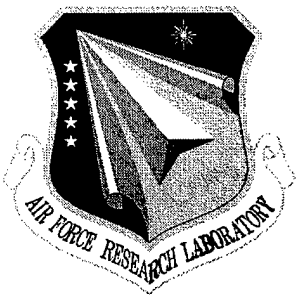
AFRL-IF-RS-TR-2000-159
Final Technical Report
November



QUANTIFYING MINIMUM-TIME-TO-INTRUSION BASED ON DYNAMIC SOFTWARE SAFETY ASSESSMENT

Reliable Software Technologies

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. A0 D331

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-159 has been reviewed and is approved for publication.

APPROVED: 

JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: 

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

QUANTIFYING MINIMUM-TIME-TO-INTRUSION BASED ON DYNAMIC SOFTWARE SAFETY ASSESSMENT

Jeffery M. Voas, Gary McGraw,
Anup Ghosh, Frank Charron,
Michael Schatz, and Tom O'Conner

Contractor: Reliable Software Technologies
Contract Number: F30602-95-C-0282
Effective Date of Contract: 28 September 1995
Contract Expiration Date: 27 September 1998
Short Title of Work: Quantifying Minimum-Time-To-Intrusion Based on Dynamic Software Safety Assessment
Period of Work Covered: Sep 95 - Sep 98

Principal Investigator: Jeffrey M. Voas
Phone: (703) 404-9293
AFRL Project Engineer: John C. Faust
Phone: (315) 330-4544

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by John C. Faust, AFRL/IFGB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1998	Final 28 Sep 95 - 27 Sep 98

4. TITLE AND SUBTITLE

QUANTIFYING MINIMUM-TIME-TO-INTRUSION BASED ON DYNAMIC SOFTWARE SAFETY ASSESSMENT

5. FUNDING NUMBERS

C - F30602-95-C-0282
PE- 62301E
PR- C929
TA- 02
WU- 01

6. AUTHOR(S)

Jeffrey M. Voas, Gary McGraw, Anup Ghosh, Frank Charron, Michael Schatz, Tom O'Connor & Brian Sohr

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Reliable Software Technologies
21351 Ridgetop Circle, Suite 400
Dulles, VA 20166

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFGB
525 Brooks Road
Rome NY 13440-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2000-159

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: John C. Faust/IFGB/(315) 330-4544

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents an overview of the results of a three year DARPA-sponsored effort investigating dynamic software security analysis. This research effort resulted in the design and implementation of two major tool sets (FIST and VISTA), each comprised of many individual tools, and the development of a methodology that provides the capability to perform a thorough security analysis on a piece of security-critical software written in C or C++. The Fault Injection Security Tool (FIST) automates white-box dynamic security analysis of software using program inputs, fault injection and assertion monitoring of programs written in C and C++. The VIsualizing STatic Analysis (VISTA) Tool provides a way of viewing and navigating static analysis properties of a program. Together these tools provide static and dynamic analysis capabilities that can identify security vulnerabilities in source code before its release. However, a major research issue remains. Though the current approach is able to discover security vulnerabilities through a process of fault injection and dynamic monitoring, the tools themself are not able to determine whether such an event could occur through standard attacker input at the program interface. This effort only scratched the surface of work on this important problem.

14. SUBJECT TERMS

Software Security Analysis, Software Fault Injection, Information Warfare, Static Data Flow Analysis, Program Data Flow Visualization

15. NUMBER OF PAGES

44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

1 Introduction

This report presents an overview of the results of Reliable Software Technologies' three-year DARPA-sponsored effort investigating dynamic software security analysis. The research effort resulted in the design and implementation of two major tool sets (FIST and VISTA), each comprised of many individual tools. Both tool sets will be explained in this document. A methodology was developed so that a security analyst can use the tools to perform a thorough security analysis on a piece of security-critical software written in C or C++. Together, the tools and methodology developed during the project present a novel approach to design for security. The core concept developed throughout the project is to begin security analysis as early as possible in an application's lifecycle in order to uncover and mitigate security problems before the application is fielded.

The current system provides a powerful analysis capability closely related to a proven approach to software safety analysis [23]. Given a security-critical software system, FIST and VISTA can help an analyst determine important security criteria. It is important to note, however, that fault injection for safety analysis has proven to be more fruitful than fault injection for security. The main difference between the two lies in the fact that security involves an intelligent malicious adversary while safety does not. Another important factor is the common application of fault tree analysis in software safety (which can help determine which perturbations to use) and the lack of a similar corresponding framework for security.

The major goal of this work, to create a complete system for helping developers and analysts perform security analysis on source code, was partially met by the project. Many of the more tedious aspects of source-code–based security analysis have been captured by our current set of automated tools. However a major research issue remains. Our current approach is able to discover security vulnerabilities through a process of fault injection and dynamic monitoring, but the system itself is not able to determine whether such an event could occur *through standard attacker input at program interfaces.*

After developing our tools and experimenting with them extensively, we developed a final set of experiments aimed at generating interface inputs to exploit a simulated vulnerability. Discovering a set of such inputs would be tantamount to creating an exploit script. It would also demonstrate in a very clear manner the need for mitigating measures inside the code. We only scratched the surface of work on this important problem. We hope to continue this research through additional initiatives.

1.1 Organization

This report is organized around a presentation of the two major toolsets developed during the project (FIST and VISTA). We will begin with a definition of the problem, a brief presentation of related work, and a definition of some terms that have helped us clarify our thinking about the problem. Next we will present the Adaptive Vulnerability Analysis (AVA) algorithm, the theoretical engine of FIST. The Fault Injection Security Tool (FIST) itself will be explained at a high level and some results of FIST experimentation will be reviewed. After a discussion of results gleaned from FIST alone, we turn to a discussion of the VIsualizing STatic Analysis (VISTA) toolset. Based on extensive use of FIST in a number of analyses, VISTA was designed and implemented to make the methodology more efficient and easier for an analyst to use. VISTA is applicable both before and after FIST, and it increases the power of the toolset as a whole. Finally, we present the results of

a preliminary experiment addressing the "back to the inputs" problem. A solution to this problem would have drastic paradigm-enabling implications for fault injection, regardless of the domain.

2 Defining the Problem

The original computer security defense strategy, circa 1970, was appropriately termed "penetrate and patch." At that time, defense was entirely *reactive* — something that happened only after an attack was detected and some damage had already been inflicted. Penetrate and patch was followed by a series of more advanced defensive techniques (*e.g.*, real-time intrusion detection and auditing tools). Unfortunately, a recent proliferation of sophisticated threats has caused defensive security schemes to come full circle, back to where they began twenty-some years ago. Penetrate and patch has once again become the status quo.

The problems in information security are more difficult to understand than those of other certification/assurance disciplines such as software safety or failure tolerance. The fact that a security threat is *malicious* adds subtleties and challenges that are different from those usually encountered in software quality assurance. For example, it is rather obvious that an "unsafe" event has occurred after an aviation disaster; but security intrusions are far less observable, and are often virtually undetectable. In a keynote address at the Eleventh Annual Computer Security Applications Conference (ACSAC'95) held in New Orleans, LA [24], Paul Strassmann stated that only between 1 in 400 and 1 in 1,000 attacks are detected. Because security violations are so hard to detect, there is a shortage of good data about them.

The detection of malicious threats is one complication. The nature of those threats is another. Unlike real military intrusions, software intrusions are "virtual." Counterintuitive though it may seem, an unsuccessful software offensive almost always strengthens the attacker (by way of gained knowledge), and does *not* strengthen the site attacked (by way of weakening the attacker). In traditional military intrusions, an unsuccessful offensive usually weakens the attacker at least as much as the site attacked. Furthermore, in traditional war strategies, the potential for retaliation provides an important deterrent to attack. On the information battlefield, however, the fear of retaliation is minimal at most, and does not affect the balance of power. Most information security techniques used today are either based on the outdated tactics of twenty years ago, or are based on tactics that apply only to conventional warfare, not to information warfare. As a result of these shortcomings, we are left weakened and ill-prepared for defense.

This report details our adaption of a software failure tolerance metric to security measurement. The aim of this work is to provide the theory that underlies a security assessment methodology which we call *Adaptive Vulnerability Analysis* (AVA) and to present a prototype for a vulnerability assessment prototype tool. AVA provides a relative measure of software security. Though AVA may fail to account for especially clever intruders who create *new* malicious threats from scratch, it is certainly capable of simulating many important *previous* security threats and (with some luck) can also be used to detect some unknown vulnerabilities. Our approach allows information system vendors to know *a priori* whether their systems are secure against a predefined set of threats, $T = \{t_1, t_2, ...t_n\}$, where T includes recurrent threats that are commonly encountered. T is open-ended in the sense that when novel intrusion schemes do surface and are debugged, T can be augmented so that they are included during security assessment. Since the evaluated metrics will vary with different sets T, we label the method "adaptive". We attempt to simulate novel threats as well as known threats during the application of AVA.

2.1 Prior Art

A number of techniques have evolved out of the software engineering discipline for analyzing software. In this section, the work of research groups from the University of Wisconsin and the University of California (Davis) in applying software analysis techniques for security assessment is summarized. Other pioneering work in this area was performed by researchers at the COAST Laboratory at Purdue University [21].

