

# A Comparison of Static Analysis and Fault Injection Techniques for Developing Robust System Services

Pete Broadwell    Emil Ong  
{pbwell, emilong}@cs.berkeley.edu

## Abstract

Static source code analysis and software fault injection are two popular approaches to testing and verifying the robustness of software. We chose a set of commonly-used applications: CUPS, Berkeley DB, the GNU file utilities, Apache, MySQL, sudo and zlib, and tested them with both static analysis and fault injection tools to discover errors. The results of our tests provide insight into the strengths and weaknesses of each technique. These results also suggest possibilities for improving each type of tool, as well as ways to form a synergistic combination of the two.

## 1 Introduction

### 1.1 Motivation

Software failures have been a problem as long as software has existed. As modern computer systems continue to increase in complexity, this problem has grown in size and scope. The advent of giant-scale Internet services and ubiquitous computing creates an environment in which small software errors can have far-reaching consequences for business and the general public. Compounding the situation is the inherent quality of systems to become more vulnerable to software errors as they grow in size. A final concern is that the abbreviated devel-

opment cycle favored by contemporary Internet software companies neglects careful programming practices in favor of earlier release dates.

While the final concern listed above is primarily a cultural one, the others represent pressing, solvable issues for the computing industry and research communities. The dangers of software failure can be ameliorated by the development of tools for software development and testing that help produce more robust and reliable systems.

### 1.2 Background

While no panacea for software failures exists, a number of techniques have been developed over the years to try to combat failures in software. Many of these techniques were initially applied only to specialized, mission-critical software packages. Recently, however, techniques that have traditionally belonged in the realms of fault-tolerant computing or software engineering have been adopted and extended by the greater systems community.

Efforts to reduce the number of software errors began with human-oriented disciplines of code-writing and quality assurance testing. These techniques eventually grew into more automated methods that use software tools to write better software applications.

One such technique is static source code

analysis, or simply static analysis (SA). The original idea behind SA was to mimic human code auditing, but then moved even further when the tools started to find problems that humans would not have found easily. SA has a long tradition dating back to the days of `lint`. Modern SA uses advanced approaches such as model checking, advanced type models and formal verification. Recent research in SA includes the Open Source Quality project at UC Berkeley, the Meta-Compilation effort at Stanford, and the work produced by the Software Productivity Tools team at Microsoft Research.

Another newer technique for improving software quality is software-implemented fault injection (SWIFI). The philosophy behind SWIFI is that there always will be the possibility of failures in software and hardware, and that SWIFI can be applied to discover how well software behaves when failures happen. The hardware world has long used fault injection to test and verify designs, especially due to the cost and difficulty of producing hardware. Until recently, software fault injection was a neglected area of research, perhaps because the higher cost of producing hardware caused more attention to be paid to hardware design. However, as the complexity and number of users of systems has scaled rapidly, so has the severity of the potential consequences of software failures. As a result, SWIFI techniques have recently been the subject of greater interest.

Outside of the traditional fault-tolerant computing community, a number of contemporary projects dedicated to improving the reliability of computer systems have explored the use of SWIFI tools. These include the Recovery-Oriented Computing (ROC) project at UC Berkeley and the Mendosus project at Rutgers.

### 1.3 Goals

In our study, we tried to determine the relative strengths of SA and SWIFI in the field of software quality verification. We believe such a study is novel because of the traditional view of these two techniques; SA is generally considered to be a developer's tool while SWIFI is thought of as a testing tool. Nonetheless, both SA and SWIFI can be used to improve software. Our aim was to determine the precise areas in which the uses of the two overlap and where they are disparate. Using the same set of software packages for both techniques as our evaluation platform, we tried to learn more about the following properties of each type of tool:

1. Operating model/operator overhead
2. Ease of use
3. Infrastructure requirements
4. Types of errors found
5. Precision/accuracy of error detection

From this comparison, we would like to be able to recommend one type of tool or the other for certain software testing scenarios, or to propose a way of somehow combining them.

## 2 Related Work

We believe that our comparison of SA to SWIFI is novel, so we were unable to find any other direct analysis. The Para-Protect study of secure coding practices [19] contains a survey of tools in both areas, but no actual testing was conducted. There is, however, extensive work in both of these fields individually, which we are able to use as a starting point.

