VULFI: An Instruction-level Fault Injection Framework

User Manual

Vishal Sharma Ganesh Gopalakrishan

School of Computing, University of Utah

{vcsharma,ganesh}@cs.utah.edu

Version 1.0, last updated January 4, 2016

Contents

1 Getting Started **3**

2 Requirements **3**
2.1 Supported Languages 3
2.2 Software Requirements 3

3 License & Copyright Information **3**

4 Installation **3**

5 Usage **4**
5.1 Step 1: Instrumenting a Target Program for Fault Injection 4
5.2 An Example Instrumentation 6
5.3 Step 2: Automating Fault Injection using Python Script 7

6 Examples **9**
6.1 Example 1 : Scalar Benchmark - Quicksort 9
6.1.1 Step 1: Compile 9
6.1.2 Step 2: Run 9
6.1.3 Step 3: Result Analysis 10
6.2 Example 2 : ISPC Benchmark - Raytracing 10
6.2.1 Step 1: Compile 10
6.2.2 Step 2: Run 10
6.2.3 Step 3: Result Analysis 10

1 Getting Started

VULFI is an instruction-level fault injection framework which is developed using LLVM compiler infrastructure[2, 3]. VULFI targets LLVM's intermediate representation (IR) level instructions for fault injections. VULFI is capable of targeting vector instructions (including architecture specific intrinsics) in addition to scalar instructions for fault injections. Visit VULFI's home page (`http://formalverification.cs.utah.edu/fmr/vulfi/`) and fill out the request form to obtain a copy of VULFI.

2 Requirements

2.1 Supported Languages

VULFI is capable of targeting any high-level program which can be compiled into LLVM IR. Currently, VULFI has been extensively tested with C, C++, and ISPC programs.

2.2 Software Requirements

- **OS** : Ubuntu 12.04 LTS.
- **LLVM** : VULFI has been developed using LLVM version 3.2. LLVM documentation: `http://llvm.org/releases/3.2/docs/GettingStarted.html#getting-started`.
- **ISPC**: In case you are targeting ISPC programs for the fault injection, please use ISPC version 1.8.1 (download link: `http://sourceforge.net/projects/ispcmirror/files/`)
- **Python**: Install Python version 2.7 or later.
- **OpenCV** (optional) : Install OpenCV version 2.3.1. Installation steps: `http://www.samontab.com/web/2012/06/installing-opencv-2-4-1-ubuntu-12-04-lts/`.

3 License & Copyright Information

Before using VULFI, please refer to the `LICENSE` file located in the topmost directory of the distribution. Please note that the VULFI distribution contains 3rd party programs which may have different license and copyright requirements than VULFI. You must always adhere to the individual license and copyright requirements of the 3rd party programs in addition to the VULFI license and copyright requirements. The license and copyright information for the 3rd party programs is provided in the `LICENSE` file.

4 Installation

1. Change directory to the LLVM lib directory and copy VULFI source code to the current directory.

```
$ cd [LLVM_SRC_ROOT_DIR]/lib/
```

2. Change directory to the VULFI lib directory.

```
$ cd vulfi/lib
```

3. Run below commands to install a shared library (`relief.so`) containing VULFI implementation .

```
$ make
$ sudo make install
```

5 Usage

Performing fault injection using VULFI is a two-step process. First, we instrument a target program using VULFI. This step involves compiling the target program into an LLVM bitcode file. The target bitcode file is then linked with the VULFI's `Corrupt.bc` bitcode file containing callable runtime fault injection APIs. The resultant bitcode is then instrumented using the VULFI LLVM pass. Second, we execute the resultant instrumented binary using python script for performing fault injection. The automation script provides a rich set of command line options to effectively analyze the result of a fault injection run, and for generating a detailed fault injection report.

5.1 Step 1: Instrumenting a Target Program for Fault Injection

Given that VULFI is implemented as an LLVM pass, a target program bitcode is instrumented by running VULFI on it with `opt` (`http://llvm.org/docs/WritingAnLLVMPass.html#running-a-pass-with-opt`). Below are the command line options supported by VULFI:

```
-fn                                  Name of the functions (separated by a single
                                     white space) to be targeted for fault injections.
                                     Example: -fn "foo1 foo2 foo3"

-fsa {data,dint,dflo,ctrl,addr}      Fault site selection algorithm; valid options:
                                     data: Target all pure-data fault sites for fault
                                     injection. Specifically, target all fault sites
                                     with its: (1) Lvalues of Integer or FloatingPoint
                                     types, and (2) forward program slice of the
                                     fault sites does contain an instruction
                                     with Lvalue of Pointer or Control type.

                                     dint: Same as "data" option except that target
                                     fault sites of only Integer types.
```

