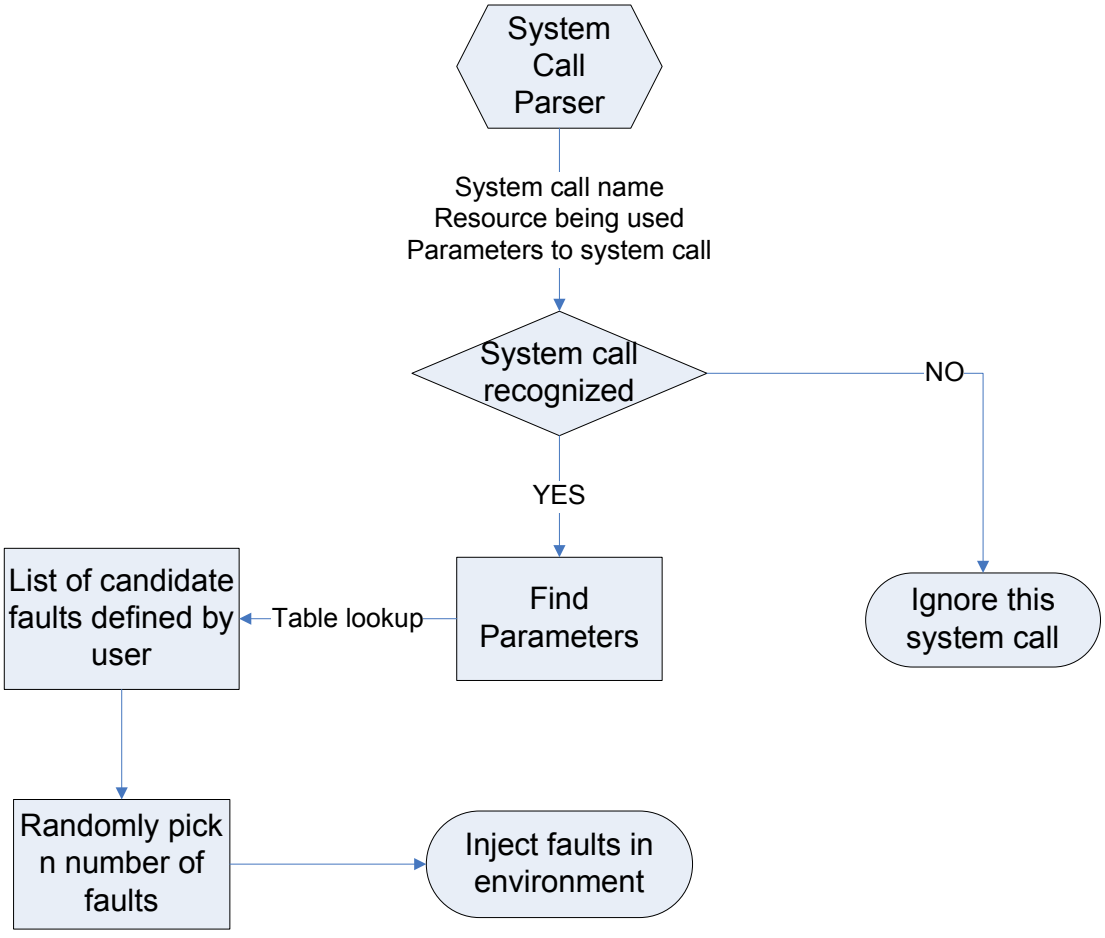

Figure 2 Fault selection algorithm

The application is executed for a second time, and in this iteration, faults are injected according to the user's input or automatically.

3.2.2 Making the fault visible in resources

Once EFIVA has chosen the type and number of faults to inject, these faults need to be made visible in the environment at appropriate times in the program's execution, i.e. when that particular resource is used by the application. EFIVA uses a time-based approach to inject faults.

During its first run with Strace, the fault injector stores the time of access of each of the system calls, along with the call parameters. This time is relative to the start of program execution. During the second run, when faults are injected into the environment, EFIVA starts two threads of execution. The first thread injects the faults, while the second runs the program under evaluation. When the fault injecting thread hits time instances that match with those recorded in the first run, it modifies the corresponding environmental resources, i.e. injects the faults. We assume that the two threads of execution are given roughly equal time slices, keeping the timing distortion to a minimum.

EFIVA stores the program output produced during this second execution to a file, and uses it in the verifier module.

3.3 Verifier

The verifier module performs checks to determine if the application execution trace has opened any doors that can be used by an attacker to launch a security violation. As mentioned before, these checks come in two categories, crashes and output matching.

In the former case, EFIVA automatically flags a crashed execution trace as a security problem, especially if the non-fault injected run exited without reporting any problems. While this may not truly be a security policy violation, it is certainly a vulnerability that needs to be fixed, or classified as one that the application writer never expects to encounter.

In the latter case, output matching is used to report a possible vulnerability. EFIVA accepts a simple regular expression term that contains the expected output, and this is matched with the output actually produced by the application.

Figure 3 below illustrates the algorithm used by the verifier to determine if the application's behavior with faults injected is indicative of any vulnerabilities being exploited.

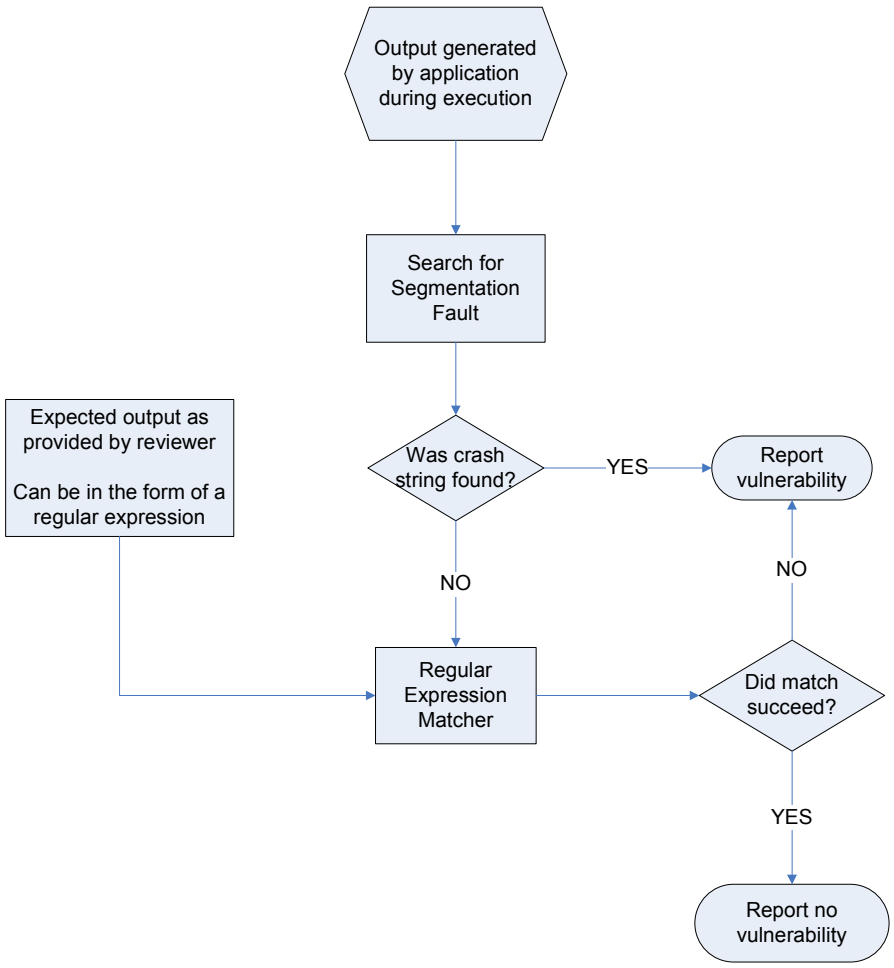

Figure 3 Algorithm to find application vulnerabilities

EFIVA writes the test application output back into a file, allowing users to apply their own conditions to discover additional vulnerabilities. The above-mentioned criteria can be very useful in characterizing application behavior when tested under a changing environment. However, the job of discovering the more contrived and complicated faults continues to remain in the realm of manual testing.