## 2.1 Static Analysis

A great number of papers have been written on static analysis techniques: [21, 12, 15, 3, 25, 24, 1, 4] There are a number of tools that have also been developed which implement these techniques. At Berkeley, the BANE project has produced software for finding Y2K bugs, race conditions, and memory errors. Also at Berkeley is a very promising new project called CQual [21, 12] which uses variable typing techniques to check properties of programs. A number of “format-string” errors have been found with CQual. The Meta-Compilation project at Stanford is a program-specific model checker [3] which has gained much publicity from the discovery of numerous bugs in the Linux kernel. The Software Productivity Tools group at Microsoft Research has produced two tools, ESP (Error detection via Scalable Program analysis) [9] for analyzing large programs and SLAM (Software (Specifications), Languages, Analysis, and Model checking) [4] for highly accurate, smaller scale analysis. The tradition of lint has been continued in Splint [17] (formerly known as LCLint) which uses programmer annotations to improve accuracy. Syntactic tools that make very fast reports based on vulnerability databases are also popular. Examples include ITS4 (It’s The Software, Stupid, Security Scanner) [24] and RATS (Rough Auditing Tool for Security).

## 2.2 Fault Injection

Three general categories of fault injection tools are hardware-implemented fault injection, simulator-based fault injection and software-implemented fault injection (SWIFI) [2]. For this study, we focused on SWIFI techniques. It is worth noting, however, that both hardware-implemented and simulator-based fault injection have long

been used for testing, particularly of hard real-time and mission-critical systems.

SWIFI has risen to prominence more recently, but even so, a perusal of the “related work” sections of most papers on SWIFI techniques for fault tolerance yields a virtual alphabet soup of project names and acronyms, such as FIAT, FERRARI, FINE, FTAPE, DOCTOR, MAFALDA, HYBRID, ORCHESTRA and GOOFI. [7]. These SWIFI implementations fall into a number of different categories:

1. Low-level processor and memory failures. This is perhaps the oldest SWIFI approach, leveraging the capabilities of the operating system to simulate transient bit-flips or permanent component failures. The goal of this type of fault injection usually is to determine the resiliency of the processing components of mission-critical and real-time systems. Examples: Xception [7], HYBRID [27].
2. Corruption of a process’s memory image. This approach often is applied to commodity operating systems like UNIX. Tools such as the `ptrace()` utility or `/proc` file system are used to test the failure-related behavior of individual applications. Examples: FERRARI [13], Janus [14].
3. SAN-based fault injection. These methods represent the application of fault injection to distributed systems. Most focus on the injection of network/communications errors, but also include large-scale events such as unexpected node shutdowns or reboots. Examples: ORCHESTRA [10], FTAPE [23].
4. Interface fault injection. While the approaches above limit their fault models to causes stemming from low-level

hardware faults, a handful of projects have sought to explore how systems and applications handle errors introduced by faulty inputs from users or other applications. Examples: Ballista [16] Fuzz [18].

5. System Perturbations. Recent efforts have emphasized a more holistic view of fault injection. Whittaker [26] advocates integrated testing of interfaces other than the application’s external user interface. Examples of such interfaces include interactions between an application and the operating system, or between an application and other function libraries. Such a setup allows for the injection of failures that may better be described as “perturbations” of the operating environment. These include problems caused by hardware failures, misbehaving user processes, human operators and system load spikes. Examples: Holodeck [11], FIG [6].

### 3 Experimental Setup

We tested a variety of SWIFI and SA tools against a common set of UNIX services. In order to make the study applicable to SA, we only chose software packages for which source code was available. SA has no functionality without source. For the SA portion of the testing, it was sufficient to run the analysis tools on the entire package. For SWIFI, however, it was necessary to run each executable program in the package through a suitable set of operations with the fault injection tool enabled. The packages tested, as well as the individual applications run, are given below:

1. *UNIX file utilities* — `ls`, `cp`, `mv`, `rm`, `chgrp`, `chmod`, `chown`, `dd`, `df`, `du`, `install`, `ln`, `ls`, `mkdir`,

`mkfifo`, `mknod`, `rmdir`, `shred`, `sync` and `touch`. Tests involved running each of these utilities in a shell in a manner consistent with its normal usage pattern.

2. *Apache 1.3.22* — an open-source HTTP server. For SWIFI testing, we started up the `httpd` service on a Linux machine and then used a standard web browser to access a set of static HTML pages through this server.
3. *Berkeley DB 4.0.14* — an open-source flat file database library. Our test application was a port from TCL to C of one of the library’s regression tests. This test loads an unsorted list of 10,000 words into the database and then reads them out in sorted order. sites.
4. *MySQL Server 3.23.36* — an open-source database server with full transactional properties and support for remote access. We used the accompanying MySQL client program to access, query and alter existing database tables that were stored on the server.
5. *sudo* — the standard UNIX service to allow non-root users to run specified applications with root privilege. SWIFI tests involved using `sudo` to access and manipulate files in the root user’s home directory.
6. *zlib 1.1.4* — file compression utilities. Our testing approach was to run the example program provided with the application. This program opens, compresses and uncompresses a sample file, using the `gzip` and `gunzip` utilities.
7. *CUPS 1.1.14* — the Common UNIX Printing Service. To test this package, we ran the line printer daemon (`lpd`) under the fault injection tools while also

using the printing tool (`lpr`) to print a PostScript file.

