

PATH-BASED FAULT INJECTION

Timothy K. Tsai
Lucent Technologies
600 Mountain Ave.
Rm. 2B-126
Murray Hill, NJ 07974

Shambhu J. Upadhyaya
Dept. of Electrical
& Computer Engineering
State University of New York
Buffalo, New York 14260

Hong Zhao, Mei-Chen Hsueh,
and Ravishankar K. Iyer
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Keywords: fault injection, control-flow graph, software testing, dependability evaluation

Abstract

This paper presents a path-based fault-injection approach to increase the efficiency of computer system dependability evaluation. Our approach utilizes knowledge about the execution path to ensure that injected faults are activated. The control flow associated with each path is analyzed to determine the faults that would be activated by that path. By selecting a set of paths that covers a significant portion of the entire code space of the test program, a comprehensive fault set is constructed that is capable of directly affecting all control-flow decision points. In contrast to this path-based approach, common fault-injection methods randomly select fault time and location parameters without considering the accompanying path and result in a lower level of fault activation. Although the path-based approach requires a pre-injection analysis, this time-cost is only incurred once and is amortized during the fault injection phase.

1 Introduction

Fault injection is an important step in the process of validating and evaluating computer system dependability. Dependable systems incorporate fault-tolerant features to first detect errors and then to mask, avoid, or recover from the effects of those errors. Testing the dependability of these systems is equivalent to testing the functionality of the fault-tolerant mechanisms. Fault injection is essential to test these fault-tolerant mechanisms so that the system behavior can be studied *a priori* under realistic scenarios.

Although the purpose of fault injection is to force the exercise of the fault-tolerant components of the target system, not all injected faults accomplish this purpose. Faults should not be placed where activation is impossible. For instance, faults in memory locations that are not assigned to an executing process will never be accessed and therefore never cause an error detection.

In this paper, faults that are accessed are labeled as *activated*. *Path-based injection* is an approach that minimizes non-activated faults through an intelligent selection of fault parameters. Path-based injection is especially useful for testing an embedded system, which executes a single program repeatedly. For such a system, an involved effort to thoroughly test the program with fault injection is justified. Path-based injection ensures that the entire fault

set is activated with a low time-cost. The number of injected faults that are not activated is minimized to zero if the pre-injection analysis is able to find paths that utilize every resource that is to be injected with a fault. Path-based fault injection utilizes the control-flow graph of the program in conjunction with the “architecture” resources of the hardware system. This allows us to determine the location and timing of faults with a greater precision.

Related fault injection work is mentioned in Section 2. Path-based injection is described in this paper first in terms of its fundamental ideas in Section 3 and a specific software-based implementation in Section 4. The results given in Section 5 show the measured increase in fault activated gained through the use of path-based injection as compared to random injection. Section 6 contains concluding remarks.

2 Related Work

Many different approaches to fault injection have been developed. A summary of the major current techniques appears in [1],[2]. Examples of tools utilizing software-implemented injection methods include FIAT [3], FERRARI [4], DEFINE [5], DOCTOR [6], FTAPE[7], and Xception[8]. Since no additional hardware is needed, this injection method is lower in cost than hardware-implemented methods. The example implementation described in this paper uses the type of software-implemented fault injection used in the FTAPE tool [7]. The FTAPE tool has been used for benchmarking of system-level fault tolerance and to demonstrate the usefulness of stress-based injection in increasing the fault-tolerant activity caused by fault injection.

Past studies have established that error rates are influenced by the workload [9], [10]. Chillarege [11] produces “fault acceleration” by injecting faults only on a page that is currently in use and by using a workload that pushes toward the limits in CPU and I/O capacity. DEPEND [12] uses a workload-related fault injection mode where the fault rate is correlated with the resource usage (for CPU, memory, I/O, etc.). Some software-implemented fault injection tools, like FERRARI [4] and DEFINE [5], indirectly address the issue by utilizing software traps and trap handling routines to force injection of faults at a point in the control flow of the test program. However, fault ac-

tivation is not guaranteed because usage of the injected resources is not considered. A pre-injection analysis of the type used in path-based injection is needed to identify the program inputs needed to activate faults injected in those resources.

Echtle and Chen [13] have developed a program model and shown analytically that deterministic fault injection offers certain benefits over random injection in fault-tolerant protocol testing. Guthoff and Sieh [14] have utilized operational profiles of register utilization to determine times and locations for fault injection in order to minimize unactivated faults. Additionally, the times for injections were selected based upon the occurrence of register read instructions. Path-based injection uses a similar idea to inject faults only into registers that will affect the program control and data flow.