	dflo: Same as "data" option except that target fault sites of only FloatingPoint types. Note:- For Integer type, the supported bit-widths are 8, 16, 32, and 64. For FloatingPoint type, the supported formats are single (32-bit) and double precision (64-bit) floating-point values. ctrl: Target all fault sites of Integer or FloatingPoint type such that its forward program slice must contain atleast one contol instruction (e.g., "cmp" instruction). addr: Target all fault sites which either have its Lvalue of Pointer type or its forward program slice must contain atleast one instruction of Pointer type. (e.g., "getElementPtr" instruction).
-rf {0,1}	Range flag used to indicate if fault injection has to be done in a specific code section (using the source code line numbers). It is disabled (set to 0) by deafult.
-flr	Name of the csv file containing range information. This is an optional argument used only when -rf is enabled.
-lang {C,C++,ISPC}	The language option is used by VULFI to check for language specific information which may be useful for fault injection. For example, language specific metadata information.
-arch {x86,neon,nvvm,mips,spu}	This architecture information is used when instrumenting architecture specific intrinsics for fault injection. This includes information such as whether an intrinsic uses mask. Currently, x86 is the only supported architecture; other architectures are just a placeholder and may be supported in future.
-dbgf	Name of the CSV file to which the list of static fault sites (instrumented for runtime fault injection) is written.

5.2 An Example Instrumentation

For a target C++ program file `foo.cpp` having function definitions `foo()` and `main()`, the first mandatory step is to add following function prototypes at the beginning of the file:

```
extern int printFaultSitesData(void);
extern int printFaultInjectionData(void);
```

Next, add the below functions calls in the function `main()` just before every return statements.

```
printFaultSitesData();
printFaultInjectionData();
```

The modified C++ program file `foo.cpp` is then compiled into a bitcode `foo.bc` as shown below:

```
 $ clang++ -emit-llvm foo.cpp -c -o foo.bc
```

Similarly, the program file `Corrupt.C` located in the directory `runtime/`, is compiled into a bitcode `Corrupt.bc`. When using clang++ to compile the Corrupt.C file, add a macro definition "__CLANGPP" as described below. However, you must not use this macro when using clang to compile the Corrupt.C file.

```
$ clang++ -emit-llvm -D__CLANGPP Corrupt.C -c -o Corrupt.bc
```

Next, both the bitcode files are linked using `llvm-link`:

```
$ llvm-link Corrupt.bc foo.bc -o foo-corrupt.bc
```

The resultant bitcode file is then instrumented by running VULFI pass using `opt` as shown below:

```
$ opt -load /usr/local/lib/relief.so -vulfi  -fn "foo" -fsa "data" -lang "C++"
-dbgf "instruction_list_foo.csv" < foo-corrupt.bc > foo-inject.bc
```

Finally, the instrumented and the uninstrumented binaries (`foo-inject` and `foo` respectively) are generated.

```
$ clang++  foo-inject.bc -o foo-inject

$ clang++  foo.bc -o foo
```

Both the binaries are executed under a given program input. The execution output of `foo-inject` is compared with that of `foo` to detect the occurrence of a silent data corruption in the former execution output.