3.4 Case Studies

In order to verify that EFIVA is helpful in finding application vulnerabilities, we ran it on several input programs. This test code was scraped from code written by the first author for undergraduate projects; often at the freshman and sophomore levels.

We picked tests that try to best illustrate the various capabilities that are programmed into the tool. In this section, we outline some of our code samples, the bugs that they hide, and EFIVA's attempts at finding them based on environmental interactions.

All programs and EFIVA were run on Linux. EFIVA itself is written in Java, and calls into our modified version of Strace, which was downloaded off SourceForge.net.

3.4.1 Application crash due to null pointer reference

A DoS (Denial of Service) attack is one of the most common methods of exploiting vulnerabilities. In the case of a console application, this attack is akin to crashing the program repeatedly, thus making it impossible to use. Consider the following code snippet:

```
1     void ReadFromFile ()
2     {
3           FILE * f = fopen ("foo.txt", "r");
4           char buf[100];
5           fgets (buf, 10, f);
6           fclose (f);
7     }
```

Figure 4 Null pointer de-reference crashes the application

This example is simple, yet hides an extremely common source of vulnerabilities. The program does not process `fopen`'s return code making it susceptible to file open failures. If `foo.txt` does not exist, the code as written in Figure 4 will not create the file; the call to `fgets` on line 5 will cause a segmentation fault. Similarly, if the user running the above program does not have the permissions to read `foo.txt`, `fopen` will return null crashing the application at line 5.

EFIVA detects this application's susceptibility to DoS attacks by modifying `foo.txt`'s permissions (setting it to be readable only by *root*) and its existence (deleting it before the call to `fopen`) properties when the application is being executed.

3.4.2 Race condition in file access

Race conditions that originate from file accesses are probably the most significant among those that result in vulnerabilities [12]. The problem stems from the existence of a time delta between the two instances when a file's property (e.g., access right) is checked, to when that property is actually used in the application. Such flaws are referred to as Time of Check to Time of Use (TOCTOU) flaws.

The textbook example of a TOCTOU fault is a setuid[1] program executing under root privileges [13].

[1] Unix has a setuid bit that allows for certain programs to grant users temporary privileges. When an executable file with its *setuid* bit turned on is executed, it assumes the privileges of its owner, as opposed to its executor (the default).

```
1       if (!access (file, W_OK))
2       {
3               f = fopen (file, "w+");
4               // Perform write operations here
5       }
6       else
7       {
8               printf ("Could not open file %s.\n", file);
9       }
```

Figure 5 TOCTOU vulnerability in file accesses

In this example, a race condition exists between the time when `access()` is called on `file`, and when `file` is actually opened. The programmer makes an implicit assumption that `file` remains unchanged between the two calls, which is incorrect! A clever attacker would have the file denote a symbolic link as opposed to a physical file name, and modify the target of the link between lines 1 and 3.

EFIVA's design and implementation makes it ideal to track such issues that arise from an application's environmental interactions. When the tool detects a call to `access()`, it has immediately found a potential injection point. In the fault injection pass, the parameter to the `access()` is set to a symbolic link, and is modified as soon as the function returns. Any future use of the file thus points to the malicious location pointed to by EFIVA.

Vulnerabilities that are born from race conditions in file accesses can be very difficult to detect without source code. At the core of the issue is the extremely small attack window (between lines 1 and 3) when `file` needs to be changed to successfully record an exploit. Our fault injection framework traces application execution making this process a lot more efficient, and easily reproducible.

3.4.3 Output comparisons for vulnerability detection

EFIVA provides users the ability to compare the expected output from an application to that actually produced when it is executed. The mechanism is inherently simple, and has the ability to produce false positives, especially if the expected output is not very well known. However, this is a better first step to finding vulnerabilities than using only a simple metric like checking for an application crash or a segmentation fault.

As an example, consider code snippet in Figure 6 below. The `ListDirectoryContents()` function does not allow a user to view the contents of the root directory. If the user provides a directory name that starts with '/' as is the case for the root directory, the function exits. Assume that the program is being executed in the /home/foo directory, and the user's input is ../../. This input combination corresponds to the root directory, but the code as written above will still list the contents of root!

```
1     void ListDirectoryContents ()
2     {
3           char dirname [100];
4           gets (dirname);
5
6           if (dirname [0] == '/')
7           {
8                 printf ("Cannot list contents of the root
                  directory");
9           }
10          else
11          {
12                DIR * dir;
13                struct dirent *entry;
14
15                if ((dir = opendir (dirname))!= NULL)
16                {
17                      while (entry = readdir (dir))
18                      {
19                            printf ("Entry : %s\n", entry=>d_name);
20                      }
21                }
22          }
23    }
```

Figure 6 Output comparisons for vulnerability detection

The output should have been *Cannot list contents of the root directory*, which was provided in the input to EFIVA. However, the actual output obtained from the above code snippet was much different (the list of files and directories in the root directory). A comparison between the two outputs failed, thus indicating the possibility of a security vulnerability.

These three examples illustrate how EFIVA can be used in a real-world setting to discover potential security vulnerabilities in application. In the future, the tool can be expanded upon to track more system calls, and provide enhanced verifying capabilities.

Chapter 4: Expanding on the Environmental Approach

Chapter 3 addressed the implementation of our EFIVA tool, and described some of the case studies that we used to validate that our approach is indeed workable and has potential to discover security vulnerabilities. With the confidence that dynamic, black box environmental perturbation and fault injection is an approach that can be used effectively to review the security characteristics of an application, we now address another important question. Is an Environmental Approach sufficient? Is there a mechanism to broaden this approach and discover new techniques, which when used independently or in conjunction with environmental perturbation have the potential to do an even better job of discovering vulnerabilities in applications? Through the rest of this chapter, we address these very questions, and propose a scheme that forms the basis for the rest of our work in this thesis.

4.1 The pros of an environmental approach

One of the significant challenges of a black box approach is that a reviewer does not know exactly what resources or actions a program performs at each step. Every program under execution can be traced dynamically for the system calls that it executes. These system calls provide information about the resources that the program accesses, the actions performed on the resource and the parameters that further refine the specific action. For example, consider the open system call:

```
open (filename, arguments)
```

This system call clearly indicates that the program is opening a file with name `filename`, and in the mode as specified by `arguments`, which could be "rw" which indicates that the file is opened for read and write, or "r" which means it is opened for read only, or any other valid combination. Gathering such information using a black box approach is impossible, or a very arduous process.

However, a smart fault injector can choose with ease, and automatically, the type and number of faults to inject when it encounters certain system calls and their corresponding parameters. Tracing the system calls executed by a process provides good insight into how an application has been programmed. Further, environmental fault injection is fully extensible; one can easily add new system calls and their corresponding candidate faults to the fault injector, thus broadening the scope of problems that can be discovered and exploited with this approach.

4.2 The cons of an environmental approach

As discussed earlier, the most significant problem associated with a black box approach to finding vulnerabilities is detecting when a vulnerability has been exposed in the program. The challenge is determining the behavior and output of a program that has been compromised.

In the environmental approach, we used two criteria to detect a vulnerability, an application crash, and difference in the expected and actual program outputs. The application crash is easy to see; no program should crash because of faults injected either in it or in its environment. However, comparing expected and actual outputs is not so straightforward; verifying that a vulnerability has indeed been exposed

involves more than just the comparison between two character streams. Such a technique depends on a human security reviewer to provide the fault injector with expected output from the application for a certain set of injected faults, and is therefore dependent on the reviewer's understanding of the program behavior. The reviewer provided output might be correct or incorrect. If correct, then a mismatch between expected and actual output indicates a vulnerability. On the other hand, if the reviewer is wrong, then a mismatch in the two outputs could mean either the program does not have a vulnerability (program output was correct, reviewer's was wrong), or it indeed has one (both program and reviewer outputs were wrong).

Therefore, output comparisons could show both false positives and false negatives, making the role of the human verification all that more critical. In an environmental approach, until a better, automated solution is found, human verification will remain the primary mode of determining if a program has vulnerabilities. The fault injector can provide hints that at best could ease this process.