Path-based injection belongs to the same family as the techniques above. Here, however, the emphasis is on the actual control flow of the program in execution.

3 Fundamentals of Path-based Injection

The level of fault activation is defined as the fraction of injected faults that are activated. The fault activation level has a maximum level of 1.0, which is attained when every injected fault is activated.

Before discussing path-based injection, a few terms will be defined. The *test program* is the program that is executed as the fault is injected. An *input* is the set of data that the test program processes and may include command-line arguments, contents of files, environment variables, and file system state. A *path* is the sequence of test program instructions that are executed based upon a given input. *Resources* are the system state components that may be injected with a fault. A fault may have several parameters. Two important fault parameters are time and location. The time can be expressed in terms of the currently executing instruction in the test program, which is called the *stop address*. The fault location is either the register or memory location.

In order to explain the details of path-based injection, a comparison with *random injection* will be made. In this context, random injection refers to the injection of a fault with a randomly selected test program input. For purposes of comparison, if an injected fault is not activated by random injection, that fault is injected again with a different test program input. Attempts to activate the fault are made until the *surrender threshold* is reached, which is an arbitrary limit placed on the number of attempts. Thus, multiple faults may need to be injected to produce a single activated fault. This inefficiency can be reduced using path-based injection.

Path-based injection guarantees activation of each fault, as long as the fault location is in the set of live resources at the fault time for at least one path. By assembling an input set that results in paths which cover most of the test program code, a large set of interesting activatable faults can be achieved.

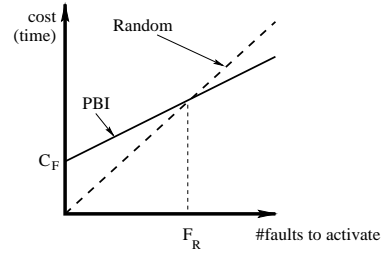


Figure 1: Time cost vs. # faults to activate

The desirability of a high fault activation level can be illustrated with Figure 1. The graph shows the cost in time incurred in order to activate a certain number of faults that are injected. The costs associated with path-based injection are represented by the solid line, while the dashed line represents the random injection costs. The slope of each line is determined by the cost to activate one fault. The time-cost for path-based injection increases less rapidly than that for random injection because path-based injection maximizes the fault activation level and thus minimizes the cost to activate each fault.

Since path-based injection requires a pre-injection analysis, a one-time cost is incurred. This cost is represented by C_F in Figure 1 and is referred to as the *fixed cost*. This fixed cost initially increases the time-cost of the path-based injection approach. However, as more faults are activated, the fixed cost is amortized and eventually at some number of faults, F_R , the total time-cost of path-based injection is lower than that for random injections. This fact is true regardless of the fixed cost of the pre-injection analysis.

The next section describes an implementation of path-based injection, which will be used to provide quantitative measurements to compare path-based and random injection. The implementation described in the remainder of this paper focuses on the injection of control-flow faults. Other type of faults can also be handled (e.g., data-flow faults if the resource analysis includes data flow among registers and memory locations).

4 Implementation

Two major tasks are needed to implement path-based injection: (1) pre-injection analysis, which associates paths with faults, and (2) injection of the fault, including selection of an appropriate input. These two tasks are described in the following subsections.

4.1 Pre-injection Analysis

As mentioned in Section 3, a pre-injection analysis must be performed in order to associate paths with faults. This task of associating paths with faults is the key to path-based injection.

The implementation was performed on a Tandem Integrity S2 fault-tolerant computer, which is based on the MIPS R2000 microprocessor. The test program is *compress*, the standard UNIX compress utility and is

written in the C language. The description of the implementation in this section refers specifically to the S2 and `compress` in order to present a concrete example. However, the same procedure can be performed using any computer system and any test program.

The following steps are needed to accomplish this task:

1. Derive an input set based upon a knowledge of the test program, including command-line options, documentation, and a knowledge of the program’s high-level language code.
2. For each input in the input set, determine the associated path. The path is represented as a list of test program basic blocks (sequential group of instructions) that are executed by the associated input.
3. Determine the faults that can be activated by each path.

The analysis of the target software program is performed at the assembly level for several reasons. First, there is no dependence on any single high-level language. Second, compiler optimizations such as the deletion, duplication, or reordering of code do not have to be considered. And third, analysis at the assembly level permits direct access to physical register names with any need to map variables to physical registers.