5.3 Step 2: Automating Fault Injection using Python Script

A binary instrumented using VULFI reads fault site selection and fault injection related information from a configuration file. The python script `driver.py` (located at `vulfi/scripts/`) automates the creation of the configuration file, running the instrumented binary, and generating the fault injection report. Therefore, we must always using the python script `driver.py` for fault injection runs. The command line options supported by the python script are listed below:

```
Usage: python driver.py [-h] [-e] [-s] [-c] [--ov] [--pv] [--iter EXEC_COUNT]
                  [--fcp bex,img,num] [--th TH] [--exec1 EXEC1] [--cmd1 CMD1]
                  [--out1 OUT1] [--cho] [--cfs] [--exec2 EXEC2] [--cmd2 CMD2]
                  [--out2 OUT2] [--rslt RSLT] [--ficsv FICSV] [--fia cbr,abr]
                  [--fib 1,2,3,4,5,6,7,8] [--fid msb,lsb] [--fbu FBU]
                  [--fbl FBL] [--np NUM_PROB] [--dp DEN_PROB]
                  [--ff eql,max,min,nlm] [--fc FICOUNT]

Descriptions of the arguments:
-h, --help                Shows this help message and exits.

-e                        Execution mode.

-s                        Standalone mode, always used in combination with
                          -e option.

-c                        Comparison mode, always used in combination with
                          -e option.

--exec1 EXEC1             Location of the 1st executable, EXEC1, to be
                          executed with -s or -c mode.

--cmd1 CMD1               Command line args (if any) for the exe EXEC1.

--out1 OUT1               Name (including complete path) of the output file
                          generated by EXEC1.

--exec2 EXEC2             Location of the instrumented executable, EXEC2,
                          to be executed only with -c.

--cmd2 CMD2               Command line args (if any) for the exe EXEC2.

--out2 OUT2               Name (including complete path) of the output file
                          generated by EXEC2.

--ov                      Calculate overhead.

--pv                      Override user provided fault injection probability
                          and perform fault injection using a fault injection
                          probability of 1/N where N is the total number of
                          dynamic instructions.
```

```
--iter EXEC_COUNT          Number of iterations.

--fcp {bex,img,num}        Method used for comparing outputs of a fault-free
                           execution and a fault execution. Valid options for
                           comparing the outputs:
                           1. bex = Compare and match the outputs bit by bit.
                           2. img = If the outputs are image files, perform
                           PSNR based comparison.
                           3. num = If the outputs are of numeric type then
                           express the difference between the outputs
                           as L2Norm.

--th TH                    Threshold value; only valid with --fcp=img or num.
                           Report SDC if the difference in the outputs calculated
                           using "img" or "num" based method exceeds the threshold TH.

--cho                      This performance flag indicates that use cached output
                           for exec1 if available. This is disabled by default.
                           WARNING: Enable this option ONLY if you are sure that the
                           execution output of exec1 is expected to be the same
                           during every execution iteration.

--cfs                      This performance flag indicates that use cached fault
                           site count. Use it only when --cho flag is enabled.

--rslt RSLT                Name of the output file where result will be written.

--ficsv FICSV              Name of the input csv file where fault injection report
                            information will be written.

--fia {cbr,abr}            Fault injection algorithm, valid options:
                           1. cbr - cumulative byte ordering; target one or more
                           contiguous bytes for fault injection.
                           2. abr - absolute bit range;target a bit-range for fault
                           injection.

--fib {1,2,3,4,5,6,7,8}    Number of contiguous bytes to be considered for fault
                           injection.
                           Note: this option must be provided when using -fia=cbr

--fid {msb,lsb}            Direction from which no. of contiguous bytes to be
                           considered for fault injection.
                           Note:this option must be provided when using -fia=cbr.

--fbu FBU                  Upper bound for the fault injection bit-range.
                           Note:this option must be provided when using -fia=abr.

--fbl FBL                  Lower bound for the fault injection bit-range.
                           Note:this option must be provided when using -fia=abr.

--np NUM_PROB              Numerator of fault injection probability expressed as a
```

```
                          fraction.

--dp DEN_PROB             Denominator of fault injection probability expressed as
                          a fraction.

--ff{eql,max,min,nlm}     Puts restriction on the fault injection count;
                          valid options:
                          1. eql - continue injecting faults until fault injection
                          count becomes equal to the value passed using --fc.
                          2. max - continue injecting faults until fault injection
                          count is less than or equal to the value passed using
                          --fc.
                          3. nlm - inject faults in all eligible fault sites.

--fc FICOUNT              Fault injection countm this value is used in combination
                          with --ff.
```

6 Examples

6.1 Example 1 : Scalar Benchmark - Quicksort

6.1.1 Step 1: Compile

1. Change the directory to the source folder:

```
$ cd vulfi/benchmarks/scalar_benchmarks/src
```

2. Make sure that the paths provided in `common.mk` file are correct.

3. Change directory to qsort folder and compile the program:

```
$ cd qsort/
$ cd make
```

6.1.2 Step 2: Run

1. Change the directory to the run folder:

```
$ cd vulfi/benchmarks/scalar_benchmarks/run
```

2. Make sure that the paths provided in `common.mk` file are correct.

3. Change the directory to the qsort folder. The `common_qsort.mk` file contains the python command line used for performing the fault injection. The script file `run_qsort.sh` launches the execution accepting a single argument to denote the number of fault injection runs to be carried out.

```
$ cd qsort/
$ ./run_qsort.sh 100
```

6.1.3 Step 3: Result Analysis

1. **Fault Sites:** Refer to the files `addr_dbgData_qsort_addr.csv` and `ctrl_dbgData_qsort_ctrl.csv` located at `\vulfi\benchmarks\scalar_benchmarks\src\qsort`. These files list the fault sites chosen for runtime fault injection for `address` and `control` fault site categories respectively. You may notice that there are no `data` fault sites found for the qsort program.