Another issue to consider, although obvious, is that environmental fault injection only traces those functions and resources that have system calls associated with them. There are numerous APIs, such as the string library in C, where the functions do not have corresponding system calls, yet are important sources of vulnerabilities. A most basic example would be the `strcpy()` function, which when used without proper bounds checking could lead to a buffer overflow.

4.3 Can we do better?

Environmental interactions are a large source of program vulnerabilities, and we can address them with our fault injection scheme as discussed above. Yet, there are other sources of vulnerabilities as well that need to be addressed if one desires a complete and thorough analysis of a program's security characteristics. Over the past several years, numerous techniques, both static and dynamic have been proposed to find general program bugs, and those that are exploitable, i.e. vulnerabilities. If we look at the security review process through the eyes of a reviewer, the kind of approach that he/she can choose depends entirely on how much the reviewer knows about the program, and how detailed of an analysis is desired.

To aid in this effort, we define a hierarchy of five approaches that extends from all black box where a compiled executable is analyzed, to all white box where the complete source code listing for the application is available. We use a set of four criteria to evaluate their relative effectiveness and usefulness. The next few sections motivate the use of this hierarchical approach with fault injection to find vulnerabilities.

4.4 Why do we need a hierarchy of approaches?

Static techniques, software testing and fault injection can each form a significant component of the software vulnerability finding process. The kind of approach used in this process depends on the available program abstractions and their relative effectiveness in achieving the goals of the reviewer. A program's abstraction

may range all the way from being just a compiled program (EXE), to having some amount of programmer-supplied metadata, to the program's full source code listing.

Our research focuses on constructing a hierarchy of approaches that can be used for finding vulnerabilities in applications given different amounts of program information. In particular, we start by assuming that the only data available about a program is the fully compiled executable, and relax our constraints to create a new level in the hierarchy, enabling better vulnerability analysis. Said differently, we start with a complete black-box approach, enter the grey-box mode, and ultimately end in a full white-box approach, each time reviewing the usefulness and effectiveness of the technique used in that particular level in finding vulnerabilities. At each level, we use fault injection as the tool to perform a security review, and use a set of four defined criteria to compare and contrast each level with its predecessors in the hierarchy. Having separate levels provides a platform for security reviewers to build upon depending on their requirements. These requirements may include, but are not restricted to the amount of time they are willing to spend on the process, the level of confidence that they desire in the software and the amount of information that they have available to perform a review. Even in the presence of an application's full source listing, one may not desire a full static analysis of code, and may be quite happy with just running the program on certain select input combinations. On the other hand, one may want to perform extensive security analysis because of the critical nature of the application under review. Having a multi-tiered approach to finding vulnerabilities in programs allows for different degrees of review based on one's unique requirements.

4.5 Why do security vulnerabilities happen?

A bird's eye view of software development reveals three primary sources of security vulnerabilities in software. First is incorrect program specification and architectural design, the second is poor implementation by the programmer, and the third is environmental interactions. A common example of poor architectural design is parameter tampering in web URLs. An attacker may be able to modify the parameters that are sent to the web server from the client, and view information that he/she is not authorized to access.

The second source of vulnerabilities are programming bugs, and are caused by programmer oversight, carelessness, or sometimes an incomplete understanding of the nuances of programming language features. In the C world, an example of the latter would be the assumption that `strcpy (src, dest)` only copies the portion of `src` that fits in `dest`, when in fact the function performs a blind copy, potentially leading to a buffer overflow. While static analysis tools would easily catch this buffer overflow if provided with source code, finding this vulnerability in a compiled executable may not be as trivial. An example of programmer carelessness would be, not freeing all the memory that a program allocates dynamically, or trying to free a pointer twice. Finally, an example of programmer oversight would be not encrypting passwords, instead storing them as plain text in a file or a database.

The third source of vulnerabilities, Environmental interactions refers to vulnerabilities introduced in an application by the use of resources or information from the environment where it is executed [14].

When a programmer writes code, he/she has a view of the program and an understanding of how control and data flows through it. Quite often, the programmer's view is very different from how the written code actually behaves. It is in this gap between expected and actual application behavior that a program is most susceptible to functional bugs and vulnerabilities [15]. A smart hacker will either modify a program's input, or change its execution environment or apply some other mutation to have the application move in a direction that is different from what the programmer ever expected it to take. This may involve exploring various execution paths, corrupting program data or even modifying sensitive information that an application depends on for its proper functioning.

A security professional can use the very same techniques to identify and fix security vulnerabilities in a program before a hacker compromises it. He/she can use various techniques, such as static analysis, testing and fault injection. We use fault injection as the technique of choice for the various approaches that we discuss in this thesis.

4.6 Fault Injection and a Hierarchy of Approaches

Fault injection provides a framework that allows an external tool or person to inject faults in information that is important to the application, such as in its inputs, or internal program state or even its execution environment. Fault injection is a dynamic

approach, and is extremely flexible, making it an ideal foundation on which to build the techniques that are discussed in this thesis. Faults can be injected anywhere inside an application, i.e. program variables or in external information sources such as program input, or even in the environment where the program is executed. This versatility of fault injection techniques allows us to use it effectively in the whole spectrum of vulnerability finding approaches.

Every application or program can be thought of as a graph, where each node is a state and the edges are transitions between the states. For every state that a program is in, a fault injector can modify parameters and force the application into new states; states that potentially expose exploitable security holes, i.e. security vulnerabilities. While the ability to modify internal program execution paths sounds very encouraging, it makes a very fundamental assumption; information such as the internal state of an application, its parameters and control flow are available for use by the fault injector. The basic premise of our approach is that we start with an executable, i.e. compiled binary code, making deciphering such information extremely challenging, or in some cases impossible. We thus define levels of abstraction to perform a program's security analysis, with each level using different amounts of information. This information includes the view of the program (full source code, partial source code, etc), and extent of documentation available to the security reviewer. The fault injector may infer some of this information automatically, and will depend on the user to provide the rest. Notably, we define five different abstraction levels, Black Box Approach with an Executable, Black Box Environmental Approach, Basic Flow Graph Approach, Flow Graph Approach with

parameter metadata and finally White box approach with Full Application Source Code.

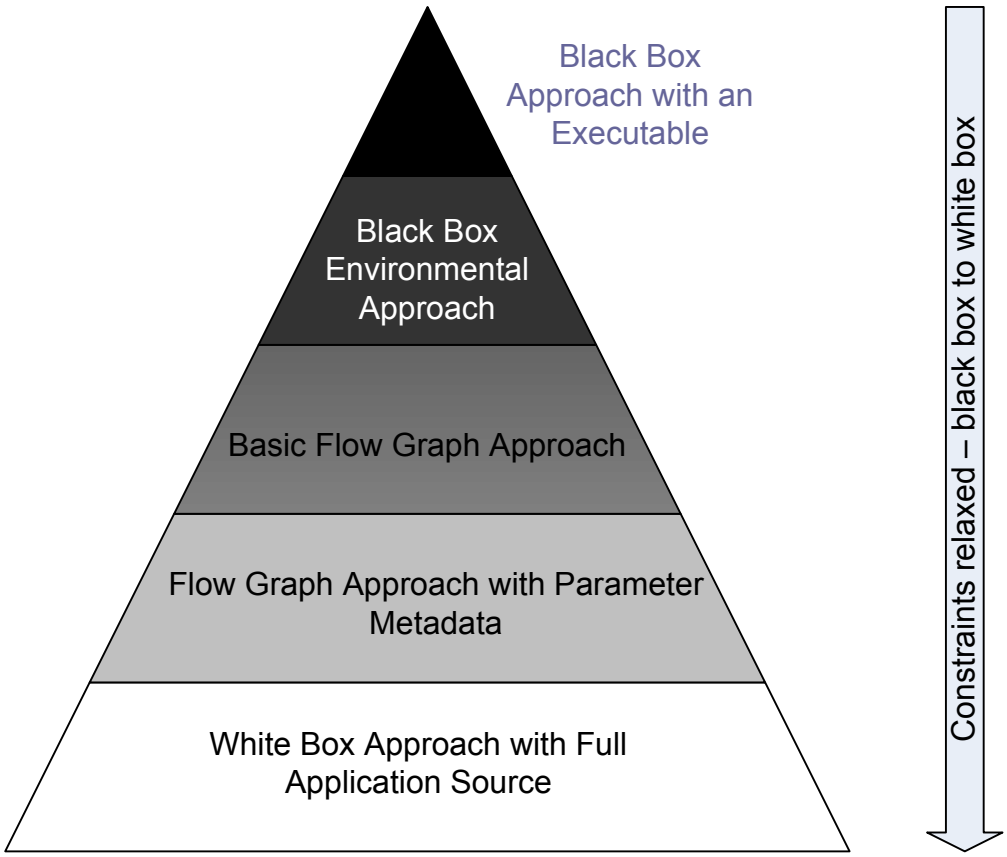

Figure 7 Black Box to White box - A hierarchy of fault injection based approaches for finding vulnerabilities

