Hardware Fault Injection Using Dynamic Binary Instrumentation: FITgrind

Ute Wappler  Christof Fetzer
Technische Universität Dresden
Department of Computer Science
Dresden, Germany
{ute.wappler,christof.fetzer}@inf.tu-dresden.de

Abstract

To test software implemented hardware fault tolerance (SIHFT) mechanisms, injection of hardware faults is the appropriate instrument. Existing tools are mostly processor dependent, difficult to use, and do not allow for a fine grained fault propagation analysis. FITgrind, which uses an artificial hardware architecture provided by Valgrind [6], alleviates these issues.

1 Motivation

For testing fault tolerance mechanisms fault injection is a useful and accepted tool. Injection of transient, intermittent, and permanent hardware faults is required to analyze SIHFT whose objective is to detect such hardware faults. The most realistic but also most expensive hardware fault injection approaches are hardware-based and use, e.g., radiation or pin-level fault injection. Since these are very costly and might even destroy the hardware, many software-based solutions have been developed over time, which either use simulation of a hardware architecture or inject faults via changing the execution of a program either by directly modifying hardware or software state.

We have been developing a new fault injection tool FITgrind which uses dynamic binary instrumentation provided by Valgrind [6]. The tool abstracts from the underlying hardware architecture and faults are injected into the artificial architecture provided by Valgrind. The tool alleviates problems of existing tools, such as Xception [1] or FERRARI [5]:

- Processor independence which is neither provided by FERRARI nor Xception. Ferrari needs specification of the processor’s instruction set and Xception uses the processor’s debugging features. FITgrind can be used on all platforms supported by Valgrind which are currently x86, amd64, ppc32 and ppc64 running under Linux.
- Monitoring & taint analysis: similar to [2] or [7] is realizable using Valgrind’s API. This will enable fine granular analysis of the propagation of faults and thus, a precise evaluation of SIHFT mechanisms. To the best of our knowledge none of the existing injection tools supports fine-grained fault propagation analysis. At best, they log the context when a fault is injected, where which fault is injected and the outcome of the fault injection run.
- Ease of use: no specification of a hardware model (FERRARI), and no difficult testbed requiring more than one machine (Xception) are necessary.
- Injection into binaries: faults can be injected into binaries without recompiling them to assess their fault tolerance. Additionally, recompilation and a light-weight instrumentation will give the user even more control over the fault injection process and the ability to do a fine-grained taint analysis.
- Ease of implementation and extensibility: the first version of FITgrind which is able to inject transient bit flips on operands and results and transient or permanent replacement of instructions, requires only about 1000 lines of C code and is easy to extend. For each new fault type injection one has to provide a method to select the injection points and a method to add the instrumentation code.

The following fault types can in principle be injected by binary instrumentation:

- Modification of operands and results to simulate bit flips or stuck-at faults in memory, registers or on buses.
- Exchange of operands with other operands to simulate address line faults.
- Replacement of instructions with other valid instructions or groups of instructions to simulate address line faults.
- Faulty instructions execution to simulate bugs such as the famous Pentium FDiv bug [4]. Note that such a bug has to be mapped onto the hardware architecture simulated by Valgrind.
- Modification of jump conditions and destinations to simulate control flow errors.

It would be also possible to simulate bit flips in instruc-
tions or operand addresses, but most of these injections will be rejected by Valgrind, since they generate invalid code. For emulating software faults by instrumenting binaries a similar approach as described in [3] could be used.

2 Design and Implementation

Valgrind translates every basic block encountered during execution into UCode which forms the hardware model on which FITgrind is injecting faults. UCode is a single-assignment load-store architecture. The resulting basic block is given to FITgrind for instrumentation. The instrumented basic block is transformed from UCode into native binary code by Valgrind. The result is stored into a cache for later reuse and then executed. After that, the next basic block which is the destination of the jump leaving the previous one is translated and instrumented in the same fashion.

FITgrind instrumentation chooses stochastically locations within a basic block for fault injection and extends the UCode accordingly. UCode can either be modified or extended with additional UCode instructions or with so-called dirty-helpers which are called into C functions. When, for example, an instruction for a result modification is chosen, the result of the original instruction is redirected into another register by modifying the UCode. Than a dirty helper is called which randomly either stores the original result into the original destination register or a modified version. All other fault types can be implemented in a similar fashion. The fault types and their probabilities are configured by the user. The tool could also be extended to support a more detailed choice of fault injection points, e.g., by time or instruction type as provided by Xception and FERRARI.

FITgrind allows the user to define fault injection campaigns with given fault probabilities, fault types, and target applications. One campaign consists of several fault injection runs which differ in the chosen seed for the random number generator controlling the fault injection process and one golden run in which no faults are injected. Every fault injection run is repeatable by choosing the same configuration of probabilities, fault types and seed. The output of a fault injection run and the golden run are compared byte wise. The following results are possible: no output, correct output, correct but incomplete, partially correct, completely incorrect and correct but too big. The last three represent non-fail-safe runs. The error output of the program is parsed to recognize if an error was detected, e.g., by the operating system. All results, the fault injection configuration and the types and number of injected faults are stored in a database.

3 Results

We tested the first version of FITgrind with a recursive grep as target application for which we executed 1,300 faulty runs. One fault injection run was on average 21.5 times slower than the non fault injection run which is the same slow down Valgrind generates when no instrumentation is done at all. The time measurements were taken with adding the instrumentation for triggering faults but without actually triggering them. When triggering faults, the runs are almost always faster, because of crash failures. Table 1 shows how many and which errors were recognized by the execution environment which was in this case FITgrind on Ubuntu Linux. 77.9% of the faulty runs crashed or were killed by the fault injection environment. 4.6% generated correct results and 89.2% were fail-safe (no output or correct but incomplete). However, for 6.2% of the injections the output differed from the golden run, i.e., was not fail-safe. A fine grained fault propagation analysis can show if an injected error influenced the execution. These results demonstrate the necessity for SIHFT mechanisms to turn such runs into fail-safe runs. We are developing FITgrind to test our own SIHFT implementation. In this context, the tool already proved useful since it detected runs were the environment was not able to detect a failure and faulty output was generated instead of switching to a fail-safe mode.

References