We focused on three areas of software quality: (1) handling of errors from system calls and standard library calls; (2) the presence of file system race conditions; and (3) potential memory errors, particularly buffer overflows.

### 3.1 Static Analysis Test Setup

Three SA tools were used to investigate each area of software quality. For handling of errors from system calls and standard library calls, we implemented a simple model checker called Stumoch (**Stupid model checker**). ITS4 was used to detect potential file system race conditions. To check for buffer overflows, a tool [25] created by David Wagner based on the BANE toolkit, `warnbuf`<sup>1</sup>, was used.

Using ITS4 involved simply running the tool on all the C and C++ files in each project without preprocessing. The other tools required that the inputs be preprocessed and only handle C code.

Stumoch is a simplistic model checker written by the authors for this study. Given preprocessed C code, it compares each function call in the code with a user-supplied database. The database contains functions whose return values must be checked and, optionally, a set of `errno` values that must also be checked. The simple model used by Stumoch is that all return values and `errno`s must be checked before the end of the enclosing function or the next function call. A value is “checked” when it appears in a `switch` statement or as a conditional in an `if` statement. This approach led to a small

---

<sup>1</sup>The tool technically has no name, but `warnbuf` is the name of the executable for the tool. We use the title for convenience.

number of false positives, but is quite safe against false negatives.

### 3.2 Fault Injection Test Setup

We also used three tools as part of the fault injection tests. To test how applications handle errors from library calls, we used the FIG (**F**ault **I**njection in **g**libc) tool that was developed by the UC Berkeley ROC project [20]. This tool uses the well-known UNIX/Linux `LD_PRELOAD` environment variable mechanism to interpose instrumented library routines between the application and the usual library routines. These instrumented routines can then perform system call logging and fault injection, based on directions from a control file.

To simulate file system race conditions, we utilized Subterfuge [8], an open-source framework for altering a process’s state. Subterfuge uses the `ptrace()` utility in UNIX-like OSes to access and alter a process’s memory image. Using Subterfuge, it is possible to register actions (referred to as “tricks”) that may be taken before and after system calls are made and/or signal handlers are invoked. For this test, we wrote a Python script called “MoveFileTrick” that moves files to a temporary directory immediately after they have been checked by the `stat()` or `access()` permissions/existence checking system calls<sup>2</sup>. If the program being run under Subterfuge attempts to use these files later, it may manifest some type of unexpected behavior. By observing this behavior, it is possible to determine if a security weakness exists in the program.

Finally, to investigate the test applications’ resistance to buffer overflow attacks we used a tool that we called “Fuzz Lite”. This uses the same approach as Miller’s Fuzz

---

<sup>2</sup>This trick incorporates write-ahead logging and a separate recovery script to make sure no critical system files get lost in the event of a crash

tool [18], feeding input strings of random characters to an application in an attempt to cause a buffer overflow, which would then crash the application.

## 4 Experimental Results

We now present the results we observed when we applied our static analysis and fault injection tools to the test applications. It is worth noting that for the purposes of this paper, what the test results say about the target applications is not as important as what the test results reveal about the test tools themselves.

### 4.1 Static Analysis

Table 1 lists a summary of the test results of running the static analysis tools on the applications. We discuss specific results below.

#### 4.1.1 Memory Errors

The reported memory errors in Table 1 only apply to errors found with `warnbuf`. In section 4.1.3 we will see that other memory errors existed and were found by `Stumoch`. Only `CUPS`, `Apache`, and `MySQL` contained potential memory errors. All three contained a particular error that relates to the size of integers on the host machine.

Consider the following C code:

```
void foo(int x)
{
    char s[12];
    ...
    sprintf(s, "%d", x);
    ...
}
```

On 32-bit (or smaller) architectures, this code will never cause a buffer overflow be-

cause the maximum length of the decimal string representation of an integer is the length of the decimal string representation of `INT_MIN` (-2147483648), which is 11 characters. However, on a 64-bit machine, the length of `INT_MIN` (-9223372036854775808) may be 20 characters<sup>3</sup>. This subtle error could lead to potential security problems as well as difficult to diagnose crashes.