In each of the levels as mentioned in Figure 7, fault injection can be used with varying levels of usefulness and effectiveness. We evaluate each level based on four fault injection criteria, as follows:

i. *The ability to choose a fault injection point*

   Given an application, how does one know where a fault should be injected so that it affects the execution of the program in a meaningful way?

ii. *Ease of injecting a fault, i.e. the fault injection mechanism*

This is the technique that is used to have the fault injected either in the application or its environment or its input or any other location.

iii. *The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability*

A fault injector can inject faults in arbitrary locations, but analyzing the application's behavior with the fault injected would not be very useful if the fault can never happen under normal usage of the application.

iv. *The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered.*

How does one determine with a high level of confidence that a vulnerability has been discovered in the application by analyzing its behavior under fault injection?

Our set of four criteria was constructed by taking two different views of fault injection; the first being the fault injector's perspective, and the second being that of a security reviewer. The first three criteria are critical for any fault injector; it needs to know where, when and what faults need to be injected, and has to have a means to make the fault appear in the application or its surroundings. The fourth criterion is constructed from the most basic requirement for all vulnerability finding approaches: how does one know if the injected faults have exploited a vulnerability in the application.

In the next chapter, we describe each of the levels in Figure 7 in detail, explore their use of fault injection, and characterize their usefulness in the vulnerability finding process using the four criteria of fault injection defined above.

Chapter 5: The Hierarchy of Approaches in Detail

Chapter 4 explored the relative effectiveness of an environmental fault injection approach, motivated some of the reasons why applications have security vulnerabilities, and came to the conclusion that we could expand upon the environmental approach to develop new fault injection based techniques that can also be used to find security vulnerabilities. It also contained a discussion on the hierarchy of approaches, starting from the complete black box executable view of the program, to the full white box approach where the application source code is available for the reviewer to analyze. Interspersed between these two extremes were environmental fault injection and two techniques that used an internal program representation with some metadata as provided by the application's programmer.

In this chapter, we discuss each of these techniques in detail and evaluate them using the set of four criteria that was also laid out in Chapter 4.

5.1 Black Box Approach with an Executable

At the most basic level, we assume that the only information available to a security evaluator is the compiled EXE. The executable does not have to be on the same machine as the security evaluator; it may be elsewhere on a network, or could even just be a website. The most obvious choice of techniques in the absence of any other useful guidelines is a black box approach where the security evaluator presents an application with several input permutations. These include valid and invalid

inputs, and those with malicious intent. With each injected fault, the program's execution is traced and output is analyzed to determine if a vulnerability was found.

As is quite obvious from a cursory examination of this approach, the state space to be explored is enormous. Using a fault injection approach, the large number of unique faults that can be injected, when coupled with the multiple inputs that programs typically take from the user, leads to a combinatorial explosion and makes for an extremely time consuming and exhausting evaluation process. Automation clearly has the ability to speed up such an approach, and it can be used effectively to ensure some basic security properties. For example, one can easily verify that an application does not crash when subjected to fault injected input. Similarly, if the application prints password information to the console in plain text, then there is a security vulnerability.

This black box approach can be evaluated based on the four criteria stated earlier in Chapter 4.

i. *The ability to choose a fault injection point*

   In a black box approach, there is a single fault injection point, the input to the application. However, the number of input permutations to be considered is enormous, and working with each one of them could be a very laborious process.

ii. *Ease of injecting a fault, i.e. the fault injection mechanism*

   Faults are injected in the application input, which is done rather easily. The fault injector can generate any number of permutations based on criteria laid out by the security reviewer.

iii. *The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability*

Every fault injected in this scheme is totally acceptable because an application should be willing to accept any and all input. A well-written program should not crash because of invalid input, but rather print an error message or exit gracefully. However, given a fault, determining its viability is extremely difficult because of the lack of program knowledge. Most of the faults are educated guesses, but ultimately, in a brute-force approach, they are just guesses. Most of the faults may not turn up any vulnerability in the program.

iv. *The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered.*

This is the most complicated step in an all black-box approach. Without any additional knowledge about a program's features, classifying its behavior as one that indicates the existence or absence of a security vulnerability, is extremely difficult. For example, consider all the metadata that a program writes to a file for its future use. Such metadata may include user preferences, passwords and other important information. If the security reviewer is not aware of the significance of each piece of file content, he/she may miss what is security critical. This may however be caught by a smart hacker and used to compromise the applications.

The vulnerability verification step can be significantly improved if the security reviewer is provided with a detailed specification of the program, such as, the implementation spec created by the application developers. Such a document would provide a lot more insight into the design of each program component, and the interactions between them. Such program details would allow for a more intelligent choice of faults, and potentially reduce the number of faults that need to be injected in the system.

In summary, while a complete black box approach may not prove to be very reliable in finding a vast majority of application vulnerabilities, the level of confidence that one gains by using such an approach may be sufficient in certain scenarios.

5.2 Black Box Environmental Approach

Chapters 2 and 3 described the environmental perturbation approach extensively. Therefore, we only summarize the evaluation of this technique using the four criteria laid out in Chapter 4.

i. The ability to choose a fault injection point

In this approach, the fault injector injects a fault when it encounters a system call that it recognizes, making the process of choosing the injection point easy.

ii. Ease of injecting a fault, i.e. the fault injection mechanism

Environmental perturbation involves changing the external factors that impact an application, such as modifying file ownership, their contents or their existence properties (delete a file, create one when the application does not expect it). These changes are applied to the execution environment, and are thus easily implemented.

iii. *The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability*

The viability of an injected fault is also trivial because changes to the environment are independent of the execution of a program. This is the primary reason why this technique is so useful; changes can happen when an application least expects it.

iv. *The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered.*

This is the most challenging component of a truly dynamic, automated fault discovery mechanism. There is no easy way to determine if a vulnerability has been discovered without a detailed knowledge of the workings of the application, and its error states. However, two useful checks that can be used are, checking for application crashes, and output comparison. If for a given action, the output as expected from an application does not match the actual output observed, then there is the potential for a vulnerability. The ultimate check is still one that is performed by a human security evaluator. The absence of source code in

this black-box approach leads us to assume that the security reviewer is very familiar with the application through its constant use, or is provided with a specification document from the application developers to better understand the program's assumptions and workings.

5.3 Using Program Representations to find vulnerabilities

The previous two techniques, complete black box approach, and the environmental approach, try to find vulnerabilities in applications by looking at their external characteristics and behavior, i.e. from the perspective of an application user. However, a lot of useful information can also be gathered by looking at a program's internal representation, its states and control flow. These internal structures provide a great opportunity for a tool such as a fault injector to modify, with relative ease, the flow of data and control in a program. These changes can be made dynamically at runtime, allowing the fault injector to discover new paths and conditions as variables assume different values inside a program. A program exposes a deviation from its correct behavior by taking different control flow paths than what is expected for certain input. We view such deviations as a manifestation of an exploited vulnerability.

