Fast Fault Injection with Virtual Machines

Martin Süßkraut  Stephan Creutz  Christof Fetzer
Technische Universität Dresden
Computer Science Department
01062 Dresden, Germany
{martin.suesskraut, stephan.creutz, christof.fetzer}@inf.tu-dresden.de

1. Introduction

Fault injection is a widely used technique to test the robustness of software systems and to find bugs in applications. We focus on using fault injection to judge the quality of error handling in applications. Even mature applications have serious bugs in error handling [3]. To find all bugs related to bad error handling when using a certain API one has to do exhaustive fault injection, i.e., to inject all error values found in the API’s specification.

When doing exhaustive fault injection one has to do one fault injection experiment per API usage of the application to test. That means one need to run the application once per fault injection. So the run-time of all experiments together grows like \(O(n^2)\) in terms of API usages. One can reduce run-time complexity to \(O(n)\) by utilizing snapshots and rollbacks [2, 4]. The idea is to make a snapshot of the application directly before a fault is injected. After the fault injection experiment is finished the application’s state is rolled back to the last snapshot. In this way one can reuse the progress until the fault injection an application has already achieved.

Figure 1 illustrates such a single fault injection run. The original fault free execution of the application is bold. The run has 3 API usages. Each API usage is a fault injection point. Due to an API usage, e.g., a function call, can fail for multiple reasons usually more than one fault needs to be injected per fault injection point to do an exhaustive analysis. In the example, at the first and the third fault injection point two faults need to be injected and at the second one three faults. The different faults are derived from the specified error values of the called API functions. In our experiments we are not interested in a complete run after the fault has been injected. It is sufficient to execute a limited number of instructions after injecting the fault, e.g., until the currently executed function returns to its caller. That is why, the run-time complexity is \(O(n)\) instead of \(O(n^2)\) when doing each experiment in a separate run.

To underpin the complexity issues, we have parsed the documentation of the Linux 2.6.15 system call API. We have found in average 3.3 possible error return codes (faults) per system call. The maximum is `execve` with 22 unique error return codes. GCC performs more than 1,800 system calls when compiling and linking a Hello World program. Whereas compiling and linking the GNU Math Library issues more than 2.5 million system calls. If one wants to do fault injection for this run, one has to run more than 8 million fault injection experiments.

In our previous experiments, we did the snapshot by spawning a new child process via `fork`. The parent process waits until the fault injection experiment in the child process has finished. If another fault has to be injected at the current fault injection point, a new child is spawned. Otherwise, the parent continues the execution to the next fault injection point. In our recent research, we found that the snapshots done with `fork` are incomplete for various reasons:

- **Multi-Threading** Applications that use multiple processes to implement multi-threading are not supported by `fork`. The parent process waits until the fault injection experiment in the child process has finished. If another fault has to be injected at the current fault injection point, a new child is spawned. Otherwise, the parent continues the execution to the next fault injection point. In our recent research, we found that the snapshots done with `fork` are incomplete for various reasons:

- **File System** The snapshots do not contain the state of the file system. A fault injection execution might alter files in a way that influences the original execution.

- **Shared Memory** If the snapshot is done via `fork`, the original execution might be influenced by changes to shared memory. It is enough for an application to accidentally map private data as shared. It will see all changes done by the fault injection experiments.
• **Operation System Resources** The fault injection execution might bind certain resources that can therefore not be acquired by the original execution. One example for such resources are the System-V IPC Semaphores. A Semaphore hold by a process is not automatically freed upon its termination.

• **Indirect Influence** Furthermore, a fault injection execution might influence another application which itself influences the original execution again. This can be very difficult to detect.

• **Distributed Applications** The fork snapshot captures only one application. Applications distributed over more than one process or more than one host cannot be taken into one consistent snapshot.

Therefore, we propose to include the complete state of one computer into the snapshot. In the case of distributed applications, the states of all participating computers have to be included. In this way, we avoid interference between fault injection execution and original execution. Virtualization tools provide snapshots of virtual machines that include the states of the CPU, the virtual devices, the volatile and the stable storage. But there are some problems with current VM toolkits: (1) none of them are optimized for fast snapshots and rollbacks, and (2) we need to store the analyses of fault injection executions outside of the virtual machine. Otherwise, it will be discarded when rolling back to the last snapshot.

2. **Virtual Machines for Fault Injection**

Our approach is to do the fault injection within a virtual machine. We currently implement fast snapshots and rollbacks for XEN [1]. Our plan is to avoid snapshots of the file system by splitting it into an immutable part mounted as read only and a mutable part within a ram disk. Both parts are joined together by UnionFS [5]. The mutable part of the file system is stored in the virtual machine’s RAM. The RAM itself is saved within a snapshot via copy-on-write.

The following pseudo code shows how the fault injection – running within a virtual machine – cooperates with our snapshot/rollback approach:

```plaintext
1 set_state (ERROR_INJECT)
2 snapshot ()
3 if get_state () == ERROR_INJECT
4 do_fault_injection ()
5 log_analysis_results ()
6 set_state (NEXT_ERROR)
7 rollback ()
8 end
```

Because the rollback discards all changes done since the last snapshot, we have to store the analysis results outside of the virtual machine (log_analysis_results). Also, the information about the completion of the current fault injection is lost. Therefore, we need an external state that is not influenced by the rollback. The functions `get_state` and `set_state` are used to access the external state. Currently we only need one bit.

The snapshot and rollback tool run on the VM’s host (or the privileged domain XEN’s case) as shown in Figure 2. The host stores also the external state and the fault injection results. The experiments are controlled from within the virtual machine 1 by the fault injection tool. If an experiment is distributed over more than one computer, all involved computers run as virtual machines on the VM’s host. In this way, a snapshot of all of them can be created.

3. **Conclusion**

Fault injection experiments have usually a high run-time complexity. Reusing already achieved progress by doing snapshot and rollback can reduce the run-time complexity from $O(n^2)$ to $O(n)$. But one must be careful to include everything that will be changed by the fault injection execution within the snapshot. Therefore, we propose using virtual machines. With virtual machines it is even possible to easily run fault injection experiments on distributed applications.

References