A University of Wisconsin group using a tool called Fuzz subjected Unix utilities to random streams of input data. Miller et al. found that "... the failure rate of utilities on the commercial versions of UNIX ... tested (from Sun, IBM, SGI, DEC, and NeXT) ranged from 15-43%" [12, 13]. Most of these utilities failed because of errors in coding. The class of errors that caused the most failures were related to misuse of pointers and array subscripts. For example, incrementing the pointer past the end of an array was a common coding error. Using dangerous input functions, such as the `gets` call, turned out to be the second most common cause of errors that crashed system utilities. Besides being a cause of reliability errors, the `gets` call is notorious from the Morris Internet Worm incident [21]. The reason this call and other related input functions are dangerous is that they do not limit or check the length of the input they read. In the case of the Internet worm, supplying the `gets` call with over 512 bytes of data overruns the stack frame, thus enabling arbitrary input data to be executed [8]. This example emphasizes the dangers of using input functions and system calls that do not check or limit input lengths.

The work by Miller et al. managed to crash several system utilities, including `ftp` and `telnet`, by testing the bounds on input functions. In *Practical Unix & Internet Security*, Garfinkel and Spafford point out the frightening potential for security violations in standard software distributed by vendors, relative to the random black-box testing results from the Miller *et al.* study (pg 705,[8]):

> What is somewhat frightening about the study is that the tests employed by Miller's group are among the least comprehensive known to testers — random, black-box testing. Different patterns of input could possibly cause more programs to fail. Inputs made under different environmental circumstances could also lead to abnormal behavior. Other testing methods could expose these problems where random testing, by its nature, would not.

Research by Bishop and Dilger at U.C. Davis has studied a class of race condition flaws called time-of-check-to-time-of-use (TOCTTOU) flaws [1]. Their research attempted to identify a coding error in which a program checks for a particular characteristic of an object, then takes some action while assuming that characteristic still holds—when in fact it does not. This type of problem is particularly critical in SUID-root programs that attempt to verify that a user has access permissions to one file, then modify it. A cracker can exploit this flaw by creating a link from the file that has been granted access, *e.g.* `/usr/spool/mail/john`, to another file that requires higher privilege for access, *e.g.* `/etc/passwd`. If the cracker is clever enough, he or she can create the link *after* access has been granted and *before* the program accesses the file. This sleight of hand can fool the program into modifying a file it would not otherwise have permitted.

Bishop and Dilger's research has focused on a source-code–based technique for identifying patterns of code which could have this programming condition flaw. One of the limitations reported in their paper [1] for this technique is that the static analysis cannot determine if the environmental

conditions necessary for this class of TOCTTOU binding flaws exist. Their conclusion is that a dynamic analyzer will be able to test the environment during execution and warn when an exploitable TOCTTOU binding flaw occurs.

Another U.C. Davis group is using property-based assertions and software testing techniques to verify security properties of software [6]. Similar to the work presented in this report, these different research projects are employing techniques developed in other areas of software assurance (reliability, safety, testing) to the difficult problems in assuring security in computer systems.

2.2 Related work at Reliable Software Technologies

Reliable Software Technologies is also investigating dynamic security analysis of Windows-NT-based COTS applications for defensive Information Warfare(IW). This work is co-sponsored by DARPA and Air Force Research Laboratory under contract F30602-97-C-0117. The purpose of our effort is to investigate a methodology based on dynamic black-box software analysis capable of revealing existing bugs and new vulnerabilities in COTS software products. We are developing a prototype software tool that automates the discovery of existing weaknesses in executable components in the Microsoft Windows-NT environment. This technology will be delivered in the form of a black-box software analysis tool capable of automatic dynamic software analysis.

Research on the IW project includes a new study aimed at analyzing the robustness of software running on Windows NT systems. The goal of the study is to identify robustness gaps in the application software and operating system software that potentially could be exploited for violations of security. Contributions of the work include a taxonomy of failure conditions and experimental results from robustness testing of software running on the NT platform. The ongoing work under this project is developing a technique for intercepting calls between client applications and utility components in a Dynamic Linked Libraries (DLL).

Another related security project at RST is the NIST Component-based Software Advanced Technology Program (ATP) project which addresses security certification for component-based software used in Internet-based electronic commerce. Securing components used in electronic commerce is one of the most important hurdles that must be overcome if electronic commerce is ever to become a driving force in the consumer market and the software distribution industry. The project aims to develop a certification process for testing software components for security. We are developing a process and a set of core testing technologies to certify security of software components. The manifestation of our product is a stamp of approval in the form of a digital signature.

The key innovations of this project involve developing a Component Security Certification (CSC) pipeline through which a software component will be tested. If the component meets minimal thresholds for security assurance, then the component will be signed using the certifying lab's digital signature. The CSC pipeline involves a combination of white-box and black-box testing processes to providing security assurance. The success of this effort will accelerate the development and deployment of software components used in Internet commerce.

3 Fault Injection Terminology

Throughout the course of our project, we found it necessary to develop some terminology for discussing security vulnerabilities in relation to fault injection. In particular, it became apparent

that we needed to tease apart simulated vulnerabilities exploited by FIST and real vulnerabilities exploited through program inputs. We use the following terms throughout the rest of this report.

Adapted Vulnerability Analysis (AVA) AVA is performed by exercising a target program with a set of test cases. For each test case, the program is executed N times, where N is the number of locations where a bad state is to be injected. For each of the N executions, a different location l is perturbed, and if a violation occurs as a result, a counter for the given location $c[l]$ is incremented. The vulnerability score for each location is a ratio of $c[l]$ to the number of times location l was perturbed. AVA is implemented in FIST.

artificial vulnerability Artificial vulnerabilities are weaknesses in the code that lead to security problems during FIST analysis. We use the overly-strong term *artificial* to emphasize that a vulnerability discovered through fault injection may not in fact have a direct effect on program security. A location is a *potential* artificial vulnerability if it matches heuristics used to identify potential problems (such as being a stack buffer). A location is an *exercised* artificial vulnerability if perturbation at the given location yields a violation during a run of FIST. A location is a *non-exercised* artificial vulnerability if perturbation at the given location can be shown to cause the violation without actual use of the tool.

call graph A directed graph that contains a node for each function/method in the target program and an edge from node A to node B if and only if function A calls function B.

control flow graph A directed graph that contains a node for each statement in the target program and an edge from node A to node B if and only if it is possible for statement B to be executed immediately after statement A.

data dependence (flow) graph A directed graph that contains a node for each variable at each statement in the target program; there is an edge from node A to node B in the graph if and only if node A refers to a definition of a variable, node B refers to a use of the same variable, and there is a definition free path with respect to that variable from A to B in the data flow graph.

fault injection The concept of forcibly changing the program data state at a given location in a program during execution. Fault injection may be used to simulate hardware failure, software faults, bad data values introduced through Commercial-Off-the-Shelf (COTS) function calls or human interfaces, or even random memory corruption. See [23].

function input Values that are passed into a function after it is called during program execution. A function input can be a value passed in via the function argument list or via a global variable.

injected data state The entire program data state at a given location after a perturbation has been applied to a data state value.

location A location refers to a point in a source file where either a perturbation function or an assertion may be specified.

perturbation Perturbations are used to perform fault injection within FIST. A perturbation is a single data state modification introduced during a single execution of a program.

program input A value provided to an executing program from an external source, such as the user, another software component, or a hardware component.

real vulnerability If there exists at least one set of program input values that exploits an *artificial* vulnerability at a given location, then the location is called a *real* vulnerability. Real vulnerabilities are the root cause of all security problems. Exploit scripts attack real vulnerabilities.

statement Statements are sections of code that are executed in sequence; statements are executed for their effect, and do not have values; statements fall into one of the following categories: labeled statements, expression statements, compound statements (or blocks), decision statements, iteration statements, declaration statements, or jump statements.

violation A condition that indicates a violation of the application security policy if the condition is true. Assertions placed in the code can identify violations at runtime.

vulnerability score The vulnerability score for each location is a ratio of security problems encountered at a location to the number of times the location was perturbed when performing AVA.

3.1 An Introduction to Stack Smashing

One of the most useful perturbation functions developed during the project and incorporated into the prototype is a perturbation that simulates a buffer overflow attack. A brief background on buffer overflow and stack smashing helps to set the stage.

Most computer programs need to create sections of memory in which to store information. The C programming language allows programmers to create storage in two different sections of memory: the *stack* and the *heap*. When contiguous chunks of the same data type are allocated, this is known as a *buffer*. C programmers must take care when writing to these buffers that they do not try to store more data in the buffer than the defined length of the buffer. Occasionally, programming mistakes will allow programs to read and write past the bounds of a buffer. When a program writes past the bounds of a buffer, this is called a *buffer overflow*. The C language allows programs to write past the bounds of buffers. There is no run-time check for writing past the bounds of a buffer.

A special case of buffer overflows, called *stack smashing*, occurs when the buffer being overflowed is allocated on the program's stack. The stack in a C program is an internal data structure that maintains records for each function that has been called while the program is running, starting with `main`. Stack smashing attacks target a specific programming fault: careless use of data buffers allocated on the program's runtime stack. A creative attacker taking advantage of a buffer overflow vulnerability in a program can replace a running program with a completely different program. If the program that was subverted was a process running with a high privilege level, the attacker can run a program on the target machine with the same high privileges to do the attacker's work.

3.2 Implementing AVA with FIST

The software vulnerability metric that we have developed is based on observing the impact of *simulated* threats on an executing system. *Adaptive Vulnerability Analysis* (AVA) is a dynamic software analysis algorithm adapted from the extended propagation analysis (EPA) technique used

in assessing safety-critical software [7, 14, 15, 23]. Threat simulation in AVA is accomplished through fault injection during dynamic execution of a target program.