5.3.1 Choosing an internal program state representation

A program's internal representation can be described using various concepts, such as a Control Flow Graph, Data Flow Graph or a Program Dependence Graph. The Control Flow Graph (CFG) is a graphical representation of a program's control

flow and structure [15, 16]. The CFG is a directed graph where each node is a basic block and the edges represent control flow. A basic block in turn is a linear sequence of instructions with exactly one exit. When the processor starts executing a basic block, it continues execution in a single sequence until the end of the basic block; there are no branches or halts. A CFG therefore presents two distinct possibilities to affect change through fault injection. Not only can faults be injected to change the direction of execution that a program chooses, but also to modify program data.

A Data Flow Graph (DFG) is a graphical representation of a program's data flow and structure. It represents the possible changes in the state of data objects, i.e. their creation, use and destruction [15]. This graph does not have any control flow information associated with it, making its applicability in finding vulnerabilities rather limited. In tracking security vulnerabilities, we desire an understanding of a program's actions, for it is usually here that a program is susceptible to attack. The third representation, Program Dependence Graph (PDG) is a hybrid of dataflow and control flow graphs [17]. It is a directed graph where vertices are program statements and control predicates, while the edges correspond to data and control dependencies [18]. A fault injector can determine fault values using the combined data and control flow information so that it not only affects the direction that a program takes, but also the value of its variables, thus potentially creating states that were not previously considered by the programmer.

All three program representations have the potential to be used for finding vulnerabilities, but one may be more effective than the other in helping us achieve our goals. The most basic assumption in our approach is that we start with an application,

a compiled executable, when trying to analyze its security characteristics. Typically, structures such as the CFG, DFG and PDG are constructed from full program source code, forcing us to find a workaround when working with compiled binary code. Tools such as EEL [19] construct a CFG from an executable, but in the absence of source code, the graph is represented using the most basic assembly language instructions. A graph with only assembly instructions makes it very difficult to decipher deep program characteristics, such as data types of variables used in the program, the use of pointers, arrays, etc. Such information can be used effectively in fault injection; for example, integers can be tested for overflow problems, buffers can be tested for buffer overflows and characters can be tested for non-ASCII input. Further, while registers and variables can be modified at the assembly level to discover bugs, proving that such bugs are exploitable vulnerabilities can be quite challenging. Along the same difficulty level is the task of confirming that a suspected vulnerability is indeed a vulnerability. This confirmation step, except for the most basic cases such as an application crash or hang depends on user provided information. It may be less than reasonable to expect a security evaluator to provide any program hints at the assembly level, especially if the data returned from a function is not a simple character or integer, but a more complicated data type such as a class object.

This inability to make a connection between program structures and assembly code in the absence of full source code listing makes using CFGs, DFGs and PDGs as representations of binary/machine code for internal fault injection rather difficult and ineffective. We need an intermediate program representation or additional metadata

to get more information about the behavior and characteristics of compiled code. PDGs that are constructed using this intermediate representation or metadata will be an extremely useful tool in security analysis.

This kind of intermediate representation is provided today by Java (in bytecode), and C# (in Microsoft Intermediate Language, MSIL). The ease of programming offered by these languages and their corresponding managed execution frameworks (Java Virtual Machine, Common Language Runtime), and the increased tendency of software developers to use componentized code means that more and more applications in the future will be written using these new technologies. The intermediate formats contain information about the data types of variables, the names of API functions being invoked and their parameters, string values, and an excellent metadata store from which control and data flow information can be extracted with relative ease. These are exactly the tenets that we desire in a program representation to effectively choose the faults that are injected to find vulnerabilities. Through the rest of this thesis, we use Java bytecode as the foundation on which our techniques are built and analyzed. It is important to note here that our approach is in no way restricted to bytecode. Intermediate information and metadata provided by compilers or even externally by the application programmer could be used effectively in our fault injection scheme. The ideas presented henceforth are general in nature and fully extensible.

The increased availability of metadata and knowledge about program internals though the use of bytecode now allows us to choose between the various representations of application code.

5.3.2 Program Dependence Graph as Internal Representation

The Program Dependence Graph is a hybrid of a data flow and control flow, making this most suitable for fault injection, when compared to the other two representations, i.e. the CFG and DFG. The data flow component of the graph will enable the fault injector to track the use of variables in a program, and thus values, thus allowing for better, more intelligent choices when injecting faults. Similarly, the control flow component allows the injector to force different execution paths on the program by modifying internal variables. The interdependence of control and data flow can be very helpful in finding security vulnerabilities, as we shall discuss in later sections. A PDG can be constructed automatically using one of the several algorithms that have been proposed, including those for object-oriented languages such as Java [20].

5.4 Basic Flow Graph Approach

Having chosen the PDG as the representation of program structure, we now describe our first approach that uses this graph to find security vulnerabilities.

The figure below shows the PDG for one of the possible implementations of a debit () function in a financial system.

```
1     int debit (int balance, int debitAmount)
2     {
3           int tempBal = balance - debitAmount;
4
5           if (tempBal > 0)
6           {
7                 balance -= debitAmount;
8           }
9
10          return balance;
11    }
```

Figure 8 Possible implementation of debit () function in financial applications

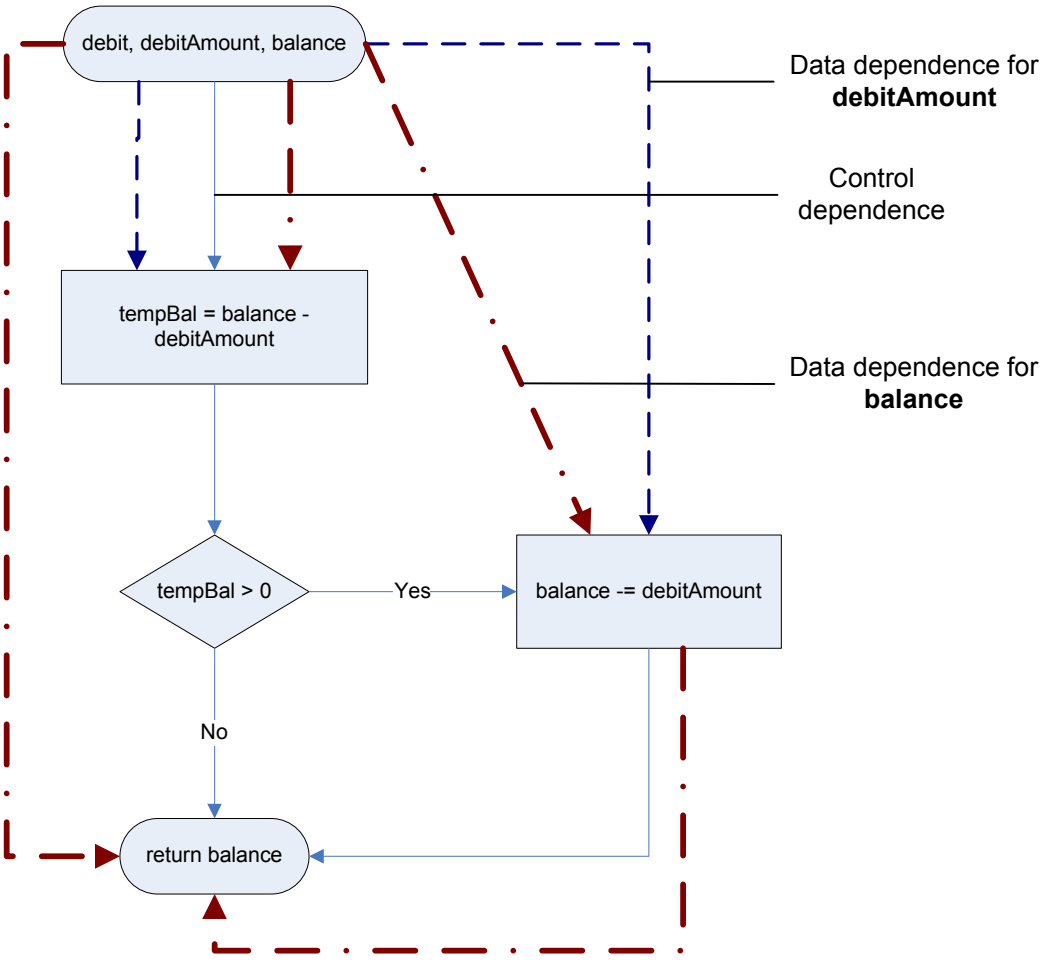

Figure 9 PDG representation for debit () function

As shown in the graph above, the function branches into two different directions based on the value of the variable `tempBal`. The fault injector could thus

modify the values of `balance` and `debitAmount` to have the branch take two different directions in different executions. If the function were to have further conditional statements, then several more paths could be constructed. The return value from this function will be used in other functions, and thus inserting a fault in the return code will automatically propagate itself through the rest of the program. From the source listing (Figure 8) and PDG (Figure 9) above, it is clear that the return value from the function could be any integer value. How does one determine whether a given return value constitutes a vulnerability in the application? In the absence of any other information, the most obvious approach is to let execution continue and wait for a violating condition, such as an application crash, or an error message printed by the program, and then flag the fault(s) injected during that execution run as ones that led to the discovery of an application vulnerability.