Deriving an Input Set:The first step, derivation of the input set, is performed manually and is not included in the fixed cost, C_F . This cost was not included in C_F because (1) timing a non-automated activity is not simple, (2) the cost is heavily dependent upon the skills of the person creating the input set, and (3) most importantly, the cost only adds to the fixed cost, which might increase F_R in Figure 1 but does not change the fact that the time-cost for path-based injection is lower than that for random injection after F_R faults are activated.

Determining the Path Associated with an Input:The second and third steps are automated, and their costs constitute the fixed cost, C_F . The second step is the most time-consuming in the pre-injection analysis. This step involves the discovery of the path associated with each input in the input set. A list of basic blocks that are executed due to a given input set is produced by the process given in Figure 2. The `compress` program contains a total of 1710 basic blocks, of which 497 basic blocks are associated with user code.

As illustrated in the flow chart in Figure 2, three programs are used to find the path for a given input: (1) `getcfg.c`, (2) `create_probe_lib.c`, and (3) `probe.c`. `getcfg.c` derives a control-flow graph for the test program, based on a disassembly of the test program. `create_probe_lib.c` then uses the control-flow graph to determine how many basic blocks the test program has and then creates a probe library containing a unique probe function for each basic block. A `probe` consists of a C-language function that records each call to that probe. This probe library is then compiled and linked in with the test program. `getcfg.c` is used to derive the control-flow

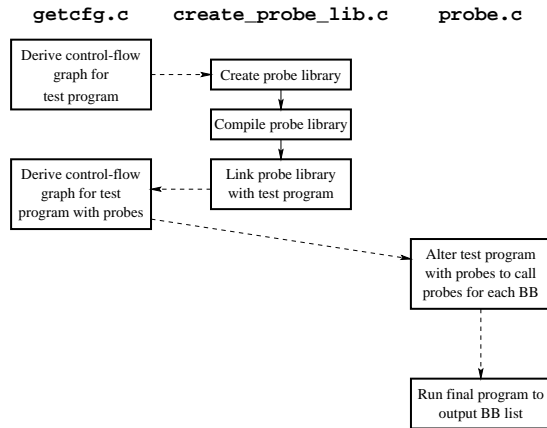


Figure 2: Flow Chart for Pre-injection Analysis Second Step

graph once again in order to account for any addresses that might have changed due to the inclusion of the probe library. `probe.c` uses this second control-flow graph to alter the new test program to call a unique probe for each executed basic block. When this is accomplished the basic block and probe are “linked.” After all basic blocks and probes have been linked, the altered program is executed and produces a list of all executed basic blocks.

Determining Faults Associated with a Path:The final step in the pre-injection analysis is to determine the faults that can be activated by each path. To simplify this step, we will only consider control-flow faults or faults that directly affect the execution of branches and jumps. With this in mind, all direct-effect control-flow faults occur when branches or jumps occur. To simplify things further, we will only inject faults into CPU registers, which means that all such control-flow faults occur at conditional branch instructions. Once the pre-injection analysis is finished, path-based fault injection can be performed. The next section describes the method used to inject faults.

4.2 Injection of Fault

The fault injection testbed is shown in Figure 3. The *target machine* is the machine on which the test program is run and faults are injected. A *DAS* logic analyzer is connected to the target machine and monitors memory bus activity in order to detect the activation of injected faults. The DAS needs to be reprogrammed before each fault injection to search for the activation of that specific fault. The reprogramming is controlled by the *control host*, which in turn communicates with the injection software on the target machine in order to know what fault is to be injected.

The injection software on the target machine consists of three programs: (1) `inject`, (2) `dbx`, and (3) `poke`. The flow chart for these three programs is given in Figure 4. `inject` is the main control program. It first initializes the DAS with the needed monitoring software and then reads in the pre-injection analysis information, which

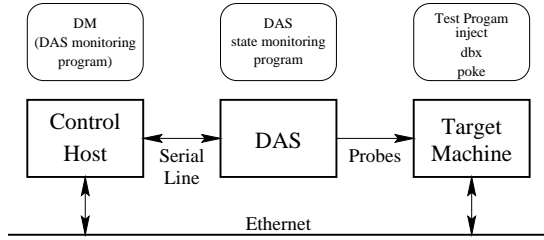


Figure 3: Setup of DAS, Host, and Target Machine

shows which inputs and faults should be paired up in order to ensure activation of the fault. `inject` selects a fault to be injected and an input that will result in the activation of that fault, and then the fault is injected. For random injection, the input is selected randomly, which may not result in the activation of the fault. If no activation occurs, then another input is chosen for the same fault, and the fault is injected again.