We developed AVA as a concrete way to rate the vulnerability of a software system. Because it is an adaptive measure, the AVA vulnerability measuring technique can be specialized to assess different kinds of threats. The AVA environment can be tuned to better assess a particular piece of software based on vulnerabilities that similar pieces of software revealed in the past. For example, if `httpd` programs have proven to be particularly open to specific attacks, new versions of `httpd` should be tested using such attacks. AVA does not implement a traditional source code-based *static* measure (such as cyclomatic complexity, Halstead's program volume, SLOC, etc. [25]). Instead, AVA measures how software *behaves* when it is forced into anomalous situations. The method by which we analyze how software behaves under malicious threats is closely based on the extended propagation analysis (EPA) algorithm described below. Our main objective is to determine whether a piece of software has weaknesses that can be leveraged into security exploits. Like its EPA ancestor, AVA measures a dynamic characteristic of software.

AVA has been implemented in the Fault Injection Security Tool (FIST). The algorithm is summarized here, see [22] for further detail.

3.3 Formalizing AVA

Let P denote the program under analysis, x denote a program input value, Δ denote the set of all possible inputs to P, Q denote the normal usage probability distribution of Δ, $\bar{Q}$ denote the inverse usage probability distribution, $\hat{Q}$ denote a special input set, l denote a program location in P, and $PRED$ denote the violation predicate.

Algorithm 1:

1. For each location l in P that is appropriate, perform Steps 2-7.
2. Set **count** to 0.
3. Randomly select an input x or input sequence from Q, $\bar{Q}$, or $\hat{Q}$, and if P halts on x in a fixed period of time, find the corresponding set of data states created by x immediately after the execution of l. Call this set $\mathcal{Z}$.
4. Alter the sampled value of variable a found in $\mathcal{Z}$ creating $\breve{\mathcal{Z}}$, and execute the succeeding code on $\breve{\mathcal{Z}}$. The manner by which a is altered will be representative of the threat class from T that is desired.
5. If the output from P satisfies $PRED$, increment **count**.
6. Repeat steps 3-5 n times, where n is the number of input test cases.
7. Divide **count** by n yielding $\hat{\psi}_{alPQ}$, the vulnerability assessment, for each line l. This means that $1 - \hat{\psi}_{alPQ}$ is the *security assessment* that was observed, given P, Q, and T.

3.4 Assessing Program Behavior

The first two steps of the algorithm are very basic. AVA is a source-code–based methodology in which instrumentation is placed between particular statements (called "locations" in the code).

Either an automated system that implements the algorithm (if it is intelligent enough) or the user must tell the system which locations are relevant for fault-injection. The first step is to localize where injection is to occur. Next, a counter is initialized to zero, since we wish to observe how many security intrusions occurred due to the simulated threats that the prototype attempted for a particular location l.

Unlike most software metrics in use today, the AVA software assessment measure does not look at software structure. It looks at software behavior. The algorithm selects test cases (*i.e.*, program inputs) upon which the program will run in Step 3. The inputs can come from different testing schemes that are more likely to trigger a successful intrusion: rare events (with respect to the operational profile), known input sequences that are unusual or likely to be threatening, totally random inputs, or even the operational profile of the system. The fourth step performs the actual program state corruption or syntactic mutation of the code (*i.e.*, this step is the fault injection step). Once the fault that is injected by Step 4 is executed during the analysis phase, the program has been altered in some way. Step 5 then determines if the problem forced during Step 4 causes the program to produce an output event that satisfies our definition for what constitutes as a security violation. If so, the counter is incremented by one. Step 5 is a non-trivial step. That is, it requires definition of a security policy for a program. The definition of a security policy is coded in the form of an assertion that states the program or its environment should never be in a particular state. In general, security policies will vary by application.

Steps 3, 4, and 5 are repeated multiple times (Step 6), which provides a statistical estimate of the frequency that security intrusions occurred from the problems injected in Step 4 with respect to the inputs employed in Step 3. This estimate is calculated in Step 7.

3.5 A Relative Security Metric

AVA's measure of information system security is not an *absolute* metric, such as mean-time-to-failure. Instead, it is a *relative* metric that allows a user to compare different versions of the same system, or to compare different (but similar) systems that have the same purpose.

For the metric we collect, the class of all potential threats is infinite. Clearly, Algorithm 1 cannot simulate all members of the set. Furthermore, for any particular P, it is likely that most members of T are irrelevant. Hence our implementation has two different means for defining the members of T that are relevant:

1. Default perturbation functions, and
2. A Perturbation Function Template that will allow the user to define the idiosyncrasies of specialized threats that are only relevant to P. This Template can be tuned to specific input signals, source-code-based defect classes, and timing.

Because the class of potential, future threats is unknown, any set of default threat classes may not adequately reflect how those threats will affect internal program states. Also, the user of the perturbation function template may not have the foresight to envision certain classes of threats. To attempt to partially accommodate this weakness in the technique, our prototype has a set of default perturbation functions that do not necessarily simulate threats, but simulate random corruptions in the state of the executing program. These random corruptions, when forced into the software, are analyzed to see whether $PRED$ is ever satisfied.

4 Fault Injection Security Tool (FIST)

The Fault Injection Security Tool (FIST) is a working implementation of AVA described above. The tool automates white-box dynamic security analysis of software using program inputs, fault injection, and assertion monitoring of programs written in C and C++. A schematic diagram of FIST is shown in Figure 1. The fault injection engine provides a developer or analyst the ability to perturb program states randomly, append or truncate strings, attempt to overflow a buffer, and perform a number of other numerical fault injection functions. The security policy assertion component provides a developer or analyst the ability to determine if a security violation particular to the software application being analyzed has occurred.

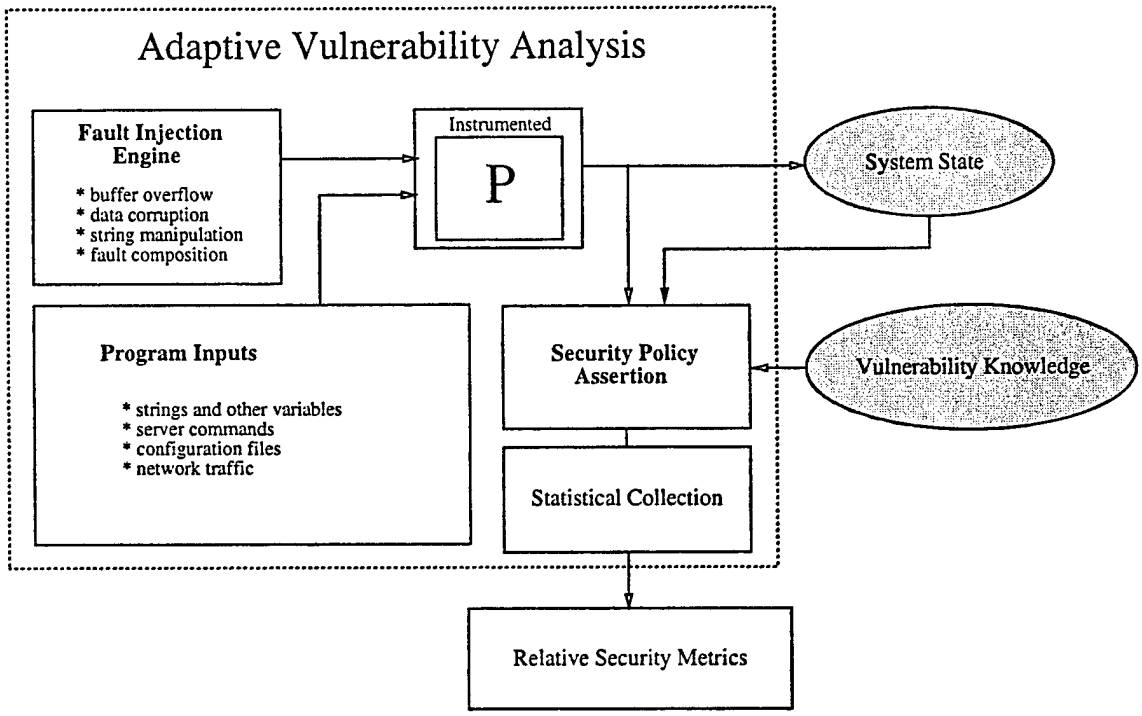


Figure 1: Overview of FIST. A program, P, is instrumented with fault injection functions and assertions which codify security policy (based on the vulnerability knowledge of the program). The program is exercised using program inputs. The security policy is dynamically evaluated with reference to program and system states. If a security policy assertion is violated during the dynamic analysis, the specific input and fault injection function that triggered the violation is identified. Algorithm 1 is used to collect statistics about the vulnerability of the program to the perturbed states. One output from the analysis is the relative security metric $\hat{\psi}_{alPQ}$.

4.1 Using FIST

FIST is a powerful security analysis tool. A side effect of its power is that it can be easily misused. In many ways, FIST is a programming environment. The ability to obtain useful results with FIST relies directly on the ability of the analyst wielding it. To use FIST to its full potential, an analyst should have extensive knowledge of computer and application security.

Obtaining useful results requires a well-defined security policy for the program being analyzed, and an accurate coding of this policy in terms of vulnerability conditions (codified as security assertions). The assertions provide observability into program state during analysis and determine when something violating policy has occurred.

FIST analysis is only as good as a combination of: the input used to execute the program, the perturbation functions selected, and the policy assertions placed in the code. If these data are well-selected, FIST analysis provides a powerful security analysis capability. It is important not to read too much into the results obtained through FIST analysis, however. FIST can certainly indicate when security problems are discovered. But if the results indicate perfect security, chances are the analyst has not exercised the program thoroughly enough.

FIST analysis is applied to a target program that itself can comprise multiple source code files. There are four steps to analyzing a security-critical program with FIST. The four steps are: Instrument, Build, Execute, and Results. (Each of these steps is explained in the following sections.) Before starting with FIST, an analyst needs a copy of the relevant source code and in-depth knowledge of the security policy the program operates under.

