COMPAS-FT-RISCV

Compiler Assisted Fault Tolerance for RISCV Architectures

Version 1.0.0

ESL Group, Chair of Electronic Design Automation
Department of Electrical and Computer Engineering
Technical University of Munich

December 16, 2021

Contents

1 User Manual **4**
1.1 Overview 4
1.2 Installation Guide 6
1.2.1 Development System 6
1.2.2 Build LLVM Project 6
1.2.3 Integrate Software Implemented Hardware Fault Tolerance (SIHFT) Passes 8
1.3 Usage Guide 9
1.3.1 SIHFT Compilation 9

List of Figures

1.1 Expected output of sanely built `llc` 7

Acronyms

SIHFT Software Implemented Hardware Fault Tolerance

User Manual

1.1 Overview

COMPAS-FT-RISCV compiler was developed as part of the research work conducted in SAFE4I project. SAFE4I is funded by the BMBF Ministry (Germany's Federal Ministry of Education and Research) under project grant 01IS17032. The authors are responsible for the content of this document.

Compiler transformations to build soft error resilience into embedded software are highly attractive owing to their superior error detection as well as automation properties. Before starting this work, we experienced a tooling gap with in the RISC-V eco-system for ensuring functional safety for embedded firmware. This work aims to fill this gap by providing a library of LLVM passes (C++ modules) that can be integrated in a modern LLVM codebase. The resulting compiler is then able to provide various state-of-the-art *Software Implemented Hardware Fault Tolerance* (SIHFT) transformations on a given C/C++ project.

Specifically, we have implemented the following SIHFT methods:

- EDDI [Oh et al., 2002b] for data-flow protection
- SWIFT [Reis et al., 2006] for data-flow protection
- NZDC [Didehban and Shrivastava, 2016] for data-flow protection
- NEMESIS [Didehban et al., 2017] for conditional branch protection
- CFCSS [Oh et al., 2002a] for control flow protection
- RASM [Vankeirsbilck et al., 2017] for control flow protection

- REPAIR [Sharif et al., 2021] for combined data-flow and control-flow protection

This document serves to be the user manual for the developed compiler. We now provide instructions on how to setup the LLVM compiler in order to use the implemented SIHFT transformations.

1.2 Installation Guide

In this section, we provide instructions on how to integrate our developed SIHFT passes within the LLVM codebase. This allows us later to carry out various SIHFT transformations during LLVM backend compilation in generating RISCV code.

1.2.1 Development System

The instructions given in this user manual are expected to work on any Linux machine. Specifically, we have prepared this manual while developing on a freshly installed Ubuntu 20.04 PC.

Ubuntu Packages

Following packages are needed to be installed:

- C++ compiler like gcc/g++
- CMake
- git

Following commands can be used to install them:

```
$ sudo apt update
$ sudo apt install build-essential cmake git
```

1.2.2 Build LLVM Project

The LLVM Project has to be compiled from source. We recommend downloading version 13.0.0 from LLVM Releases webpage:

`https://releases.llvm.org/`

With the downloaded source package extracted, execute the following commands on your bash shell to build llvm project from source:

```
$ cd <llvm_src_folder>
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release
        -DCMAKE_INSTALL_PREFIX=<install_location>
        -DCMAKE_CXX_STANDARD=17
        -DLLVM_ENABLE_PROJECTS=clang
        -DLLVM_TARGETS_TO_BUILD=RISCV
        -DLLVM_ENABLE_ASSERTIONS=ON
        ../llvm
$ make
```

NOTE: The paths mentioned above as <...> need to be adapted as per your filesystem.

In the `<build>/bin` directory, we should now see llvm executable programs and utilities like `clang`, `llc` etc. Run them to make sure they are built fine. For example, we can run the static compiler `llc` to make sure that RISCV targets are supported as shown in Fig. 1.1.

```
$ ./llc --version
LLVM (http://llvm.org/):
  LLVM version 13.0.0
  Optimized build with assertions.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    riscv32 - 32-bit RISC-V
    riscv64 - 64-bit RISC-V
$
```

Figure 1.1: Expected output of sanely built `llc`

NOTE: Make using parallel jobs (`make -jN`) would help significantly to reduce the compile time for a large codebase like LLVM. However, we encourage

to do this on a server. For systems having lesser memory, make with parallel jobs has to be used cautiously, as the command could fail due to memory exhaustion issues.