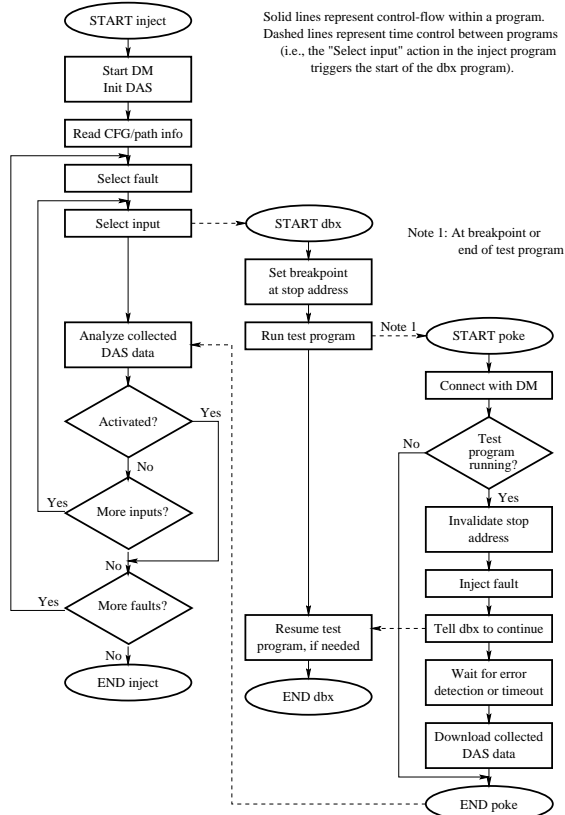


Figure 4: Flow Chart for Injection Instrumentation

`inject` calls the `dbx` program. `dbx` is a fairly common UNIX C-language debugger. `dbx` is used to control the timing of the fault injection. This is accomplished by setting a breakpoint at the stop address, which is the current test program address when the fault should be injected. `dbx` then runs the test program. If the breakpoint is encountered, the `poke` program is executed. `poke` injects the actual fault and is responsible for communicating with the control host machine to reprogram the DAS and to obtain the data collected by the DAS.

The test program used to test the implementation described in Section 4 was the `compress` program. In the pre-injection analysis, a set of 39 inputs were chosen manually based upon a knowledge of the test program functionality and command-line options.

The code coverage for our input set is given in Table 1. Here code coverage refers to the percentage of basic blocks that were executed by at least one input in the input set. Table 1 shows that 37.51% of all basic blocks were covered. However, 58.15% of the basic blocks compiled from user code were covered.

Table 1: Description of `compress` Program

	#Covered	#Total	%Covered
All basic blocks	641	1710	37.49%
User code	289	497	58.15%

Path-based injection does indeed result in a higher fault activation level than random injection. Table 2 shows that the fault activation level for path-based injection is four times higher than for random injection. In fact, for path-based injection, all injected faults were activated, which is to be expected, since that is the main purpose of path-based injection. With random injection, an average of over four injections were needed to activate each fault. Thus, the fault activation level is less than one-quarter.

Table 2: Random Injections vs. Path-based Injections

	Random	Path-based
Faults injected	623	149
Faults activated	149	149
Injections/activation	4.18	1.00
Fault activation level	0.24	1.00

More insight into the difference between path-based and random injection can be obtained by investigating the effect of an equal number of injections for both injection methods. Table 3 contains some measurements for the first 100 injections for both injection methods. Since 100 injections were performed, 100 inputs were needed for both methods. For path-based injection, all 100 inputs resulted in fault activation, while only 30 of the inputs for random injection caused fault activation. This 0.30 fault activation level is slightly higher than the 0.24 level in Table 2, but is nonetheless still well below the 1.00 level for path-based injection. Of the inputs that caused fault activation, 31 unique inputs were used for path-based injection and 23 unique inputs were used for random injection. It is interesting to note that for path-based injection a set of 31 inputs was able to activate 100 different faults, while for random injection a set of 23 inputs was only able to activate 30 different faults. Thus, with path-based injection a relatively small input set is able to activate a large number of faults. This is advantageous for validation efforts,

since the generation and management of a larger input set incurs a greater cost.

Table 3: Comparison of Paths for the First 100 Injections

Injection method	Total inputs used	Activating inputs	Unique activ. inputs
Path-based	100	100	31
Random	100	30	23

In addition to verifying that path-based injection does indeed result in a higher fault activation level, the experiments performed also allow the fixed cost (C_F in Figure 1) and the per-activated fault time-cost for path-based and random injection to be measured. With these measurements, the minimum number of faults needed to justify path-based injection (F_R) can be calculated. These values for these parameters are given in Table 4. The fixed cost of the pre-injection analysis, C_F , is 1357 seconds. This value does not include the time required to manually determine an appropriate input set, but that time was small relative to the measured C_F . The average time-cost associated with each fault before activation occurred was found to be 51 seconds for path-based injection and 213 seconds for random injection. Based on these measurements, F_R is calculated to be 8.4 faults. Thus, path-based injection is justified from a time-cost perspective if at least 9 faults are to be activated.