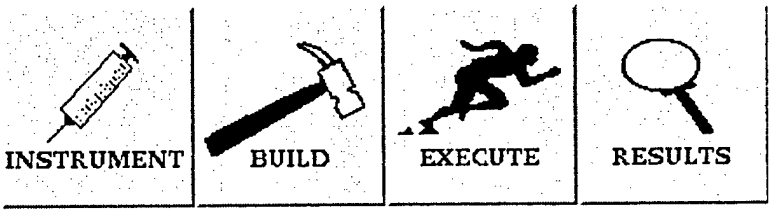


Figure 2: FIST's four steps represented as icons.

4.1.1 Instrument

The Instrument step is performed first. During instrumentation, an analyst specifies the way the target program source code is instrumented during the Build step.

There are two kinds of instrumentation to be interactively placed in the code during this step: perturbation functions and violation conditions. The FIST Source Code Browser allows users to specify locations for both kinds of instrumentation within each source file making up a program.

The analyst can specify several different varieties of perturbation, each of which will be individually applied during analysis. Violation conditions codify application security and monitor both internal program states and external system states during runtime analysis. A screen shot of the Source Code Browser used to place instrumentation is shown in Figure 3. See [18] for details.

Perturbation functions are the FIST mechanisms used to trip up the target program during analysis. Perturbation functions can be specified to simulate programmer faults, malicious usage, etc. Picking good perturbation functions is as much art as it is science. For more information see [23].

Using the Source Code Browser allows an analyst to select:

- locations in the source code to apply perturbation functions
- individual perturbation functions and corresponding parameter values to apply at each location

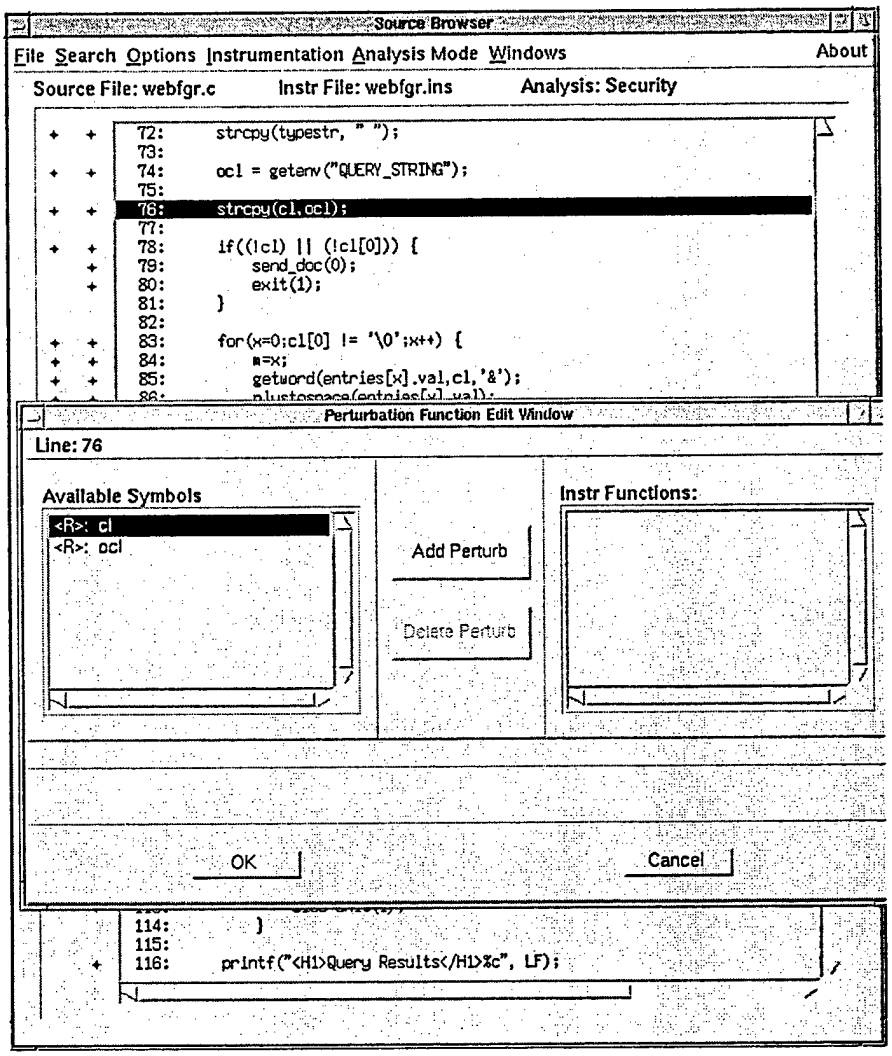


Figure 3: FIST Source Code Browser with Perturbation Function Edit Window open

Violation conditions check internal program states and system states during execution in order to detect if a violation of the security policy has occurred. These violations are specified in the Source Code Browser using either a standard C/C++ expression syntax or a predicate logic based assertion language supported by FIST. Violation conditions codify the security policy of the program under analysis and can monitor both internal program states and external system states.

Once all perturbation locations and violation conditions have been specified with the Source Code Browser, the analyst saves the instrumentation configuration and builds a copy of the target to analyze.

4.1.2 Build

The FIST Source Code Instrumenter, `securetool`, operates as a command line program that can be activated within a make file, making it easy to incorporate within any active development process.

Simply inserting the `securetool` command on the compiler line and linker line serves to instrument source code with the perturbation functions and internal violations specified during the Instrument step.

The FIST Source Code Instrumenter acts as a pre-processor to the compiler that intercepts the source code, instruments it according to the specifications made during the Instrument step, and passes the instrumented source code on to be compiled. The compiled program is instrumented in such a way that the FIST Execution Manager can gather AVA results during execution.

4.1.3 Execute

Finally the program must be executed. Once a program has been compiled with the appropriate FIST instrumentation, security analysis can be automated with the FIST Execution Manager, `securexec`. The Execution Manager component runs tests and collects the security analysis data in a results file.

4.1.4 Results

While the FIST Execution Manager runs security analysis on an instrumented executable target program, a security results file is generated. The FIST Results Browser provides an intuitive way to traverse and review the results. Results are displayed in a hierarchical manner and include a link to the source code (including particular lines where violations have occurred). Figure 4 shows a screen capture of the Results Browser GUI [18]. The FIST Security Report Generator can create a text representation of the results as well.

5 Experimental Analysis with FIST

FIST has been experimentally applied in the laboratory to a number of different security-critical programs. Much of the FIST work has been presented at peer-reviewed conferences and other academic presentations. The following publications and technical reports discuss FIST results in greater detail: [9, 22, 23]. Also see Appendix A which lists all project documents.

To give you a flavor of FIST results, we present results from analyzing five different network services. Network daemons are interesting from a security standpoint because they provide services to untrusted users. Most network daemons allow connections from anywhere on the Internet, opening them up to attack from malicious users anywhere. Network daemons sometimes run with super-user, or `root`, privilege levels in order to bind to sockets on reserved ports, or to navigate the entire file system without being denied access. Successfully exploiting a weakness in a daemon running with high privileges could allow the attacker complete access to the server. Therefore, it is imperative that network daemons be free from security-related flaws that could permit untrusted users access to high privilege accounts on the server.

The programs examined with FIST were NCSA `httpd` version 1.5.2.a, the Washington University `wu-ftpd` version 2.4, `kfingerd` version 0.07, the Samba daemon version 1.9.17p3, and pop3d version 1.005h. The source code for these programs is publicly available on the Internet. Samba, `httpd`, and `wu-ftpd` are popular programs and can be found running on many sites on the Internet. The analysis of those programs was performed on a Sparc machine running SunOS 4.1.3_U. The other programs, `pop3d` and `kfingerd`, are Linux programs found in public repositories for Linux

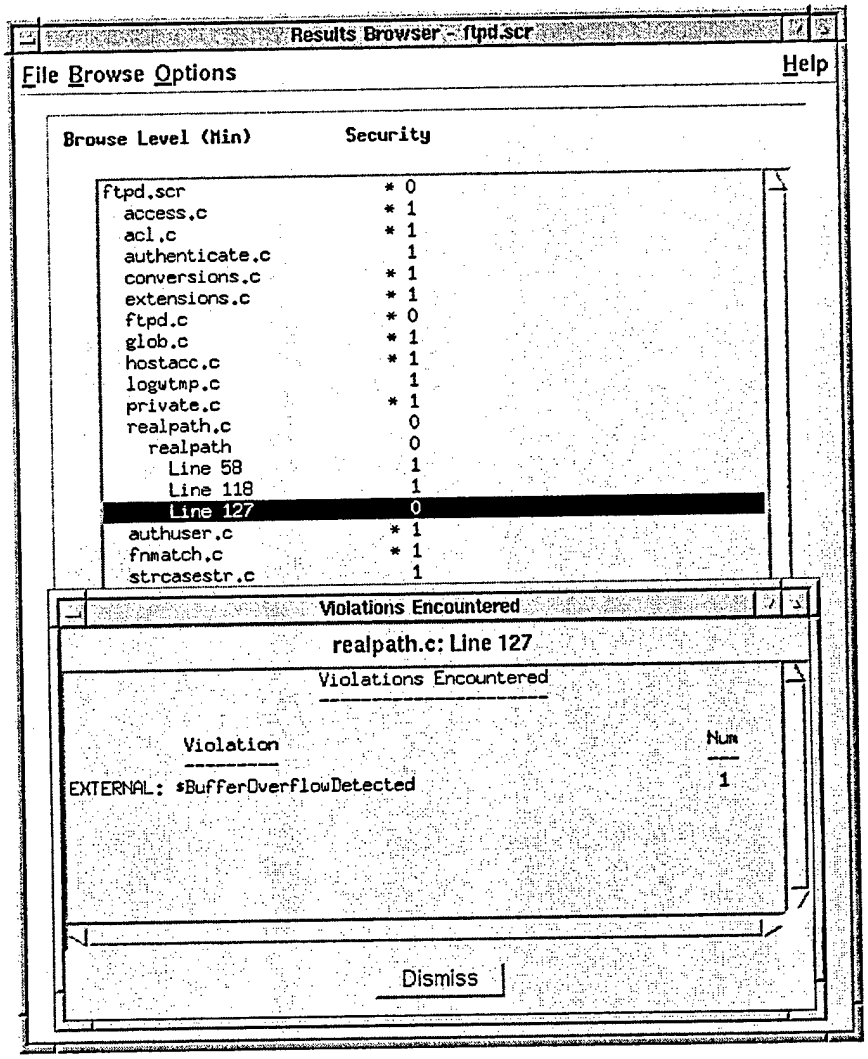


Figure 4: FIST Results Browser with Violations Encountered Window open

source code on the Internet. The analysis of those programs was performed on a Linux 2.0.0 kernel. Three of these programs were selected for analysis because of known vulnerabilities in previous versions of the same or similar software [3, 4, 5]. `kfingerd` was the only daemon analyzed that had not previously been found vulnerable.