However, this approach could lead to a state explosion. It would be impossible for the fault injector to choose a correct subset of faults to be injected that could violate the assumptions of the programmer. What this approach does provide is the ability to change a program's execution profile, and modify its variables and parameters dynamically, and adapt any changes to the path followed by the program until that point in the execution. It also allows for a very modular approach to vulnerability discovery. Each function can be considered individually, and by modifying the parameters passed to it, a subroutine can be isolated as an execution unit. This allows for fault injection on both the local level (a function), and a global level (combination of functions).

This technique, i.e. using a flow graph to find vulnerabilities in an application when measured against the four criteria we defined earlier provides the following results:

i. *Ability to choose a fault injection point*

   Under this approach, every variable is a potential fault injection point; in particular those that can affect the outcome of conditional branches. That is certainly a lot of choices, and could lead to state explosion.

ii. *Ease of injecting a fault, i.e. fault injection mechanism*

   The program would be executed using a debugger service, making the modification of variable values extremely simple.

iii. *The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability*

   The modified variable values can be traced back in the PDG to discover their dependence on external input. While it may be time consuming to compute this for every variable, a potential fault injected can be verified as one that can indeed happen in a program during its course of execution.

iv. *The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered.*

   As discussed earlier, this is the most complicated step of an automated fault injection process. In the absence of any information about the state of variables inside the program that indicate its normal functioning, one would have to depend on criteria such as an application crash or incorrect

output to verify that a vulnerability has been exposed. This step therefore requires human help and intelligence.

The two most significant issues associated with this approach are the inability to be very intelligent about the faults that are injected into the program, and the relative difficulty in verifying that an application's behavior indicates that it has a vulnerability. Both of these deficiencies can be overcome if the user provides more information about the program's behavior.

5.5 Flow Graph Approach with Parameter Metadata

As mentioned above, the two main deficiencies of using a basic flow graph approach exclusively with no additional data can be abated by having the user provide a little more information about the program. A program is a collection of functions that are executed in some sequence. When a programmer uses an API or components written by someone else, he/she does so by invoking functions that are part of the API. We thus consider a function to be a useful abstraction, around which we can build metadata and criteria to help the fault injection process.

5.5.1 Functions and their abstraction

A function can be thought of as a unit comprising three primary components, the input parameters, body of the function, and its return value. In an API, the code comments for each function as provided by the author describes the input parameters, return value, and provides a brief, high-level description of the purpose of the function. These comments sometimes include assumptions made by the function

about program input, and constraints that describe possible return values from the subroutine.

We see program comments as a source of very useful information that can be used to improve fault injection process. If for legal inputs to a function, an injected fault causes the return value to be something other than that specified by the author of the function, we have discovered a vulnerability. Similarly, a programmer's assumptions about inputs to a function may be invalid, or unexpected input not handled correctly, giving another source of program vulnerabilities. The goal of fault injection is to produce output that does not match the constraints as laid out by the programmer. The metadata that is available in a function's comments provides the fault injector hints about faults to inject so that the function produces certain (wrong!) output, while the subroutine's PDG representation provides the fault injector hints on how to go about achieving the same.

5.5.2 Using parameter information in fault injection

Consider the `debit()` function example from Figure 8 and the following constraints provided by the programmer:

a. *Return value >= 0*

   Suppose `balance < 0` and `debitAmount` is greater than `balance`, then `tempBal` is always < 0 and the function returns the original negative valued `balance` variable. This is not in agreement with the constraint provided by the programmer, and will therefore be flagged under our scheme as a vulnerability. The negative balance has the potential to

propagate elsewhere in the code under the programmer's assumption that the balance value as returned by the `debit()` function is always correct.

Consider a slightly different set of function inputs. When `debitAmount < 0`, and `balance > 0`. Then, `tempBal` would be `> 0`, and the function would return a value that is larger than the original balance in the account. This is a problem!

Similarly, assume that the `balance` variable is negative. If `debitAmount` were a negative quantity less than `balance`, then the variable `tempBal` would become greater than zero, and this would be returned as the new balance.

In the last two examples, the function returns positive (although incorrect!) quantities for the balance, and therefore satisfies the programmer provided constraint, which only requires the return value to be greater than or equal to zero. Our approach would not detect the input combinations as that potentially exploit vulnerabilities! Therefore, the success of using a metadata-based approach for finding vulnerabilities is dependent in no small part on the hints provided to the fault injector by the programmer.

One could claim that a programmer knows his code best and only makes calls to the function `debit()` with correct input, i.e. by validating parameters before it is sent to the function, and by validating output after it is returned from it. No matter what these assumptions are, based on the criteria provided for the input and output, this function when considered as a separate entity exhibits vulnerable behavior. There is always the possibility that the

code's ownership changes in the future, or it is opened up as an API or web service, at which point the damage that could be done by malicious code as listed above is enormous. The fault injector would be better off flagging such cases as potential vulnerabilities, and the effect of the incorrect return values from the function can be studied in the program's flow graph at points were `debit()` is called.

b. *Input balance > return value*

This is another possible user specified constraint, and it shows similar vulnerabilities to the ones discussed in part a. For example, consider that `balance` were negative, and `debitAmount` is a negative value less than `balance`. The difference, `tempBal` would become positive, and the function would thus return a positive value. However, this is clearly in violation of the user provided criterion, which specifies that the input balance (which is negative) should be greater than the balance value returned by the value (which in fact postitive). By our formulation, such behavior corresponds to the program having a security vulnerability. The same input combinations was used in a. and it was unsuccessful in finding the vulnerability, yet, under the constraints laid out here would the set of function inputs would help discover the vulnerability. The only difference between a. and b. is the constraint provided by the programmer.

As can be seen from the examples above, the kind of faults to be injected depends on the output that needs to be generated to show that the function's execution is in violation of the assumptions made by the programmer. This makes user provided criteria about return values from functions an absolute requirement. Input criteria, i.e. information on parameters on the other hand are not required, but their use would make finding vulnerabilities a lot easier. Consider the following code execution sequence:

```
1     // Application code
2     // A and B are variables computed
      //before this line
3     retval = FooFunc (A, B)
```

Figure 10 Usefulness of metadata provided by programmers

If the user provides input criteria for `FooFunc`, i.e. defines some constraints on A, and B, the logic until line 2 could have faults injected in it, so that these constraints would be violated. If such a code execution path is found, then the programmer's assumptions are invalid, and thus there is a potential vulnerability in the program.

Therefore, constraints defined on the input parameters and return values of a function are not only useful when that function is checked independently for security vulnerabilities, but also when calls are made to this function elsewhere in the code.

5.5.3 How are fault values chosen?

The discussion around the `debit()` function above considered possible input combinations for variables `balance` and `debitAmount` that violate some of the programmer provided criteria and assumptions. However, if this process is to be

automated, the fault injector needs to determine what fault values can be legitimately inserted at each step when the application is executed. We use a range-based approach, where at each step, the range of values that a variable may take is tracked. Each time an instruction is executed, the possible values for a variable are updated. Consider the following example:

```
1      int function (int x)
2      {
3              x = x + 2;
4              x = x * 2;
5              return x;
6      }
```

Figure 11 Example to illustrate the use of variable ranges

In this example, variable `x` can assume any legal integer value at line 2. This stays the same in lines 3 and 4. However, after line 5 executes, `x` is restricted to the set of even integers. Tracking these range of values for integer variables allows for a final comparison of the constraints as provided by the user to the range of values as deciphered by the fault injector by looking at the program's dependence graph. Any mismatch between the two signals a potential vulnerability. A fault can then be injected anywhere in the execution sequence to trigger an exploit.