Table 4: Measured Values for Cost Graph (see Figure 1)

Meaning	Value
Fixed time-cost of pre-injection analysis (C_F)	1357s
Startup time-cost of injections	9s
PBI time-cost per-activated fault	51s
Random time-cost per-activated fault	213s
Min. # faults to justify PBI (F_R)	8.4

6 Conclusion

Path-based injection is a method that greatly increases the fault activation level compared to random injection. A greater fault activation level translates into either a lower time-cost to produce the same amount of activation or a greater amount of activation for the same time-cost. In other words, path-based injection allows a system to be more thoroughly tested or to be tested with a lower cost.

The purpose of path-based injection is to aid the process of dependability evaluation. Path-based injection yields a high level of fault activation, which allows a system's fault-tolerant design to be exercised with fewer fault injections. For each injection, the effect of the fault on the system needs to be studied, for instance, if and how the fault is tolerated. Thus, path-based injection should be incorporated as an element of a total dependability evaluation package.

7 Acknowledgments

This research was supported in part by the Advanced Research Projects Agency (ARPA) under contract DABT63-94-C-0045, ONR contract N00014-91-J-1116, and by NASA grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). The content of this paper does not necessarily reflect the position or policy of these agencies, and no endorsement should be inferred.

References

- [1] Dong Tang and Ravishankar K. Iyer, *Fault-Tolerant Computer System Design*, chapter 5, Prentice Hall PTR, Upper Saddle River, NJ, 1996.
- [2] J. A. Clark and Dhiraj K. Pradhan, "Fault injection: A method for validating computer-system dependability," *IEEE Computer*, vol. 28, no. 6, pp. 47-56, June 1995.
- [3] Z. Segall, D. Vrsalovic, et al., "Fiat-fault injection-based automated testing environment," in *18th International Symposium on Fault-Tolerant Computing*, 1988, pp. 102-107.
- [4] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proceedings 22nd International Symposium on Fault-Tolerant Computing*, July 1992, pp. 336-344.
- [5] Wei-Lun Kao and Ravishankar K. Iyer, *Fault-Tolerant Parallel and Distributed Systems*, chapter DEFINE: A Distributed Fault Injection and Monitoring Environment, pp. 252-259, IEEE CS Press, Los Alamitos, California, USA, 1995.
- [6] S. Han, K. G. Shin, and H. A. Rosenberg, "Doctor: An integrated software fault injection environment for distributed real-time systems," in *International Computer Performance and Dependability Symposium*, Apr. 1995, pp. 204-213.
- [7] Timothy K. Tsai and Ravishankar K. Iyer, "An approach to benchmarking of fault-tolerant commercial systems," in *Proceedings 26th International Symposium on Fault-Tolerant Computing*, Sendai, Japan, June 1996, pp. 314-323.
- [8] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software fault injection and monitoring in processor functional units," in *Proceedings 5th International Working Conference on Dependable Computing for Critical Applications*, Urbana, IL, Sept. 1995, pp. 135-149.
- [9] S. E. Butner and R. K. Iyer, "A statical study of reliability and system load at slac," in *Proceedings 10th International Symposium on Fault-Tolerant Computing*, Oct. 1980, pp. 207-209.
- [10] X. Castillo and Daniel P. Siewiorek, "Workload, performance, and reliability of digital computer systems," in *Proceedings 11th International Symposium on Fault-Tolerant Computing*, Portland, Maine, June 1981, pp. 84-89.
- [11] Ram Chillarege and Nicholas S. Bowen, "Understanding large system failures - a fault injection experiment," in *Proceedings 19th International Symposium on Fault-Tolerant Computing*, June 1989, pp. 356-363.
- [12] Kumar Goswami and Ravi Iyer, "Depend: A simulation-based environment for system level dependability analysis," Tech. Rep. CRHC-92-11, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1992.
- [13] K. Echtle and Y. Chen, "Evaluation of deterministic fault injection for fault tolerant protocol testing," in *Proceedings 21st International Symposium on Fault-Tolerant Computing*, Montreal, Canada, June 1991, pp. 418-425.
- [14] J. Guthoff and V. Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection method," in *Proceedings 25th International Symposium on Fault-Tolerant Computing*, Pasadena, California, June 1995, pp. 196-206.