In the `readline` module of `MySQL`, we also found unchecked buffer errors in some string formatting code where `vsprintf()` is used with a bounded buffer allocated in the program's data section. The formatting is not limited to integers as with the previous (64-bit) problem.

#### 4.1.2 Race Conditions

We used `ITS4` to detect file system race conditions in the packages. File system race conditions generally come in the form of "Time-Of-Check-To-Time-Of-Use" (TOCTTOU) errors [5]. These problems involve using only the name of a file to do access checking. For example, a program with elevated privilege might check if a user may access a certain file. Based on the check, the program might allow the user to manipulate the file. Because file names are only pointers to files, a malicious user could carefully time an attack by asking an elevated privilege program to access a file and after the permissions check had been made, but before the actual access is done, relink the file name to another file which he or she may not usually access. The attacker now has unauthorized access.

---

<sup>3</sup>In many cases, this code is safe even on 64-bit machines because many vendors define the minimum size of 'signed int' to be the 32-bit minimum and the minimum size of 'signed long int' to be the 64-bit minimum. However, the C standard allows 'signed int' values to be 64-bit.

Package	Memory Errors	Race Conditions	Error Checking	Lines of C code
CUPS	64-bit unclean	No errors found	216 unchecked return values or errnos	787081
Berkeley DB	No errors found	No errors found	6 unchecked return values or errnos	432305
GNU Fileutils	No errors found	1 possible RCs found	36 unchecked return values or errnos	116673
Apache	64-bit unclean	No errors found	25 unchecked return values or errnos	461065
MySQL	64-bit unclean Possible overflows	No errors found	52 unchecked return values or errnos	1206621
sudo	No errors found	2 possible RCs found	30 unchecked return values or errnos	40149
zlib	No errors found	No errors found	10 unchecked return values or errnos	19431

Table 1: Static Analysis Test Results

These errors often manifest themselves in Unix with the use of the `access()` and `stat()` calls. ITS4 warns specifically about TOCTTOU errors with these calls and, based on its output, we found one potential race condition in GNU Fileutils and two potential race conditions in sudo. The race condition in GNU Fileutils is less worrisome because the programs in this package are rarely installed to run with elevated privilege. The race conditions in sudo however, are cause for concern because sudo *must* be installed to run as the superuser<sup>4</sup>.

### 4.1.3 Error Checking

The results that we obtained from Stumoch were encouraging from the standpoint of static analysis, but disappointing from the standpoint of those wanting reliable software. We created a small list of functions, `open()`, `fopen()`, `read()`, `write()`,

<sup>4</sup>Both of the race conditions found in sudo are probably of low risk because of their particular location in the program. They both involved use of a variant of the `stat()` function call.

`select()`, `malloc()`, `close()`, and `send()`, and an associated list of possible error numbers for each function. We used this list of functions to evaluate each of the software packages.

In almost all of the packages, we found unchecked return values and in some packages, error numbers (errno) were largely ignored. Berkeley DB is a shining exception however, with no unchecked return values. There were some “unchecked” errno, but even in these cases an errno-specific message was printed.

In the other packages, however, some disturbing results appeared. The return value of `close()` is largely ignored. Error number checking is rarely consistent if even present. The most amazing result that we found was that many programs use buffers returned from `malloc()` without checking if they are null!

## 5 Fault Injection

Table 2 lists a summary of the test results of running the fault injection tools on the applications. We discuss specific results below. Note that the output from running fault injection tests is considerably different from that provided by static analysis tools. SA points out discrete errors, while SWIFI only triggers certain behavior on the part of the program that it is then up to the tester to interpret.

Note also that we were unable to obtain any meaningful test results from the `sudo` utility. This is because the formidable security measures employed to safeguard this tool from inappropriate use also disallow most types of fault injection. For instance, `sudo` re-writes the environment variable array `envp[]` upon invocation, removing any variables beginning with the letters ‘LD’. Obviously, this renders `FIG` useless. In addition, it checks that it is actually being run `setuid root`. Since `Subterfuge` requires that the test program be run within its framework of Python scripts (not `setuid root`), this check fails and `sudo` terminates.

