L4Sys Fault Injection Campaign – User Manual

Martin Unzner (munzner@os.inf.tu-dresden.de),
Björn Döbel (doebel@os.inf.tu-dresden.de)

August 8, 2013

Abstract

This document describes how to use the L4Sys experiment suite. However, this is not a complete documentation. When in doubt, please read the source code or contact me. Still, I would like you to read this whole document before investigating further.

1 Overview

This is the user manual of the L4Sys generic system test framework. The framework builds on Fail* and provides means to perform fault injection experiments for applications running on top of the Fiasco.OC/L4Re microkernel-based operating system as well as the underlying microkernel.
L4Sys provides four experiment types:

1. *GPRFlip* simulates bit flips in general purpose registers.
2. *RATFlip* simulates errors in the association between the physical register file and general purpose registers.
3. *IDCFlip* simulates errors occurring during instruction decoding.
4. *ALUInstrFlip* simulates errors in the processor's arithmetic logic unit.

L4Sys currently works for x86/32 running in Fail/Bochs only. This is partly due to some issues with timing — as soon as a valid model of time in the target emulator as well as an assembler/disassembler functionality in the Fail* framework are established, I would recommend a backend change, as Bochs' reliability is very limited.

2 Framework Setup

To prepare a fault injection campaign you will first need to configure and build Fail* itself. This process is described in `doc/how-to-build.txt`. The following CMake flags need to be set:

- `BUILD_BOCHS = ON`
- `BUILD_X86 = ON`
- `CONFIG_BOCHS_NO_ABORT = ON`
- `CONFIG_EVENT_BREAKPOINTS = ON`
- `CONFIG_EVENT_IOPORT = ON`

- `CONFIG_SR_RESTORE = ON`
- `CONFIG_SR_SAVE = ON`
- `EXPERIMENTS_ACTIVATED = l4-sys`

Enabling `CONFIG_FAST_BREAKPOINTS` may speed up the experiment clients significantly. Enabling `CONFIG_BOCHS_NO_ABORT` is necessary to detect whether Bochs stopped because of a bad instruction induced by IDCFlip. Keep in mind that this implies the risk of a deadlock in the campaign system, because packets (i.e. experiment descriptions) are resent if the client does not answer and finally, all clients might fail because they tried to execute the same faulty instruction.

3 Emulator Setup

The next step is to prepare an L4Re application setup to run in Bochs. To setup your system, first, you need a dedicated `bochsrc` file. It has proven useful to have a Bochs resource file or an independent Bochs instance with GUI enabled for the initial testing, however the experiments are intended to be conducted without graphical output.

Bochs should be booted using a CD image containing your setup. To obtain this setup, first build Fiasco.OC and L4Re separately as described in their respective build instructions. Make sure your setup is running, e.g., in QEMU. Once this works, create an ISO image using the L4 build system's `make grub2iso E=<entry>` command. Validate that this ISO boots and runs in Bochs.

4 Client Setup

Now that we have Fail* and the L4Re setup running, we can prepare our fault injection campaign. This requires three (+ one optional) steps:

1. *OPTIONAL:* If we want to perform a campaign that only targets a single application, we need to determine this application's address space ID.
2. *REQUIRED:* We perform an initial run of our setup in Bochs until the point where Bochs is booted and the application in question starts. At this point we take a snapshot of the emulator so that we can skip everything upfront in the remaining runs.
3. *REQUIRED:* The L4Sys campaign uses `L4SYS_NUM_INSTR` to determine the set of instructions to inject faults in. We need to perform one run of our setup to determine this number.
4. *REQUIRED:* We need to perform a *golden run* without any fault injections. Later faults are then compared against this run.

All parameters of the L4Sys experiment can be configured via file `experimentInfo.hpp`. Normally, it should not be necessary to change the program flow directly. However, the interested reader is invited to take a look at `experiment.cc`, too.