The programs were instrumented with both simple fault injection functions as well as the buffer overflow functions where applicable.

A summary of results from the analysis is shown in Table 1. The table shows the total number of instrumented locations together with the number of simple perturbations and buffer overflow perturbations that resulted in security violations. The last column shows the percentage of the functions in the source code that were executed as a result of the test cases employed. Higher coverage results may result in more potential security hazards flushed out through the analysis. The results should not be interpreted to mean that the locations identified in the analysis are necessarily exploitable, only that they require closer examination from the software's developers to

Program	Instrumented Locations	Successful Simple Perturbations	Successful Buffer Overflows	Function Coverage
Samba v1.9.17p3	1264	12	15	45.5%
NCSA `httpd v1.5.2a`	463	27	3	40.14%
`wu-ftpd v2.4`	476	11	3	58.62%
`pop3d v1.005h`	73	2	1	63.64%
`kfingerd v0.07`	146	12	5	38.1%

Table 1: **Results from FIST analysis of network daemons.**

determine if they can be exploited from input and whether fault-tolerant mechanisms should be employed.

Case studies of two of the analyzed network services are summarized. See [9] for more detail. The case studies describe the fault injection techniques applied and the resulting security violations.

5.1 Samba

Samba is a server message block (SMB) daemon for Unix. It allows a Unix file system and printers attached to Unix machines to be accessed by a machine running a Microsoft Windows operating system. Motivating analysis of this daemon was a vulnerability made public in a previous version of Samba. The test data used consisted of commands to navigate the shared Unix file system and retrieve a file. We instrumented 1264 locations in the code. Simple perturbations caused security violations at 12 locations, while the more complex buffer overflow perturbations resulted in 15 security violations.

In one of the test cases, perturbing a Boolean value in the daemon code allowed the client access to the file system with an invalid password. This finding means that the logic at that location in the daemon code had better be correct or else a security hazard may result. The tool also detected a buffer overflow violation. However, upon futher examination of the buffer overflow, it was determined that no user input could exploit this condition.

5.2 `pop3d`

The test case run against the Post Office Protocol 3 daemon for buffer overflow analysis consisted of commands to authenticate a user, open their mailbox, list the contents, retrieve a message, and quit. When testing using simple perturbations, the test case attempted to open a user's mailbox when supplying an invalid password.

Only one buffer overflow and two simple perturbations were detected out of the 73 locations we instrumented. A `strcpy` performed without checking the destination buffer size was the culprit in this case. The buffer being used as the source argument to `strcpy` was populated by the `gethostname()` function. This means that the hostname of the computer this program is running on would have to be extremely long and be the character representation of machine instructions that would run an exploit script or program in order to exploit this potential vulnerability.

6 Are FIST Vulnerabilities Real?

Fault injection is interesting because it places directed experimental stress on running programs and observes the results [23]. The faults injected during analysis are simulated faults involving data state corruption, syntactic mutation, and so on. The question is, to what extent are simulated problems indicative of real problems in the code?

Whenever FIST finds a perturbed data state that leads to a security violation, this is of some interest. It is often important to determine the causes of such problematic events and mitigate the security violations through additional fault-tolerant mechanisms. In addition to this approach, another relevant question to ask in the face of simulated security problems is: *can a dangerous data state be reached by executing the program without fault injection?* The challenge is to determine whether direct program inputs lead to the same problematic data state discovered through injection. Sometimes it may not be possible to reach such a state.

The problem statement above can be expanded slightly for clarity. We are interested in determining any one of the following:

1. program input(s) that lead to the data state (caused by fault injection) that resulted in a violation,
2. program input(s) that lead to the violation itself,
3. convincing evidence that it is impossible to find program inputs that lead to the violation-causing data state,
4. convincing evidence that it is impossible to find program inputs that lead directly to the violation.

If the target data state in item 1 includes all data states that the violation condition depends on, then item 1 implies item 2. It is meaningful to determine item 1 instead of item 2, for two reasons. By focusing on the location where the fault injection occurs, we are closer (in the control and data flow) to the program inputs than we are if we focus on the location where the violation condition occurs. An emphasis on earlier program states may lead to a significant savings in case the fault injection occurs much earlier in execution than the violation test itself. A second advantage to pursuing item 1 instead of item 2 is that the data state target may be more precisely defined than the violation condition target. This may provide an easier target to work towards, regardless of the methods employed. For example, the violation condition may test whether a core file is generated or "`/etc/passwd`" is accessed. It intuitively seems much easier to find inputs that lead to the data states that cause these conditions than it is to generate inputs that lead to the violations themselves.

Probing the validity of items 3 and 4 is extremely useful as well. If we can identify that it is impossible to get to a given state or condition through manipulation of the program input space, we can make the assertion that the program *cannot* be exploited in such a way to cause the injected data state or violation to occur. In this case, we can ignore the FIST result in question.

Each of these four problems is extremely difficult to solve. In some cases it will be impossible to find the solution to any of them. Our approach to this undecidable problem is to provide a number of different tools to assist the user in understanding and coming closer to solving the security problem.

7 VIsualizing STatic Analysis (VISTA)

Throughout the course of using FIST in experiments, it became clear that another tool could make FIST analysis easier and more powerful. We addressed the identified needs by creating a second tool (unanticipated in the original proposal) to visualize and navigate source code—the VIsualizing STatic Analysis Tool. VISTA's primary purpose is to provide a way of viewing and navigating static analysis properties of a program as culled by the static data flow tool (described below in Section 7.1). VISTA is useful as a standalone tool, but its power for the current project lies in its integrated use with FIST.

7.1 Static Data Flow

The Static Data Flow (SDF) Tool computes and provides access to static data flow information about a target program [20]. The SDF Tool is made up of a two components, the SDF library, and the `dataflow` pre-processor.

The main component is the SDF library and API. The library provides access to static data flow information for a program through a public API which other programs can use. The purpose of this component is to provide a generic interface to static information about a program, so that other programs can easily access it. The library is responsible for maintaining Call Graph, Control Flow Graph, Data Dependence Graph, and Control Dependence Graph information derived from a given source code target. Users of the SDF library query this information through API calls. VISTA uses this API to navigate a target's SDF data.

The other component of the SDF Tool is the `dataflow` utility. This utility parses C programs and creates an SDF datafile that can be read in by the SDF library. This utility is used to automatically generate SDF information. It integrates easily within a build process, in a similar fashion to FIST's `securetool` pre-processor described in Section 4.1.2.

The SDF Tool was created to assist with AVA/FIST security analysis. In particular it helps with the determining whether inputs exist that actually execute an artificial vulnerability.

FIST analysis determines locations in code that are executed artificial vulnerabilities. To determine whether or not an executed artificial vulnerability is a real vulnerability, one must search for an input that exploits the vulnerability.

There are three ways that the search for an input can be approached: by randomly generating inputs, through human intervention, and with intelligent automatic input generation. Randomly generated inputs are unlikely to create an exploit since the search space is immense; and human intervention is extremely time consuming and error prone. For this reason we targeted intelligent automatic input generation as a task for Year Three of our project (see Section 9). In order to have information about what inputs might exploit a vulnerability, a `pathfinder` needs to know how input relates to the particular vulnerability. This includes information about how data is passed through the program and how control flows through the program. The SDF tool's goal is to provide this information.

7.2 VISTA views

VISTA implements several different powerful ways of viewing static analysis data [19]. For inter-functional data, VISTA provides a call graph (which shows how program functions are connected to each other), and a Caller/Callee List. The latter can be seen in Figure 5. This list can be used

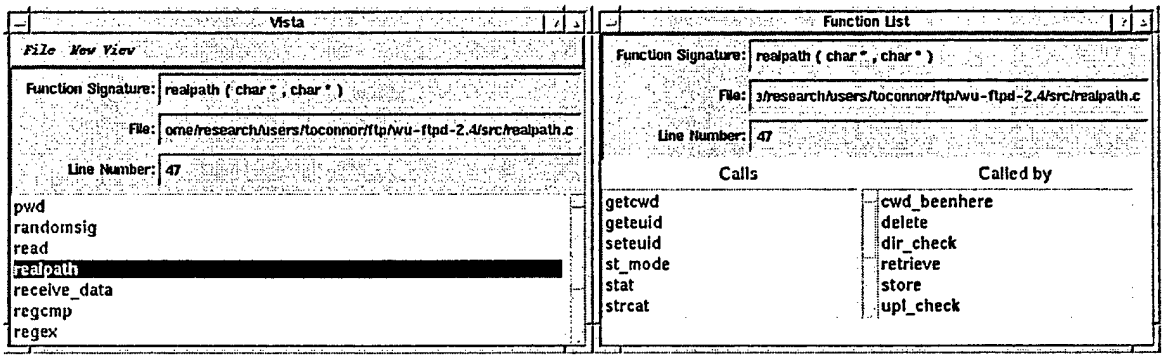


Figure 5: VISTA Main Frame and Caller/Callee Frame

to navigate up and down a program's call tree. For intra-functional data, VISTA provides Control Flow, Control Dependence, and Data Dependence Graphs. These graphs can be viewed alone, or super-imposed on top of each other. To assist in tying this information back to the original source code, functional information (signature, location in file, line number) is provided for each function. A Source Code View displays the source code scrolled to highlight the selected function. All of these views are integrated so that selecting information in one view causes the other views to be automatically updated.