5.5.4 Usefulness of bytecode

Consider the code snippet below, which is a simple function that accesses a database, and retrieves results by executing a query.

When the fault injector locates the `executeQuery()` method, it can automatically modify the connection string to include a SQL injection attack. Similarly, since the value returned from the function is a `ResultSet` object, the

programmer could define some rules and constraints about the expected output from any queries that the function executes. For example, the programmer could list the relationship between certain columns in the output with the inputs to the function (and thus the SQL query). Alternatively, one could define a constraint on the number of results returned by the query. Specifying and using criteria as exhaustive and informative as those listed above would be impossible, or extremely hard using just assembly code and without any additional hints or metadata.

```
1     public ResultSet accessData (String name, String username,
      String password) throws SQLException
2     {
3           String query = "Select * from FooTable where name="+name;
4           Connection conn = DriverManager.getConnection(query,
            username, password);
5
6           ResultSet r = conn.createStatement().executeQuery(query);
7
8           return r;
9     }
```

Figure 12 Java code that uses a very simple SQL query

The above methodology could in general be extended to any function, be it an API function in the Java class libraries, or one written by the application programmer himself. The abstraction of a program at a higher level, with more information about the functions invoked, and the input parameters allows for the fault injection scheme to be more advanced, and also allow for a better vulnerability verification process.

This overall technique, which utilizes a PDG along with programmer provided metadata and constraints on function input and return values, can be summarized by looking at our four evaluation criteria.

i. The ability to choose a fault injection point

Faults can be injected arbitrarily at any point in the source code because it is done though a debugger. Injecting a fault is the same as modifying a variable's value.

ii. *Ease of injecting a fault, i.e. the fault injection mechanism*

As mentioned in i., it is just a simple use of the debugger to modify variable values.

iii. *The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability*

The PDG allows each variable to be traced back to the sources on which it is dependent, such as external input, or their un-initialized use in a function. Once such a source has been found, the injected fault becomes entirely viable.

iv. *The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered.*

This ability is one of the biggest gains of this method. The moment the fault injector discovers that one of the criteria provided by the user about input parameters to a function, or a function's return value do not match up with the information inferred through analysis of the PDG, the program suffers from a potential vulnerability. This is because a programmer's assumptions and understanding of the behavior of a program manifest themselves on the constraints that he/she defines, and when these are violated, a gulf appears in the programmer's view of the program and the

actual behavior based on source code. This is a source of program vulnerabilities.

5.6 White Box Approach with Full Application Source Code

The final level in our hierarchy is when an application's entire source code listing is available for the consuming application. In this case, one could use both static and dynamic techniques to discover program vulnerabilities. Having metadata as described in the previous section could still be useful because it would enable a tool to find the differences between the programmer's assumed and actual implementations. Dynamic techniques such as environmental perturbation would be a good complement to static verification methods, as they deal with the effect of changing external factors on an application. Such conditions may be difficult to infer by just looking at the source code.

The availability of source code also allows for a unique form of fault injection. Application code can be modified so that when the program starts execution, control is short-circuited to a part of the program where a preliminary analysis by the user identified a potential vulnerability. This portion of code could then be subject to faults injected through the program input. Modifying source code to implement such short cuts has the potential to reduce the amount of time spent evaluating the application because unnecessary code is ignored; such an optimization would be impossible if the application were only available as a compiled executable. When measured in terms of the four fault criteria introduced earlier, we make the following observations:

i. *The ability to choose a fault injection point*

With the full source code available, a reviewer will be able to analyze the program in detail and determine exactly the fault injection point desired. Source code can then be modified to have control jump directly to location with suspected vulnerabilities and this can be tested using faults injected either in one of the program variables or external input.

ii. *Ease of injecting a fault, i.e. the fault injection mechanism*

A fault can be injected anywhere inside the program by modifying source code or outside it by changing inputs or environmental resources.

iii. *The ability to determine if an injected fault is actually viable, i.e. is a bug exploitable thus making it a vulnerability*

An analysis of the source code would reveal if there is a control flow path to the vulnerable segments, and an input combination that forces the program down this path. Once such a path is discovered, it can be verified by injecting a fault and tracing program execution.

iv. *The ability to verify that the application's behavior with the fault injected indicates that it has been compromised and a vulnerability has been discovered.*

With access to the full source code, one should be able to determine the security policy assumptions of the application. When the program behaves differently from the assumptions, then a vulnerability has been found.

In summary, when full source code is available, any number of static techniques and tools [4, 5, 6] can be used, and dynamic approaches are a worthy complement to static approaches. They can expose certain sources of vulnerabilities (for example race conditions) that could be missed by a purely static analysis of the source code. It is up to the security reviewer to choose the balance between the dynamic and static approaches that can be used when one has full access to the program's source listing.

5.7 Comparative Summary of the five different approaches

The table below summarizes the advantages and shortcomings of each of the five approaches that were described above. The columns represent the approaches, while the rows represent the four criteria that we defined earlier to aid us in evaluating each technique.

	Complete Black Box Approach with an executable	**Black box Environmental Approach**	**Basic Flow Graph Approach**	**Flow Graph Approach with Parameter Metadata**	**White Box Approach with Full Application Source Code**
i	Application Input. There are numerous possible input combinations potentially leading to state explosion.	Environmental resources depending on system call executed. Fewer insertion points, but every fault is important because the environmental resource in which the fault is injected is	Every variable, especially those that affect control flow. For variables, try different possible values based on branch conditions, thus forcing	Every variable in program. The metadata on return values and parameters can be used to pick intelligent fault values, such that the conditions as provided by the programmer are violated.	Source code when analyzed would reveal possible vulnerable sections in the program. The entry points to these code sections, for example, the enclosing function would

		definitely accessed. The faults to be injected are chosen based on the system call made, and the parameters to the function.	new execution paths. This could lead to state explosion, as amount of intelligence used in injecting faults is minimal.	Choosing such a value requires more analysis, but it also provides better results.	then become a fault injection point.
ii	Very easy because application input can be easily modified	This is complicated because system calls need to be hooked and fault injected at the appropriate times. The initial fault injection framework is a little complicated, but once implemented, actually modifying the environmental resources is simple.	This approach would be executed under a debugger, thus making modifying variable values extremely simple.	This is also executed under a debugger, and modifying variables is easy.	Faults can be injected anywhere external to the program, such as input or the environment using techniques explored in previous sections. Source code can also be modified to inject faults inside the program.
iii	The viability of a fault is poor because most of the faults are injected in a brute-force manner without using a lot of knowledge about the application.	In this approach, faults injected are more viable because we know exactly which resources a program accesses, and how it uses them.	The PDG allows any injected fault to be traced back to external input or to an un-initialized variable. Once such a source has been identified, the fault is entirely viable.	The viability measure is the same as the previous technique (Basic Flow Graph).	An analysis of source code would reveal if the vulnerable sections of code are accessible, and would be able to find the corresponding input combinations to make this happen. A fault injected

					to execute this portion of code then becomes viable.
iv	Verifying that certain behavior indicates a vulnerability is extremely difficult without completely understanding the program's characteristics. Simple cases such as application crashes can be tracked easily, but not more complicated ones such as information sent out over the network.	Just like the black box approach, without any help from a human reviewer, classifying the non-trivial cases as vulnerabilities is difficult. If the security reviewer actually provides the expected output, then it can be compared against actual output, and any difference flagged as a potential vulnerability.	For the same reasons as the previous two approaches, the lack of information about a program can make the vulnerability classification process difficult.	This approach is unique from the others in that one has almost all the information necessary to determine if a vulnerability has been discovered. The moment the fault injector is able to find a viable fault that violates the conditions laid out by the programmer, a potential vulnerability has been discovered.	With the entire source code available, a reviewer has a full understanding of the program's security assumptions and framework. Any discovered violations of this framework would then make the program vulnerable to attack.