### 5.0.4 Memory Errors

The “Fuzz Lite” tool ultimately was unable to expose any input buffer management vulnerabilities in the applications targeted for testing. Obviously the state of input management behavior on the part of UNIX applications has improved since Miller’s historic study. However, it is possible that buffer overflow vulnerabilities still exist in some of the test applications. Input buffer vulnerabilities are some of the most insidious security weaknesses, and tend to crop up in the places where developers and testers least expect them to appear, as the discovery in 2000 of a buffer overflow vulnerability in the handling of email message headers by

Microsoft Outlook [22] demonstrated. Our “Fuzz Lite” tool was only able to check the user interfaces of the test programs, and it is possible that a more sophisticated tool could have uncovered vulnerabilities in other interfaces.

### 5.0.5 Race Conditions

Detection of race condition errors (or any other type of security vulnerability) by fault injection is a difficult proposition. To be effective, the tool must actually attempt to compromise the security of the application, or at least engage in behavior that will make the presence of a security vulnerability visible to the tester. As a result, such tests tend to be targeted toward a particular application or set of applications, in contrast to the model-based approach employed by SA.

Our test for file system-related race conditions attempts to expose potential security holes that arise from the existence of “Time-Of-Check-To-Time-Of-Use” (TOCT-TOU) errors. By moving a file between the time its filename is checked by `stat()` or `access()` and the time it is read from or written to directly, we hoped at least to expose the *potential* for file system race conditions of the type described above to exist in the target applications.

Although several of our test programs exited with an error when run under the `MoveFileTrick` (an understandable reaction, given the circumstances), we only witnessed suspect behavior on the part of one test program, the file utility `cp`. When attempting to copy from a file that suddenly no longer existed, `cp` returned the error “source and destination file are the same.” Obviously this error does not accurately reflect the situation, but analysis of the source code was subsequently able to determine that no file system race condition actually existed in this case.

Package	Memory Errors	Race Conditions	Error Checking
CUPS	N/A	N/A	Crashes
Berkeley DB	None	Returns error	Ok
GNU Fileutils	None	cp: wrong err msg	Ok
Apache	None	Returns error	Ok
MySQL	None	No effect	Ok
sudo	N/A	N/A	N/A
zlib	None	No effect	Crashes

Table 2: Fault Injection Test Results

### 5.0.6 Error Checking

The FIG project [6] had a great deal of success with analyzing an application’s behavior when it is faced with failed system calls. Our tests for this project continued that work, and by and large the results of our tests corroborate the conclusions given in the FIG paper. Once again, we found that an alarming number of applications do not check return values from functions such as `malloc()`, and thus crash when they attempt to make use of the erroneous value returned from the failed call. Of the applications we tested, the `zlib` compression utilities and the `CUPS` printing services were guilty of this behavior.

Other applications exited due to erroneous return values from `malloc()`, but at least provided an error message explaining that the program could not continue running due to memory exhaustion. The UNIX file utilities actually retried the failed call to `malloc()` a bounded number of times before giving up.

The assertion from the FIG paper that server-side applications generally are engineered with greater recoverability was thrown into doubt by the negative results from `lpd`. Yet server-side applications responsible for large-scale, networked activity fared better: both `Apache` and `MySQL` aggressively retried failed calls, uti-

lized “worker pools” of child process for greater aggregate availability and used resource pre-allocation (reserving all file handles and memory space at startup) in order to provide a more reliable service model.

Most of the applications were at least somewhat intelligent in their handling of other failed system calls. Most programs would give up an I/O operation immediately upon receiving an `EIO` (I/O error) error number, but would retry calls that failed with an `EINTR` (system call was interrupted by a signal), as well as network `recv()` calls. Network `send()` calls usually were not retried automatically, however, given that aggressive retries might have the effect of compounding a communication channel bottleneck.

Table 3 provides a comprehensive list of the results of the error checking fault injection tests.

## 6 Discussion & Analysis

### 6.1 General Comments on Static Analysis Tools

In our testing using static analysis tools, the difficulty of using these tools became apparent. Most of the difficulty can probably be attributed to the fact that a large number of the tools are simply “proof of concept” im-

	read		write		select()	malloc()
	EINTR	EIO	ENOSPC	EIO	ENOMEM	ENOMEM
lpd	<b>exit</b>	<b>exit</b>	<b>exit</b>	<b>exit</b>	<b>exit</b>	<b>crash</b>
Berkeley DB	retry	detected	Xact abort	Xact abort	n/a	Xact abort
File Utils	retry	warn	warn	warn	n/a	retry/warn
Apache	o.k.	req dropped	req dropped	req dropped	o.k	n/a
MySQL	Xact abort	retry, warn	Xact abort	Xact abort	retry	restart
sudo	n/a	n/a	n/a	n/a	n/a	n/a
zlib	warn	warn	warn	warn	n/a	<b>crash</b>

Table 3: Results of testing lpd (CUPS), Berkeley DB, GNU File Utilities, Apache, MySQL, sudo and zlib (example program) under various system call failures with FIG. The programs and their configurations are listed at left. Along the top, we list the different library calls we tested and the particular error being injected. Boldfaced items indicate poor or undocumented behavior.

plementations for certain techniques. The documentation is poor, if available at all. There is still a great amount of user interaction necessary in running these tools because of two main factors: the parsers and false positives.