As a standalone tool VISTA can be used to quickly and easily locate particular locations in a program and to traverse between locations. This is particularly useful when investigating FIST vulnerabilities such as buffer overflows [16].

VISTA's graphical display abilities also help an analyst understand how a program is structured with reference to the information contained in the SDF libraries.

8 Integrating FIST and VISTA

Together, FIST and VISTA provide both static and dynamic information about a program under analysis. The way the two toolsets work together can be best understood with an example. In this section, we discuss the application of VISTA and FIST to carry out a stack smashing analysis.

One of the most pervasive computer security problems involves what is known as stack smashing [9, 17]. FIST was built with a stack smashing perturbation function that simulates a stack smashing attack on whatever program is under analysis. As we have discussed, when FIST discovers an exercised artificial vulnerability, there is no guarantee that a real vulnerability exists. That is, FIST can simulate a security problem with fault injection, but can the problem ever *really* happen?

The process of examining exercised artificial vulnerabilities can be performed by hand. This requires the analyst to scan through the source code manually, following execution and data flow. With larger programs spread across multiple source files, libraries, and directories, more time is spent locating source code than actually tracing through the program. VISTA gathers all the static information about a program and provides a user-interface that provides easy navigation through program structure.

In addition to this post facto use of VISTA, some vulnerability analysis can be performed prior to running FIST. Some source code based flaws can be identified statically. This process is tedious

and time consuming. VISTA can assist an analyst by identifying places in the source code to instrument for dynamic FIST analysis.

When performing stack smashing analysis, VISTA can be usefully applied both before and after running FIST [16]. Using VISTA before running FIST helps to reduce the number of instrumentation points required for a thorough FIST analysis, making analysis more efficient. When FIST uncovers one or more executed artificial vulnerabilities, VISTA can again be used to assist in determining if any of these vulnerabilities are real.

8.1 Using VISTA before FIST

A utility program called `stackbuf` processes source code and reports the names of variables that are character buffers allocated on the stack (*i.e.*, have local function scope). This includes user defined data types that are aliases for character buffers, and aggregate user defined data types that contain a character buffer. Instances of all these types have the potential to be used as a target for a buffer overflow attack. For each source file processed, `stackbuf` generates a listing of the variables it has identified as buffers allocated on the stack. The information recorded includes the name of the function, the line number on which the function implementation begins, the buffer data type (always char), and the name of the buffer variable.

Armed with the data from `stackbuf` processing, the analyst can narrow the search for problem code. By concentrating on variables flagged by `stackbuf` an analyst can zero in on the data that absolutely must not overflow. In order to be certain no potential stack buffer overflow conditions could arise during execution, the usage of each flagged variable must be examined in the source code. Dangerous usage consists of using the variable as the target of a dangerous library function (such as `strcpy`), or writing into the contents of the buffer with a loop that may have a flawed invariant. The analyst can use VISTA to determine relevant locations for instrumentation and further dynamic analysis.

8.1.1 Searching for Buffers

One method for using VISTA is to take all the functions listed by all the `stackbuf` generated files and search for them in the source. The main VISTA frame includes a listing of all the functions in alphabetical order. Opening a Source Code View will cause the source to hop from source file to source file as different functions are selected. This saves the analyst time while locating all the flagged variables in the source. The analyst must scan the source code for all uses of the flagged variable in each method. If at any point the flagged variable is used as an argument to another function call, the analyst can quickly navigate to the implementation of the function to which the buffer was passed. Examination of the buffer's usage can continue in the called function, as it should, without time being spent in locating the implementation of the function, since VISTA already knows how to find it.

Another method for quickly identifying the uses of a variable is to take advantage of VISTA's graphing capability. VISTA can generate data dependence graphs. These graphs indicate how data flow through the statements in a function. VISTA uses double green edges to show data dependency. The first step is to scann the graph to identify nodes that contain a reference to the stack buffer in question. Any node referencing the stack buffer that has an outgoing edge, is modifying the stack buffer at that node. Incoming edges to nodes referencing stack buffers get value from some other variable in the function.

Statements that modify stack buffers are good candidates for the FIST buffer overflow instrumentation function. Usually just looking at the variables is not enough to determine if a location modifying a stack buffer is a good candidate for instrumentation. Certain functions that operate on stack buffers signal good instrumentation locations. Searching for these functions once stack buffers have been identified is the topic of the next section.

8.1.2 Searching for potentially-vulnerable functions

Programs that use certain library functions are more likely to be susceptible to buffer overflow and stack smashing attacks than programs that do not. Library functions notorious for being leveraged in programs vulnerable to stack smashing are mostly found in the C string library. Functions such as `strcpy`, `strcat`, `scanf`, and `sprintf` do not check their destination parameter to see if it can hold the amount of data contained in the source parameters. When these functions have stack buffers as destination arguments, the potential for stack smashing exists. Already having a list of stack buffers from `stackbuf` makes identifying which uses of dangerous library functions operate on stack buffers easier.

One way to determine if programs are vulnerable to buffer overflows through the use of "bad" library functions is to scan the source code of the program with a utility like `grep` for the names of the library functions. This will find every place in the source code that these functions are used. However, this will not say anything about how the functions are used, just that they are used. If your coding policy states that these functions should not be used at all, then this would be a sufficient test for the use of certain functions; but the functions themselves do not cause the buffer overflow. The use (or misuse) of the function in the code will determine susceptibility to a buffer overflow attack, which cannot be inferred by looking at the output from `grep`.

Using the static data flow analysis utility like VISTA, an analyst can search intelligently for functions known to contribute to buffer overflow vulnerabilities. Once these locations have been identified, FIST can be used to inject real buffer overflow attack data into the program at that point. Using dataflow and VISTA reduces the number of locations in the program source code that must be instrumented by FIST to perform buffer overflow analysis, saving execution time and analyst effort.

Each statement identified as potentially hazardous, in that it modifies a stack buffer with a known problem library function, should be instrumented for analysis with FIST. The results of a FIST analysis will determine whether a buffer overflow condition exists in the program.

8.2 Using VISTA after FIST

If FIST finds that a location instrumented with a buffer overflow perturbation function violates security, the data that flow to that location need to be examined. As we made clear elsewhere, just because FIST was able to overflow the target buffer does not mean that an exploit exists. Fault injection says nothing about the existence of an exploit. It only says that the program is susceptible to a buffer overflow attack at the point of instrumentation. The analyst must examine the source code and data flow of the program to determine if a real exploit is possible. Each FIST-identified location that violates security must be examined.

FIST and VISTA are not able to determine automatically if an exploit exists for a given vulnerability. This is a difficult problem. FIST and VISTA in concert can assist an analyst in determining the feasibility of an exploit, or the existence of functionality that mitigates the risk of an exploit.

The first step is to examine the source code around the vulnerable location for any steps that might have been taken prior to the vulnerable location that would mitigate the risk of an attack. Steps would include doing any checks on the length of the source variable before it is copied into the destination variable, or scanning the source variable for specific characters and removing them. If these checks do not exist in the source of the current function, then data that enter the function through parameters could cause the buffer overflow. This can be determined by examining the Data Dependence Graph. An analyst can check to see if an edge exists from the entry node to the vulnerable statement node. If so, then all the functions that invoke the current function containing the vulnerable location need to be examined. This can be performed efficiently with the Caller/Callee list. If mitigating code does exist, searching the current path can be aborted and searching a different path can begin.

With the function containing the vulnerable location selected, the Called By list will contain all functions that use the current function. Each of these functions must be checked for their usage of the vulnerable function. The same methods and heuristics used to check data flow and control flow in the function containing the vulnerable location can be used here as well. An analyst can click on each function in the Called By list in turn to determine if the original function (the one containing the vulnerability) is being used improperly. The Source Code View navigates to the implementation of the selected function. Instead of focusing on the vulnerable location, an analyst can focus on the function that contains the vulnerable location. Originally, analysis focused on the exact location of the violation and the function that contains the vulnerability. If the vulnerability is not protected against in the local function, functions that use the function containing the vulnerability must be examined. When examining functions that use the function containing the vulnerability, it is possible to treat the call to the function containing the vulnerability as the vulnerable location itself. This helps when scanning the source code for functionality that could prevent an overflow. This process is repeated as the analyst navigates into different functions in the Called By list, eventually reaching the source of the data, either a constant (safe) or user input (unsafe).

Determining a real exploit is difficult, and VISTA does not have enough functionality to provide the analyst the solution automatically (that is, if it even exists). Using the graphs provided, the analyst can build up a list of criteria that exploit data must have in order to reach the vulnerable location on a given path (call chain). Through a process of trial and error, an analyst can then attack the application with exploit code to determine whether the proposed exploit will crack the software. Examining the source of functions becomes time consuming, however VISTA manages the navigation through source code by always navigating to the function of interest with a single mouse click. The analyst is free to concentrate on what's important, determining the existence of exploits, rather than scanning source code for the implementation of functions.