Table 3 Comparison between different fault injection approaches

Chapter 6: Conclusions and Future Work

6.1 Conclusions

With the ever increasing complexity of software, and developers' reliance on code not written by them (but which is not always well tested and security analyzed either), the need for a security analysis framework that can be used to analyze and evaluate the security characteristics of an application or library is urgent. This problem can be approached from any number of different directions, either static analysis or dynamic schemes such as software testing and fault injection. The use of fault injection in finding security vulnerabilities is a nascent field and provides a lot of opportunity for innovation and improvisation of existing approaches.

Every program executes in an environment, and uses information and data that it collects by interacting with its surroundings. This presents a significant opportunity to find program vulnerabilities by injecting faults in resources as they are used by programs during runtime. Towards achieving this goal and testing our hypotheses, we developed EFIVA, Environmental Fault Injector and Vulnerability Analyzer, which perturbs the environment and uses all three steps of a fault injection process: identifying a fault injection point, injecting the fault, and ultimately verifying that a vulnerability has been exposed in the program.

While environmental perturbations are certainly a significant source of vulnerabilities, other factors, such as an application's use of API functions that do not necessarily interact with the environment play an important role in determining the

application's resilience to attack. We therefore propose a more extended hierarchy of approaches that a security reviewer can choose from depending on the application's available abstractions. Clearly, the more a person knows about a program and its behavior, the better ability he has to analyze the program's security policies to find potential vulnerabilities and attack points. In realization of this fact, the different levels of our hierarchy each assume different program abstractions and amount of information available about the application.

Such a hierarchy would not be useful unless one is able to compare and contrast each approach. We therefore use a set of four criteria to analyze each technique's strengths and weaknesses. Such a comparison will allow any person to choose a technique that is right for their situation depending on how much is known about the application, the time available to perform the security review, and ultimately the level of confidence that is desired in the robustness and security properties of the application.

6.2 Future Work

The work presented in this thesis can be expanded in the future, and we describe some of the possibilities below.

i. Expanded environmental interactions

Du and Mathur's results in [10] suggested that the number of security vulnerabilities associated with network and inter-process communication were significantly lesser than those that were attributed to file system interactions. This conclusion led us to implement only file system

interactions in our fault injector. With attackers inflicting damage of increasing magnitude and applications become increasingly connected over the network, network communication will be a critical source of vulnerabilities in the future, and should be modeled in our fault injector.

Similarly, there are other sources of vulnerabilities in programs such as multiple threads of execution and resources such as the system clock that can be modeled as well.

ii. An improved vulnerability verification scheme

A critical limitation of the black box environmental scheme, and the black box approach with an executable is the inability to claim with a high level of confidence that a program's behavior with a fault injected does indicate the presence of a security vulnerability.

While output matching is a start, tests that are more conclusive need to be developed so that one can claim with confidence that an application has been compromised. A possible approach could be the use of application specifications to infer expected behavior, but a standard format needs to be constructed that all program specifications should adhere to. This enters the realm of formal verification methods, where a lot of work has already been done to address similar problems.

iii. Objected Oriented concerns and a specification scheme for parameter metadata

In our flow graph with metadata approach, we assumed that the programmer provides constraints that he/she believes to exist on the input

parameters, and return values from a function. For return values and parameters that are objects, a programmer might be reluctant to expose hidden fields in the corresponding class. This is a valid concern, because providing information about private variables violates the encapsulation properties of Object Oriented Programming. However, providing such a capability could improve the vulnerability finding process.

In addition, a scheme needs to be developed for representing all of this metadata information provided by the programmer. One possibility is the use of XML which has been gaining increased momentum and is now the standard format for several web technologies such as WSDL, XML-RPC and RSS.

iv. More approaches in the hierarchy, and domain specific techniques

We currently have five approaches in our hierarchy, but there is always room for more. With greater refinement of the constraints and abstractions of programs, one may be able to include more levels so that a reviewer has an increased set of options to make the right choice.

Further, our approaches are good for generic, executable applications, but with an explosion in the number of websites and online commerce, web applications are a significant class of programs that need analysis. The kind of security problems that they face may be very different from what console applications experience, especially with issues such as authentication, session and state. Therefore, our generic approach

proposed here can grow to include more domain specific applications, and address their requirements in detail.

Bibliography

[1] S. Panjwani, S. Tan, K. Jarrin, and M. Cukier, *An Experimental Evaluation to Determine if Port Scans are Precursors to an Attack*, in Proc. International Conference on Dependable Systems and Networks (DSN-2005), Yokohama, Japan, June 28-July 1, 2005, pp. 602-611.

[2] A. Adelsbach, D. Alessandri, C. Cachin, S. Cresse, Y. Deswarte, K. Kursawe, J.C. Laprie, D. Powell, B. Randell, J. Riordan, W. Simmonds, R. Stroud, P. Verissimo, M. Waidner and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D1*. 2000.

[3] A. Adelsbach, D. Alessandri, C. Cachin, S. Cresse, Y. Deswarte, K. Kursawe, J.C. Laprie, D. Powell, B. Randell, J. Riordan, W. Simmonds, R. Stroud, P. Verissimo, M. Waidner and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D2*. November 2001.

[4] D. A. Wheeler. *FlawFinder* http://www.dwheeler.com/flawfinder

[5] *Secure Software Solutions. RATS - rough auditing tool for security*. http://www.securesw.com/rats/ 2001

[6] J. Viega, J.T. Bloch, Y. Kohno, G. McGraw. *ITS4: A static vulnerability scanner for C and C++ code*, acsac, p. 257, 16th Annual Computer Security Applications Conference (ACSAC'00), 2000.

[7] G. McGraw. *Testing for Security During Development: Why We Should Scrap Penetrate-and-Patch*, in IEEE AES Systems Magazine. April 1998.

[8] A. K. Ghosh, T. O'Connor, and G. McGraw. *An automated approach for identifying potential vulnerabilities in software*. in Proc. 1998 IEEE Symposium on Security and Privacy, Los Alamitos, CA, USA, pp. 104-14, 1998.

[9] N. Neves, J. Antunes, M. Correia, P. Veríssimo, R. Neves. *Using Attack Injection to Discover New Vulnerabilities*, submitted to International Conference on Dependable Systems and Networks (DSN06), Philadelphia, PA, June 25-28, 2006, to appear.

[10] W. Du and A. P. Mathur. Testing for Software Vulnerability Using Environment Perturbation. In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)

[11] W. Akkerman et al. *Strace*, http://www.liacs.nl/~wichert/strace/. March 2006.

[12] G. McGraw and J. Viega. *Building Secure Software : Race Conditions*. Addison Wesley, 2001

[13] M. Bishop and M. Dilger. *Checking for Race Conditions in File Accesses*, Computing Systems 9 (2) pp. 131-152, Spring 1996.

[14] W. Du and A. P. Mathur. *Testing for Software Vulnerability Using Environment Perturbation*. In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)

[15] B. Beizer. *Software Testing Techniques*. 2nd edition. Van Nostrand Reinhold, 1990

[16] F. E Allen. *Control flow analysis*. In Proceedings of a Symposium on Compiler Optimization. SZGPLAN Not. 5, 7 (July 1970), 1-19

[17] F. Tip. *A Survey of Program Slicing Techniques*. Technical Report: CS-R9438. 1994. CWI (Centre for Mathematics and Computer Science) Amsterdam, The Netherlands, The Netherlands

[18] J. Ferrante, KJ Ottenstein, J.Warren. *The Program Dependence Graph and Its Use in Optimization*. ACM Transactions on Programming Languages and Systems, 1987

[19] J. Larus, E. Schnarr. *EEL:Machine-Independent executable editing*. ACM SIGPLAN Notices. Volume 20, Issue 6. p291-300. June 1995

[20] F. Umemori, K. Konda, R. Yokomori, K. Inoue. *Design and implementation of bytecode-based Java slicing system*. Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003.