4.1 Constants

Some values are constant throughout all steps of the preparation and also when the workload program is run. The most important constant is `L4SYS_BOCHS_IPS`, which has to be consistent with your `bochsrc` setting and is used for several timely calculations in the client.

4.2 Step 0: Determine the address space

First, we need to find the start and end instruction addresses for our workload program and our given experiment. For this purpose, use a disassembler, such as `objdump` or *IDA Pro*. Determine the first and last instruction for your campaign and set `L4SYS_FUNC_ENTRY` and `L4SYS_FUNC_EXIT` in the header file accordingly. `L4SYS_NUMINSTR` is determined automatically in a later preparation step and can be ignored for now.

If you want your campaign only to affect a specific address space (e.g., because you are only interested in faults at the application level), L4Sys leverages Fail*'s address space filtering mechanism. To determine the address space identifier, you will have to use Bochs' internal debugger and perform the following actions:

1. Compile Bochs with support for the internal debugger. This can either be done by configuring and rebuilding the fail client accordingly or using a separate Bochs installation - we don't need Fail* functionality here.[1]
2. Boot your system in Bochs. The debugger prompt (or window) will appear. Use the `lbreak` command to set an instruction breakpoint to an address in your application. (Hint: Remember you already figured out `L4SYS_FUNC_ENTRY` previously.)
3. Run Bochs until the breakpoint is hit. Verify that you are in the right address space (instruction pointers may be similar in different applications as L4's BID links all programs to the same starting address by default).
4. Use the `creg` command to look at the current control registers. Set `L4SYS_ADDRESS_SPACE` to the value of the CR3 (page table control) register.

If you are not interested in address space filtering, you may set `L4SYS_ADDRESS_SPACE` to `ANY_ADDR`. Note that in this case you will probably encounter instruction pointers across various address spaces and may not get the unique results you want.

4.3 Step 1: Save the initial state of the machine

Make sure `PREPARATION_STEP` is still set to `1`, and you have set `L4SYS_ADDRESS_SPACE` accordingly. Now recompile and execute the framework code again, this time with the graphical user interface disabled. The experiment client runs until `L4SYS_FUNC_ENTRY` is reached and then saves the complete configuration.

[1]BD: I saw differing values when using another Bochs installation, though. Perhaps it's safer to use the same Bochs build for testing and injection.

4.4 Step 2: Determine the instructions to execute

For this part, it depends on how you want to conduct the injection experiments. Setting `L4SYS_FILTER_INSTRUCTIONS` stores all instructions by default, and enables the filter functionality to store only those instructions that match the filter. Each instruction in the trace requires an address plus an unsigned breakpoint counter, which means 8 bytes per instruction on a 32-bit system and 12 bytes per instruction on a 64-bit system.

If `L4SYS_FILTER_INSTRUCTIONS` is not set, the instruction to perform the fault injection at is determined by single-stepping through the program from the beginning, which is quite slow. I only recommend it for long programs, where a complete instruction trace would require several hundred megabytes of data.

No matter which method you choose, the default implementation of the campaign server reads the total instruction count from `L4SYS_NUMINSTR`. Thus, it is mandatory to set this value to the number of instructions available.

To obtain this number and optionally the instruction trace, set `PREPARATION_STEP` to `2` and recompile, then execute the experiment client. You do not have to pass parameters to Bochs any more, because the configuration is overwritten with the state saved in step 1.

After the program has finished, you will get a summary on the total of instructions executed.

If you have `L4SYS_FILTER_INSTRUCTIONS` enabled, this is not the value you look for; it merely claims how many instructions have been processed at all. To set `L4SYS_NUMINSTR` correctly, you need to look for the number before the word *accepted*, which points out how many instructions have been accepted by the applied filter. Of course, if no filtering is selected, these two figures should be equal. Please contact me if that is not the case.

If `L4SYS_FILTER_INSTRUCTIONS` is disabled, you should get a statistical output on how many of the instructions were executed in userland and kernel space, respectively, but the interesting figure in this case is of course the overall sum of executed instructions.