Parsing C code seems to be one of the most difficult tasks for writers of these tools, as we found when writing Stumoch. Modern C code is awash in preprocessor statements and compiler specific extensions. The extensions used by GCC and Microsoft Visual C++ pose a serious parsing issue for many of these tools. Some code that we encountered was in compliance with the newest C standard, C99, but not with the older, more common standard, C89. The transition to the newer standard has caused many problems. These parsing problems require the user to “babysit” the tools, performing code modification or even manual evaluation when a parsing error is encountered. This time represented the majority of the effort spent on actually running these tools. When the tools are actually successful in parsing however, most are quite fast and take time on the order of code compilation.

The majority of the time spent analyzing the output of the tools is on false positives. In general, the more sophisticated the analysis algorithm was, the fewer false positives existed. ITS4, a syntactic checker, produced a large amount of output, only a small part of which was useful. David Wagner’s buffer overflow tool produced far fewer false positives and the output was more manageable. Nonetheless, all the tools that we evaluated produced much more output than the average developer could be expected to comprehend, much less read.

When the tools did find an error however, they were very useful. The source file and exact line number (sometimes even the exact column number) were proffered for each error, resulting in very quick user confirmation. ITS4’s database provides detailed explanations of why it chose to flag a particular problem and even provides potential alternatives. Warnbuf is able to give exact numbers for buffer overflows in many cases. Stumoch can list the location of each instance of an unchecked error and every possible unchecked error number.

## 6.2 False Negatives in Static Analysis

When using static analysis tools, the abundance of misleading output makes two kinds of errors difficult to find: those reported and those not reported. False positives make finding errors in the output of the analysis like finding a needle in a haystack. Unfortunately, they may also lead the trusting user to believe that there is no error that could not have been caught. Ironically, this deluge of output comes because of conservative analysis in an attempt to prevent false negatives. In most cases, there is the possibility that more errors exist. Only with formal verification can the chance of false negatives be reduced.

## 6.3 General Comments on Software Fault Injection Tools

One of the main drawbacks to the SWIFI tools we used is the amount of direct tester intervention they require. With static analysis, it is only necessary to run a given tool once on an entire package of utilities. For SWIFI tools, however, it is necessary to isolate the applications in the package that should be targeted for fault injection, determine a suitable test method and fault injection profile, and then run the tests manually. It is certainly true that a good portion of the process can be automated as the tools mature; in fact, FIG already contains a substantial number of automation-related features. However, it ultimately remains the job of the tester to look through the output generated by the tests and determine where errors may have occurred.

Another issue with software fault injection that became evident while testing the `sudo` utility is the possibility that the

operating mode of the tool may be circumvented by security-related features in the programs being tested. Since many SWIFI techniques (LD\_PRELOAD hacks, using `ptrace()`, etc.) can be used for malicious as well as legitimate purposes, it is not surprising that security-conscious applications may be difficult to test with these tools. Given access to the program developers or program source code, it usually is possible for the fault injection tool designer to get around these obstacles, but she may find herself spending an increasing amount of time adding special cases to the SWIFI tool to avoid these corner cases.

As will be discussed in greater detail in the “head-to-head” comparison section below, one area in which SWIFI does not compare favorably to SA is in the precision with which it can isolate errors in the test application. At best, SWIFI can identify the particular module that is responsible for a crash, but determining the root cause of the crash and its location in the code is left to the human tester.

Finally, it is necessary to discuss for a moment the run-time overhead incurred by fault injection tools. If these tools are to be deployed in a development environment, it is important that the amount of overhead they add to the actual running time of an application is not exorbitant. To ensure that this is the case, we measured the time necessary for Berkeley DB to complete its test program (described above) with and without the FIG fault injection tool. Using FIG, we enabled various options: suppressing log messages, flushing to disk after generating a logging message (i.e., calling `fsync()` after a `write()`), and not flushing after logging. All tests were done using three runs and averaging the results.

Our tests indicated that FIG adds a modest 5% increase in running time when used without library call logging. With logging

enabled, the run-time overhead incurred by FIG increases significantly, up to around 100% with the most verbose logging options enabled. We believe that this performance is indicative of fault injection tools as a whole and is acceptable in a development environment.