1.2.3 Integrate SIHFT Passes

The first step is to clone our open-source `compas-ft-riscv` repository within the LLVM source tree. Afterwards, the patch is applied on RISCV backend in order to register the new transformation passes. Finally, the LLVM project is rebuilt.
The corresponding shell commands are shown below. Here `<path_to_patch>` variable should be substituted with a compatible patch file. For this, we provide various patches in our passes repository. Each patch is compatible with a specific llvm project source tree. In case this tutorial is followed to build llvm project using Sec. 1.2.2, the corresponding patch is to be found at:

`compas-ft-riscv/patches/llvm13.0.0_src.patch`

```
$ cd <llvm_src_folder>/llvm/lib/Target/RISCV/
$ git clone https://github.com/tum-ei-eda/compas-ft-riscv
$ cd <llvm_src_folder>
$ patch -s -p0 < <path_to_patch>
$ cd build
$ make
$ make install
```

Here, at the end we install the built llvm compiler infrastructure. Again make sure that the compiler is installed fine by using Fig. 1.1.

1.3 Usage Guide

Various SIHFT passes can be invoked via command line arguments to the LLVM static compiler `llc` program. This chapter assumes that the LLVM compiler has been installed as per instructions in earlier sections. We refer the installation location of this compiler to `<llvm_install>` in rest of the chapter.

1.3.1 SIHFT Compilation

Each source file (translation unit) in C/C++ project has to be first compiled to LLVM IR code using `clang`. Following command can be used to generate human readable LLVM IR code representation of given C source file.

```
$ <llvm_install>/bin/clang
    -emit-llvm -S <other options>
    main.c -o main.ll
```

The next step in the compilation is to pass the LLVM IR file to `llc` program:

```
$ <llvm_install>/bin/llc
    <SIHFT options> <other options>
    main.ll -o main_hardened.s
```

`<SIHFT options>` refer to various SIHFT transformations that we have implemented. A brief overview of these options is provided in Table 1.1.

Table 1.1: Supported SIHFT options

Option	**Description**
-NZDC=foo,bar	apply NZDC transformation on *foo,bar* functions
-SWIFT=foo,bar	apply SWIFT transformation on *foo,bar* functions
-RASM=foo,bar	apply RASM transformation on *foo,bar* functions
-CFCSS=foo,bar	apply CFCSS transformation on *foo,bar* functions
-FGS	use fine-grain scheduling for NZDC code
-REPAIR	use REPAIR transformation on NZDC code

The final step in code generation is to use `gcc` for RISCV architectures to assemble and link the assembly code for a particular RISCV processor.

Example

For the sake of example, assume we want to protect the `crc` program from the MiBench suite. The compiled program is to run on the RISCV spike simulator. The C project contains the following C files:

- crc.h: header for CRC library functions
- crc.c: source for CRC library functions
- main.c: contains the `main` program to invoke crc library functions

Following commands are issued on bash shell to create the RISCV binary with

- NZDC, RASM protection on `crcSlow` function
- RASM protection on `crcFast` function
- NZDC, CFCSS protection on `main` function

```
$ cd <crc-project-source>

$ <clang> -emit-llvm -O2 -S --target=riscv64
    -march=rv64gc -mabi=lp64d
    -isystem <riscv64-gcc>/riscv64-unknown-elf/
    main.c -o main.ll
$ <clang> -emit-llvm -O2 -S --target=riscv64
    -march=rv64gc -mabi=lp64d
    -isystem <riscv64-gcc>/riscv64-unknown-elf/
    crc.c -o crc.ll

$ <llc> -O2 -march=riscv64 -mattr=+m,+a,+d,+c
    -NZDC=main -CFCSS=main
    main.ll -o main_hardened.s
$ <llc> -O2 -march=riscv64 -mattr=+m,+a,+d,+c
```

```
    -NZDC=crcSlow -RASM=crcSlow,crcFast
    crc.ll -o crc_hardened.s

$ <riscv64-gcc>/bin/riscv64-unknown-elf-gcc -O2
    -march=rv64gc -mabi=lp64d
    main_hardened.s crc_hardened.s -o crc_hardened.elf

$ spike pk crc_hardened.elf
```

Bibliography

[Didehban and Shrivastava, 2016] Didehban, M. and Shrivastava, A. (2016). Nzdc: A compiler technique for near zero silent data corruption. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, New York, NY, USA. Association for Computing Machinery.

[Didehban et al., 2017] Didehban, M., Shrivastava, A., and Lokam, S. R. D. (2017). Nemesis: A software approach for computing in presence of soft errors. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 297–304.

[Oh et al., 2002a] Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002a). Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122.

[Oh et al., 2002b] Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002b). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75.

[Reis et al., 2006] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. (2006). SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254. IEEE.

[Sharif et al., 2021] Sharif, U., Mueller-Gritschneder, D., and Schlichtmann, U. (2021). Repair: Control flow protection based on register pairing updates for sw-implemented hw fault tolerance. *ACM Trans. Embed. Comput. Syst.*, 20(5s).

[Vankeirsbilck et al., 2017] Vankeirsbilck, J., Penneman, N., Hallez, H., and Boydens, J. (2017). Random Additive Signature Monitoring for Control Flow Error Detection. *IEEE Transactions on Reliability*, 66(4):1178–1192.