4.5 Step 3: Determine the correct output

This is the easiest step: Set `PREPARATION_STEP` to `3`, recompile the client and execute it in the target directory. It runs the complete program and logs the output. You can check the resulting file (by default `golden.out`), and if it does not comply with your expectations of a valid run, you should correct the entry and exit point, the address space or, in the worst case, your Bochs settings.

5 Campaign Setup

To setup the actual campaign, you need to edit `campaign.cc`. The full language capabilities of `AspectC++` are at your hand to define the course of your experiments; a sample covering all experiment types at random is already provided. In the experiment client, set `PREPARATION_STEP` to `0`, which means there is nothing more to prepare.

After you have successfully compiled both programs, you need to start both the campaign server (`l4-sys-server`) and the experiment client. By default, they should run on the same machine, but you can adapt the `L4SysExperiment` constructor in `experiment.cc` to connect the `JobClient` to a remote server instead of `localhost`. Each experiment client processes exactly one experiment and exits. To complete your campaign, you should use the `client.sh` script in the `scripts` subdirectory of Fail*.

6 Format of the result file

When the campaign is finished, the campaign server generates a report file (by default called `lfsys.csv`) in a primitive CSV dialect. The only syntax rules are that the columns are separated by commas, that the respective data sets are separated by line breaks (`\n`), and that the cells do not contain line breaks or commas.

This section lists and describes the columns in the report generated by the campaign server, from left to right.

1. `exp_type`
   Names the experiment that generated the return data. If it is none of the following, a writing error occurred:

   - Unknown
   - GPR Flip
   - RAT Flip
   - IDC Flip
   - ALU Instr Flip

   For *Unknown*, a debug report should be provided. If not, something went completely wrong, and you should check the logs.

2. `injection_ip`
   The instruction pointer of the fault injection in lowercase hexadecimal notation. Note that the injection happens right *before* this instruction.

3. `register`
   When the fault injection experiment affects a general purpose register, it is listed here. This column should have one of the following values; if it does not, a writing error occurred:

   (a) Unknown
   (b) EAX
   (c) ECX
   (d) EDX
   (e) EBX
   (f) ESP
   (g) EBP
   (h) ESI
   (i) EDI

4. `instr_offset`
   The offset of the executed instruction, relative to either all executed instructions or to all listed instructions in case you applied a filter (see above). This offset includes multiple runs of the same instruction. For

example, this is useful when you have loops in you program and need a rough idea how many runs your loop had executed until the injection.

5. `injection_bit`
   The bit at which the injection was performed. This value is only used in GPRFlip and IDCFlip. GPRFlip inverts the bit at position `injection_bit` in the register, counted from the right. IDCFlip inverts the bit at position `injection_bit` of the current instruction, counted from the left.
6. `resulttype`
   The result of the fault injection. This column should have one of the following values; if it does not, a writing error occurred:
   (a) Unknown
   (b) No effect
   (c) Incomplete execution
   (d) Crash
   (e) Silent data corruption
   (f) Error
7. `resultdata`
   The meaning of this field can vary for each experiment. At the moment, all of the experiments use it to store the last instruction pointer of the emulator (in decimal notation). This information can be used to determine when a fault turned into a failure.
8. `output`
   The output on the EIA-232 serial line generated by the workload program. Undisplayable or reserved characters are escaped in a C conformant octal manner (e.g. `\033` for the Escape character).
9. `details`
   This column provides various details on the experiment run, which may help to trace errors or to reconstruct the injected fault. ALUInstrFlip uses this column to provide the opcode of the new instruction.

7 Known bugs

If you need support for more than one processor, you will have to extend the code accordingly: at the moment, when in doubt, it uses the first CPU.

8 To Be Continued

This is everything I consider important so far. If you still encounter problems you may contact me and I will try to set the record straight. Happy experimenting! :)