9 Pathfinder

As a final experiment in this project, we modified the FIST prototype to use call chain information from the SDF tool to attempt to solve the "back to the inputs" problem. This section reports the results of our preliminary experimentation. It is clear that a much larger effort is required to address this open research issue.

When applying dynamic fault-injection, an analyst who discovers unacceptable program behavior must determine whether this behavior can occur during non-fault-injection executions. The question boils down to whether program inputs can be used to drive similar behavior to that ob-

served during analysis with fault injection. Among the first criteria that must be satisfied during the search for such inputs are whether both the program location where data were perturbed and the location where the violation was encountered are reachable. The fact that they are clearly reachable during a run in which faults are injected is neither here nor there. Inputs through normal program interfaces may not be able to cause similar behavior. This can be illustrated by the simple case in which a `boolean` value is reversed by fault injection directly before a branch statement conditional upon its value, and a subsequent violation is detected in the otherwise-untaken branch.

An analyst can search for inputs by hand, but this search is both tedious and inefficient. We are interested in automating the task as much as possible. To this end, a software framework has been developed with the purpose of automating the task of finding input sets that exercise the different paths between a program's entry point and some program location of interest l. An obvious application of this technique is the generation of inputs that, via different courses of execution, will execute some location deemed suspect through a fault-injection analysis. Use of the Pathfinder framework can aid an analyst in discovering test cases that follow all possible routes to the location.

9.1 Identifying Paths

A "Path" **P** from location l_0 to l_1 that the Pathfinder searches for is defined as a sequence of contiguous control-flow edges [20] connecting location l_0 to location l_1 of a program. Typically, l_0 is the entry point of a program or subprogram. A single path represents a possible course of execution between the two locations.

To produce test cases, the Pathfinder begins with data from the Static Dataflow Tool (see Section 7.1) and instruments the source code of the program under analysis so that the execution traces with respect to the control-flow edges of interest can be easily identified. Secondly the possible paths that exist from the program's entry point to the location in question must be identified and stored for later reference. Finally, for each of these identified paths, program inputs need to be found that cause the program to execute accordingly. This search-and-destroy step is assisted by use of the Genetic Algorithm Tool [10]. Upon the discovery of any input satisfying the desired path, Pathfinder creates data in the FIST input file format so that they may easily be used in a subsequent fault-injection analysis. The algorithm for enumerating possible paths to a location l is developed in [2].

Once the set of possible paths has been identified, the remaining task is to find (for each of these paths) a set of inputs that cause the program to exercise that path of execution. RST's Genetic Algorithm Tool has been applied to aid in the solution of this problem. For each path, the GA is called upon to randomly create a population of inputs based on some user-supplied parameters. The program under analysis is run using each generated input and a fitness score is calculated based on comparing the program's execution under test inputs to the actual desired path.

9.2 Experimentation

We performed three different experiments to get a handle on the effectiveness of this approach for finding program inputs. Each experiment was run with three different solvers: a normal genetic algorithm, a differential GA, and a random input generator. For more on the methodology employed in this preliminary experiment, see [10, 11].

The first experiment was performed on a simple program that accepts ten integer inputs. Each input controls the outcome of a decision later in the program, and, consequently controls whether

or not particular program locations will be reached. For each decision in the program, the TRUE and FALSE branches were equally likely to be taken. The Random solver and the Differential GA found 100% of the paths and the Normal GA found 95% of the paths.

The second experiment was very similar to the first experiment, with the difference being that the program was modified to have a deeper call graph. As a result, both the number of paths and individual path length were greater than in the first experiment.

Once again, each decision was equally likely to be TRUE as it was to be FALSE. However, since the number of paths was increased, each individual path was less likely to occur on a given run. In this case, the Differential GA solver found 86% of the paths, whereas the Normal GA and the Randoms solvers found 42% and 40% of the paths respectively.

The third experiment was similar to the second, with the difference being that the decisions were no longer equally likely. Instead of each choice being equally likely, some of the branches were nine times as likely to be taken than their respective counterparts. This causes some of the paths to be much rarer than others. This better simulates a less contrived, "real" program. The Differential GA found 100% of the paths, however the Random solver outperformed the Normal GA 86% to 65%.

These preliminary results demonstrate that using machine-learning techniques is more effective than simply employing random generation[2]. In addition these experiments suggest that differential GA can at times perform much better than the normal GA. This may just be an indication of poorly chosen parameters.

The fitness evaluation scheme utilized in these experiments was overly simple. It is likely that a more elaborate, more intelligent approach to determining test case fitness will force convergence on solutions more effectively.

The pathfinder experiments are a simple first step in the back to the inputs problem. They demonstrate the feasibility of our approach. Even a partial solution to this problem will significantly enhance the potency of fault injection for security.

10 Conclusion

We have presented an overview of the results of Reliable Software Technologies' three-year DARPA-sponsored effort investigating dynamic software security analysis. Two complete toolkits, FIST and VISTA, were presented in a framework explaining their utility in whitebox security analysis. Together the two tools provide static and dynamic analysis capabilities that can identify security vulnerabilities early in the software lifecycle.

Using the methodology presented in this report (and elsewhere, see Appendix A), an analyst can use FIST and VISTA to perform a thorough security analysis on a piece of security-critical software written in C or C++. Ours is a novel approach to the security conundrum that emphasizes early security analysis integrated with system development.

Software fault injection has proven useful in many software domains [23]. This project has examined the utility of software fault injection for security analysis. Overall results are somewhat mixed. Though our system can clearly identify vulnerabilities, it is unclear whether the vulnerabilities identified are real. That is, we can find problems by injecting faults, but the question of whether these problems will occur in the wild remains open. We hope to work on this critical issue in future research.

It is worth mentioning that even if artificial vulnerabilities found with FIST do not turn out to be real, they still have two important features. Firstly they can be used to fortify the target program against potential attacks, and secondly they can be used to aid in the design of Information Warfare attacks against the code. In this sense, FIST is a powerful defensive and offensive IW weapon.

FIST and VISTA are powerful tools, and they can help an analysts perform security analysis on source code before its release. Many of the tedious aspects of source-code–based security analysis have been captured and automated in FIST and VISTA.

References

[1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. In *The USENIX Association, Computing Systems*, pages 131–152, Spring 1996.

[2] B.Sohr and M. Schatz. Pathfinder: Generating test data to fulfill control-flow requirements. Technical Report RSTR-059-1007, Reliable Software Technologies, Sterling, VA, September 1998.

[3] CERT. CA-97.09: Vulnerability in IMAP and POP, April 1997.

[4] CIAC. F-11: Unix NCSA `httpd` vulnerability, February 1995.

[5] CIAC. H-110: Samba servers vulnerability, September 1997.

[6] G. Fink and M. Bishop. Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4), July 1997.

[7] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, 14(4):73–83, July 1997.

[8] S. Garfinkel and G. Spafford. *Practical Unix & Internet Security.* O'Reilly & Associates, Inc., 2nd edition, 1996.

[9] A.K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 104–114, Oakland, CA, May 3-6 1998.

[10] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. Technical Report RSTR-018-97-01, Reliable Software Technologies, Sterling, VA, December 1997.

[11] C. Michael, G. McGraw, M. Schatz, and C. Walton. Genetic algorithms for test data generation. In *Proceedings of the Twelfth IEEE International Automated Software Engineering Conference (ASE 97)*, pages 307–108, Tahoe, NV, November 1997.

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

[13] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[14] J. VOAS AND K. MILLER. Dynamic testability analysis for assessing fault tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.

[15] J. VOAS AND K. MILLER. Predicting software's minimum-time-to-hazard and mean-time-to-hazard for rare input events. In *Proc. of the Int'l. Symp. on Software Reliability Eng.*, pages 229–238, Toulouse, France, October 1995.

[16] T. O'Connor. A methodology for determining buffer overflow and stack smashing vulnerability. Technical Report RSTR-041-1007, Reliable Software Technologies, Sterling, VA, July 1998.

[17] Aleph One. Smashing the stack for fun and profit. Online. Phrack Online. Volume 7, Issue 49, File 14 of 16. Available: `www.phrack.com/Archive/`, November 9 1996.

[18] Reliable Software Technologies, 21515 Ridgetop Circle, Suite 250, Sterling, VA. *WhiteBox FIST User's Manual*, September 1997.

[19] Reliable Software Technologies, 21515 Ridgetop Circle, Suite 250, Sterling, VA. *VISTA User Manual*, August 1998.

[20] M. Schatz. What is a program dependence graph. Technical Report RSTR-034-1007, Reliable Software Technologies, Sterling, VA, December 1997.

[21] E.H. Spafford. The Internet worm program: An analysis. *Computer Communications Review*, 19(1):17–57, January 1989.

[22] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proceedings of the 11th Annual Conference on Computer Assurance*, pages 250–263, June 1996.

[23] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley and Sons, New York, 1998.

[24] B. Werner, editor. *Eleventh Annual Computer Security Applications Conference (ACSAC'95)*, Los Alamitos, CA, December 1995. IEEE Computer Society Press.

[25] H. Zuse. *Software Complexity: Measures and Methods.* DeGruyter, Berlin, 1990.

11 Appendix A: Document repository index

The following is a complete index of all documents related to the FIST/VISTA project. The repository is available in electronic form from contract administrators. The repository is set up as a set of subdirectories containing categories of documents. Each major heading below corresponds to a directory in the repository.