## 6.4 Head-to-Head Comparison

The results from the tests that were run reveal some interesting properties of both SA and SWIFI. Figure 1 gives a visual summary of the overlapping and disparate capabilities of both tool categories.

In addition to this high-level understanding of the similarities and differences between SA and SWIFI, it is useful to delve into the details that set each tool type apart from the other. First we note that SA does not require a running system, but SWIFI does. All of our tests were done on 32-bit machines (Pentium-based), but SA allowed us to find errors present only on 64-bit machines. This property extends beyond architecture differences to even operating system and configuration variations. SWIFI is only able to check a system available to the user. In addition, it is readily apparent from our tests that SA can point out errors in software much more precisely than SWIFI and is easier to automate, although the need to filter through falsely positive error results from SA tests mitigates this advantage somewhat. Finally, we note that the tests done by SWIFI can never be as exhaustive and comprehensive as those done by SA. It is a basic fact of the nondeterministic nature of program execution that certain code paths may never be exercised by any testing tool (even though this is the primary purpose of fault injection tools), and thus some errors may go undetected. This is not an is-

sue with static analysis, which always checks all of the code made available to it.

SWIFI, on the other hand, does not require source to be present, while SA obviously does. This ability does not help a user improve the software, but it does allow evaluation and verification of functionality for a particular purpose. Because SWIFI does not depend on source, it is also independent of source language. Our testing of MySQL is partially incomplete because a module in the program is written in C++, for which there are very few sophisticated SA tools. SA and SWIFI also scale differently in relation to program size. SA scales as a function of the size of the source. SWIFI's performance is independent of source size, but it may be a function of how long the program takes to run or how many fault injection scenarios the tester decides to run on it. In addition, long-running or slow programs may take a long time to reach a state that is suitable for fault injection. A final advantage of SWIFI relates to its stated purpose of verifying error recovery methods in software packages. Static analysis may be able to determine whether or not a program tests for and attempts to handle a particular error, but it has no way of determining if the error handling code will be successful. This is the primary purpose of fault injection testing.

## 7 Further Work

It is apparent from our experiments that static analysis and software fault injection are used to achieve the same ultimate goal—more reliable software—and that it is possible to combine the two in mutually beneficial ways. Although such a utility would be limited by the language-dependency of SA and the lack of precision of SWIFI, it is possible that a tool combining the two approaches could prove valuable to software developers.

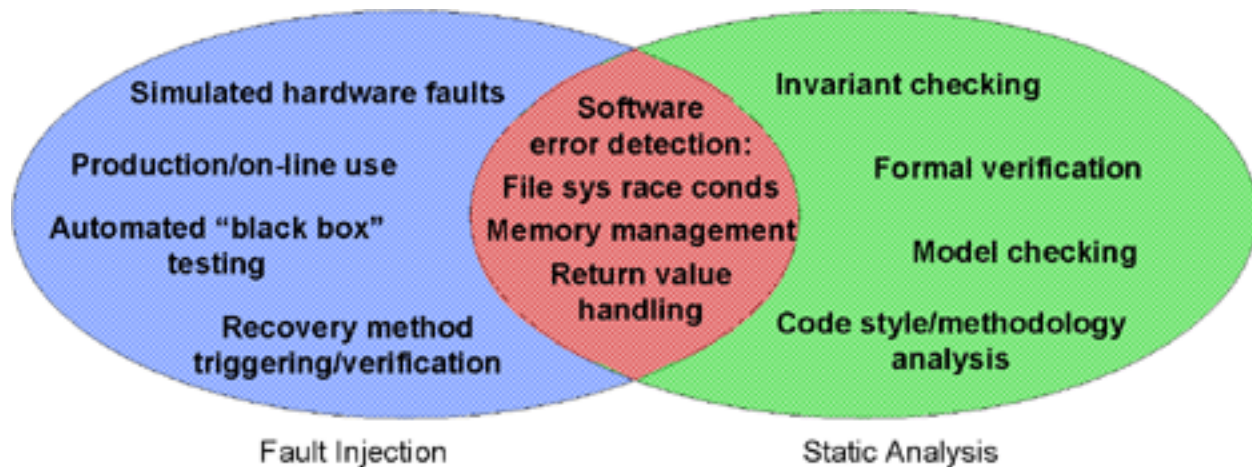


Figure 1: The overlapping verification domains of static analysis and software-implemented fault injection.