2. **Fault Injection Result:** The fault injection result for each fault site category is saved in the directory `\vulfi\benchmarks\scalar_benchmarks\run\qsort\serial`. For example, each row in the file `result_qsort_ctrl.csv` correspond to a unique fault injection run during which a fault was injected. The result file provides useful information such as instruction and bit position chosen at runtime for the fault injection. It also categorizes the result of the fault injection into one of the categories - `sdc, benign, crash,` and `error`.

6.2 Example 2 : ISPC Benchmark - Raytracing

The steps to run `Raytracing` benchmark is similar to the previous example. However, this benchmark requires the ISPC compiler[4, 1] to be present on the machine. A precompiled version of the ISPC compiler which is known to work with the current version of VULFI could be downloaded from: `https://www.dropbox.com/s/5qjilxsqho14ell/ispc.tar.gz?dl=0`. Please make sure that you follow the license requirement of the ISPC compiler (`https://ispc.github.io/`) before downloading the precompiled binaries.

6.2.1 Step 1: Compile

```
$ cd vulfi/benchmarks/ispc_benchmarks/src/rt/
$ make
```

6.2.2 Step 2: Run

```
$ cd vulfi/benchmarks/ispc_benchmarks/run/rt/
$ ./run_rt.sh 5
```

6.2.3 Step 3: Result Analysis

Figures 1 and 2 are the outputs generated by the `Raytracing` benchmark for a fault-free and a faulty runs respectively.

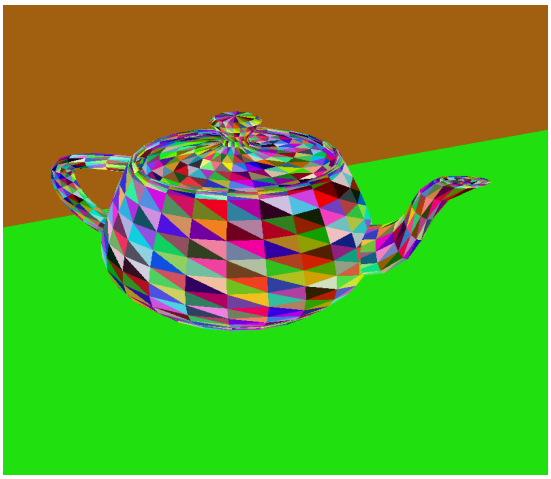

Figure 1: Golden Image

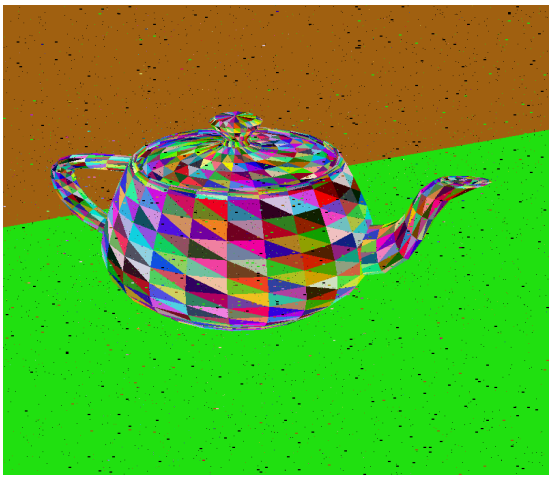

Figure 2: Corrupted Image

1. **Files listing fault sites:**

```
addr_dbgData_rt_avx2.csv
addr_dbgData_rt_sse4.csv
addr_dbgData_rt_ser.csv
ctrl_dbgData_rt_avx2.csv
ctrl_dbgData_rt_sse4.csv
ctrl_dbgData_rt_ser.csv
data_dbgData_rt_avx2.csv
data_dbgData_rt_sse4.csv
data_dbgData_rt_ser.csv
```

2. **Files with the fault injection results:**

```
result_rt_avx2_addr.csv
 result_rt_sse4_addr.csv
 result_rt_ser_addr.csv
 result_rt_avx2_ctrl.csv
 result_rt_sse4_ctrl.csv
 result_rt_ser_ctrl.csv
 result_rt_avx2_data.csv
 result_rt_sse4_data.csv
 result_rt_ser_data.csv
```

References

[1] Intel SPMD Program Compiler. `https://ispc.github.io/index.html`.

[2] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[3] The LLVM Compiler Infrastructure. `http://llvm.org/`.

[4] M. Pharr and W.R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.