- **experiments** Contains documents written during experimentation.
  - **exp-ideas.txt** Lists a number of known security exploits that FIST might want to be able to find
  - **ftpd** Contains documents about the FTP daemon

- **wuftpd.html** Analysis of the Buffer Overflow in WU-FTPD detected by FIST
- **report.txt** Discussion of the wu-ftpd experiment
- **smash.txt** Smashing The Stack For Fun and Profit (not written by RST)
- **httpd** Contains documents about the HTTP daemon
  * **security-analysis.html** An internal tech report written describing the results of analysis on httpd
  * **httpdcrash.txt** Rough sketch of the analysis process for httpd along with some preliminary results analysis

- **conference papers** Contains conference papers that were written during this project
  - A. Ghosh and T. O'Connor. "Analyzing Programs for Vulnerability to Buffer Overrun Attacks," To appear in *Proceedings of the 21st National Information Systems Security Conference*, October 5-8, 1998, Crystal City, VA.
  - A. Ghosh, T. O'Connor, G. McGraw."An Automated Approach for Identifying Potential Vulnerabilities in Software." *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA. May 3-6, 1998, pp. 104-114.
  - J. Voas, G. McGraw, A. Ghosh, F. Charron, K. Miller. "Defining an Adaptive Software Security Metric from a Dynamic Software Failure-tolerance Measure," *Proc. of the 11th Annual Conference on Computer Assurance (COMPASS'96)*

- **presentations** Contains the slides for presentations that were given during this contract.
  - **afrl** Presentation given to AFRL in June of 1998 (end of Year 3).
  - **darpa** Presentation given to DARPA in 1998.
  - **ieee** Presentation given a IEEE Security and Privacy 1998.
  - **nissc** Presentation given at NISSC in 1996.
  - **Rome** Presentation given at Rome Labs
  - **tahoe** Presentation given at DARPA Component Wrappers Workshop, August 1997.
  - **vista-development** Presentation given internally at RST talking about things learned during the development of VISTA.
  - **year1** Presentation given at RST at the end of year 1 of this contract.
  - **year2** Presentation given at RST at the end of year 2 of this contract.

- **prototypes** Contains documents written during the building of the prototypes that were created during this project
  - **FIST** Contains documents that were generated involving the FIST prototype
    * **demo** An online demo of the FIST tool
    * **development** Contains docs about the development of FIST
      · **ARPA-tasks.ps** A document listing describing tasks that were to be done for FIST

- **assertmon** Contains documents about the AssertMon component that was written for FIST
  1. **1stdraft** A first draft of a requirements doc for AssertMon
  2. **2ndraft** A second draft of a requirements doc for AssertMon
  3. **spec1.ps** Specification document for AssertMon
  4. **design.ps** A preliminary design document for AssertMon
  5. **d2.ps** A design document for AssertMon
  6. **input.ps** A design doc for the Input Handler
  7. **output.ps** A design doc for the Output Handler
  8. **asserthand.ps** A design doc for the Assertion Handler
  9. **mainrout.ps** A design doc for the MainRoutine
- **buffer-overflow** Contains documents about how to test for buffer overflows
  1. **boverflow.html** Proposal on how FIST could support buffer overflow analysis
  2. **rtil_bover.ps** Documentation on the buffer overflow perterbation functions in the RTIL.
- **cbrowse** Contains documents about how the CBrowse component was modified for FIST
  1. **CBrowseTestPlan.txt** Describes how to test CBrowse
  2. **to_be_done** Describes tasks that need to be done for CBrowse to support FIST
- **dev97-tasks.ps** Lists tasks for 1997
- **dev_tasks.txt** Asks questions that led to what was developed
- **execman** Contains documents about how the execman component that was modified for FIST
  1. **execman.ps** A document listing some of the changes required of execman to support security analysis
  2. **exec_proc_ctrl.ps** Describes how Execman will deal with process control for daemons
  3. **ftpd.ps** A document describing what is required to test network daemons
  4. **ftpd.fig** A figure to go along with the above document
  5. **input_points.txt** Ideas on how input can be specified to a program being tested
  6. **redesign.ps** A document describing the redesign of execman
- **grammar** To assist in testing httpd a grammar generator was created. Docs about this are in this directory.
  1. **ideas.txt** Info about the grammar
  2. **user.html** User manual for the grammar generator
- **httpd.ps** A high level internal doc describing changes needed to support analysis of httpd
- **httpdmore.ps** An internal doc listing changes that were required to be made to FIST to support analysis of httpd

      - **rtil** Contains documents about how the Run-Time Instrumentation Library had to be modified for FIST.
        1. **instr_lib_mods.ps** Describes some of the changes that needed to be made to RTIL
        2. **pert_funcs.txt** Describes perterbation functions
      - **schedule.ps** A schedule created for adding socket control to FIST
      - **socketman.ps** An internal document describing work done to handle sockets
      - **TestPlan.txt** A test plan for FIST
      - **DY3-Vision.html** The vision for FIST for DY3
    * **install.txt** Instructions on how to install FIST
    * **technology** Contains docs about some of the technology learned during FIST
      - **HowTTI.txt** Describes how Time To Intrusion is calculated
      - **threat-class.txt** Lists some threat classes
    * **user-manual** Contains the FIST user's manual
- **misc** Contains documents that were generated during development that don't fit under any one prototype
    * **source-merge** Contains documents that describe what was done to merge all of the work that has been on different prototypes into one branch in the source code revision management system.
- **PathFinder** contains documents that were generated during development of the path finder prototype.
    * **requirements** the requirements for the pathfinder.
    * **design** the design for the pathfinder.
- **SDF** Contains documents that were generated involving the SDF sub-project
    * **design** Contains documents describing the design of the SDF prototype
      - **require.html** Contains the requirements that the SDF library is supposed to fulfill
      - **data-structure.html** Describes the design of the data structures that store the SDF info.
      - **analyzer.html** Describes the design of the analyzer that takes a parse tree and converts it into the SDF data structures.
      - **sdf-file-format.txt** Describes the file format that is used to store SDF files
    * **api** Contains API documentation for the SDF library
- **VISTA** Contains documents that were generated involving the VISTA sub-project
    * **api** Contains the API documentation for VISTA
    * **demo** An online demo of the VISTA tool
    * **design** Contains design documents for VISTA
      - **graph** Document describing the design of the graph package used by VISTA
      - **vista** Document describing the design of VISTA

    * **misc** Contains miscellaneous VISTA documents
        - **hacks.html** A document describing different implementation details that might not be obvious at first glance what they do
        - **future.html** A list of features that we would like to see in VISTA if we had the time
    * **user-manual** The VISTA user manual

- **status** Contains the status reports written for this project

  - **nov95.ps** Status for November 1995
  - **dec95.ps** Status for December 1995
  - **jan96.ps** Status for January 1996
  - **feb96.ps** Status for February 1996
  - **mar96.ps** Status for March 1996
  - **apr96.ps** Status for April 1996
  - **may96.ps** Status for May 1996
  - **jun96.ps** Status for June 1996
  - **jul96.ps** Status for July 1996
  - **aug96.ps** Status for August 1996
  - **sep96.ps** Status for September 1996
  - **oct96.ps** Status for October 1996
  - **nov96.ps** Status for November 1996
  - **dec96.ps** Status for December 1996
  - **jan97.ps** Status for January 1997
  - **feb97.ps** Status for February 1997
  - **mar97.ps** Status for March 1997
  - **apr97.ps** Status for April 1997
  - **may97.ps** Status for May 1997
  - **jun97.ps** Status for June 1997
  - **jul97.ps** Status for July 1997
  - **aug97.ps** Status for August 1997
  - **sep97.ps** Status for September 1997
  - **oct97.ps** Status for October 1997
  - **nov97.ps** Status for November 1997
  - **dec97.ps** Status for December 1997
  - **jan98.ps** Status for January 1998
  - **feb98.ps** Status for February 1998

- **mar98.ps** Status for March 1998
- **apr98.ps** Status for April 1998
- **may98.ps** Status for May 1998
- **jun98.ps** Status for June 1998
- **jul98.ps** Status for July 1998

- **tech-reports** Contains technical reports that were written
  - **CDRL_4.1.1.ps** Contract Data Requirements List 4.1.1
  - **CDRL_4.1.2.ps** Contract Data Requirements List 4.1.2
  - **CDRL_4.1.3.ps** Contract Data Requirements List 4.1.3 and 4.1.4
  - **CDRL_4.1.5.ps** Contract Data Requirements List 4.1.5
  - **CDRL_4.1.5.ps** Contract Data Requirements List 4.1.6
  - **report1.ps** Six month technical report
  - **wuftpd.ps** Internal report on WU-FTPD security analysis experiment
  - **Methodology.html** RSTR-041-1007 which discusses how to use the FIST and VISTA prototypes to search for buffer overflow vulnerabilities in source code
  - **Pathfiner.html** RSTR-059-1007 which describes the Pathfinder experimentation that was performed.
  - **what-is-pdg.html** RSTR-034-1007 which describes the concept of program dependence graphs
  - **final.ps** Final Report

DISTRIBUTION LIST

addresses	number of copies
JOHN C. FAUST AFRL/IFGB 525 BROOKS RD ROME NY 13441-4505	10
RELIABLE SOFTWARE TECHNOLGIES CORP 21351 RIDGETOP CIRCLE SUITE 400 DULLES VA 20166	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL 1
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDER, CODE 4TL000D 1
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

CDR, US ARMY AVIATION & MISSILE CMD 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

REPORT LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

ATTN: D'BORAH HART 1
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

ATTN: KAROLA M. YOURISON 1
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

USAF/AIR FORCE RESEARCH LABORATORY 1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.