For example, the tool could first run a set of fault injection tests on a large program for which full-scale SA would be impracticable. If the SWIFI tests turned up errors or suspect behavior in any module, the SA portion of the tool could then be applied to that module.

In addition, the use of static analysis tools prior to tests with fault injection would cut down on the number of obvious programming errors that would otherwise be left to fault injection to expose. Giving the arguably lower user interaction cost of static analysis, as well as the probability that fault injection might not even detect certain programming errors, it seems advantageous to combine the tools in this manner as well.

## 8 Conclusion

We have determined that a fairly large functionality overlap exists between static analysis and fault injection methods (see Figure 1), and that within this overlap, the strengths of the two types of tools can be combined in mutually advantageous ways. In addition, we have discovered that novel

methods of extending the functionality of one type to mirror some of the capabilities of the other exist. An example of this is extending SA tools to look for unchecked return values and error types from library calls, an operation at which they are ultimately more effective than SWIFI. Another example is extending SWIFI tools to expose possible file system race conditions (through the MoveFileTrick), an operation previously carried out only by static analysis.

We have also gained insight into the relative strengths and weaknesses of the two testing realms. In particular, we have witnessed the ability of static analysis to check many things that fault injection simply is unable to test or monitor with complete effectiveness, such as memory management and program structure. On the other hand, fault injection is able to verify the correct operation of error recovery code, while static analysis can only determine that the code exists.

## 9 Acknowledgments

We would like to thank the BANE group, especially David Wagner and Jeff Foster for their help and insights. Shaun Flisakowski's CTool was invaluable in developing Stumoch. We thank him for its development and his personal assistance. Many thanks also to Mike Coleman for developing the Subterfuge library and to Naveen Sastry and Jonathan Traupman for creating the prototype FIG tool.

Finally, thanks to course instructors Mike Franklin and Kim Keeton for their guidance and suggestions throughout the project.

## References

- [1] A. Aiken and E. Wimmers. Solving Systems of Set Constraints. In *Proc. 7th Symp. Logic in Computer Science*, pages 329–340. IEEE, 1992.
- [2] H. Ammar, B. Cukic, C. Fuhrman, and A. Mili. A Comparative Analysis of Hardware and Software Fault Tolerance. *Annals of Software Engineering*, 10:103–150, 2000.
- [3] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes.
- [4] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 268–283, 2001.
- [5] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 2(2):131–152, 1996.
- [6] Peter Broadwell, Jonathan Traupman, Naveen Sastry, and David Patterson. FIG: A Prototype Tool for Online Verification of Recovery Mechanisms. Submitted to SHAMAN Workshop, ACM Supercomputing, 2002.
- [7] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.
- [8] M. Coleman. SUBTERFUGUE: A Framework for Observing and Playing with the Reality of Software. <http://subterfuge.org>, 2002.
- [9] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM, 2002.
- [10] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems Via Protocol Fault Injection. In *Symposium on Fault-Tolerant Computing*, pages 404–414, 1996.
- [11] Center for Software Engineering Research. Holodeck. <http://se.fit.edu/holodeck/>, 2002.
- [12] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. 2002.
- [13] J. Abraham G. Kanawati, N. Kanawati. FERRARI: A Tool for the Validation of System Dependability Properties. In *FTCS-22, Digest of Papers, IEEE*, pages 336–344, 1992.

- [14] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *6th Usenix Security Symposium*, 1996.
- [15] V. Gouranton and D. Le M'etayer. Formal development of static program analyzers, 1997.
- [16] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *28th International Symposium on Fault-Tolerant Computing*, pages 464–468, 1998.
- [17] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities, 2001.
- [18] Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.
- [19] Para-Protect. Secure Coding: The State of Practice, 2001.
- [20] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies. *UC Berkeley Computer Science Technical Report UCB—CSD-02-1175*, March 2002.
- [21] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers.
- [22] Internet Security Systems. Buffer Overflow in Microsoft Outlook and Outlook Express Mail Clients. [http://www.iss.net/security\\_center/alerts/advised57.php](http://www.iss.net/security_center/alerts/advised57.php), July 2000.
- [23] Timothy K. Tsai and Ravishankar K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Eighth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, 1995.
- [24] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code.
- [25] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities.
- [26] James A. Whittaker. Software's Invisible Users. *IEEE Software*, pages 84–88, May/June 2001.
- [27] L. T. Young, R. K. Iyer, K. K. Goswami, and C. Alonso. A Hybrid Monitor Assisted Fault Injection Environment. In *Third IFIP Working Conference on Dependable Computing for Critical Applications*, September 1992.