Intel SIMICS Compiler Sanitizers

A collaborative project with Intel

Paweł Stzelczyk
Piotr Baryczkowski
Marcin Strzesak
Maciej Szefliński

Supervisor: mgr inz. Magdalena Sroczan

The Faculty of Computing and Telecommunications
Poznań University of Technology
January 2024

Contents

1 Intel SIMICS Compiler Sanitizers

The RISC-V Bare-Metal Sanitizer Integration Project focuses on enhancing the security and robustness of software applications running on RISC-V architectures by incorporating sanitization mechanisms. Key components of this project should include the application of sanitization techniques to RISC-V bare-metal environments, the development of a dedicated sanitizer backend, and seamless integration into GCC.

Key pieces:

1. Application to RISC-V bare-metal.
   - The project runs directly on hardware without operating system.
2. Sanitizer backend.
   - A dedicated software that implements sanitization mechanisms for detecting various types of software vulnerabilities, such as memory corruption, data races, and undefined behavior.
3. Integration with compiler.
   - Integration of the sanitizer into the backend of widely used compilers, specifically GCC. This integration ensures that sanitization processes become an integral part of the compilation pipeline.
4. Demo application for testing.
   - A demo application is provided to test effectiveness of the sanitizers. It includes intentionally broken code to trigger various types of vulnerabilities.

2 Getting Started

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes. See deployment for notes on how to deploy the project on a live system.

2.1 Prerequisites

What things you need to install the software and how to install them

- Intel® Simics® Simulator
- RISC-V GNU Compiler Toolchain

2.2 Installing

2.3 Simics

1. Install The Simics® simulator, ideally linux distribution Simics Download.
2. Create a new project.
3. Copy to project's targets the risc-v-baremetal directory.
4. Launch it in simics to confirm correct configuration.

2.4 Riscv-gnu-toolchain

1. Get riscv-gnu-tollchain GitHub.
2. Configure the toolchain build options.

   ```
   ./configure --prefix=/opt/riscv --with-cmodel=medany --with-arch=rv64gc --with-abi=lp64
   ```
3. Build the toolchain.

   ```
   make -j $(nproc)
   ```

4. Recompile snake game example located in *risc-v-baremetal/images/src* (you may remove build options for 32bit binaries).

5. Move built binary to the source directory.

6. Launch risc-v-baremetal example again to confirm correct toolchain configuration.

7. Try to add some code to e.g. snake.c file that uses functions from libc, e.g. *malloc()*.

8. Test if building the game works.

Remember to add */opt/riscv/bin* to the PATH variable and to set CROSS_PREFIX (used in Makefile) to 'riscv-unknown-elf-' or use *export.sh* script to build the application and to move binary file to target location.

3 Deployment

To launch the project on risc-v bare-metal platform please follow these instructions:

1. Go to the project that contains bare-metal target plaform.

2. Run:

   ```
   ./simics
   ```

3. Then, in launched simics console, load the target:

   ```
   simics> load-target target="risc-v-baremetal/bare-metal"
   ```

4. And run it:

   ```
   simics> r
   ```

5. To close simulation, run:

   ```
   simics> q
   ```

A serial console should open in which you should see output from the application.

4 Report

The following report describes the progress that was made during the implementation project for Intel. This document is divided into seven sections: Toolchain, libc, Makefile, Linker script, Sanitizer, UBSAN, malloc. Some additional information that could help to continue the project is also included.

4.1 Riscv-gnu-toolchain

The RISC-V GNU Toolchain is a set of programming tools and utilities designed to support the RISC-V instruction set architecture. This toolchain includes a cross-compiler, which means it can generate executable code for RISC-V architectures on a host system with a different architecture. This allows us to build software for RISC-V based systems without the need for specialized hardware. It also supports bare-metal applications, which run directly on the hardware without an underlying operating system.

For the purpose of this project, the toolchain has to be configured to support 64-bit RISC-V bare-metal platform. Proper configuration is described in project's README.

4.2 libc

The C standard library, often referred to as libc, is a fundamental component of software development, providing a collection of pre-written code that offers standard functionality to C programs. However, when it comes to bare-metal programming, where software runs directly on hardware without an operating system, the libc implementation becomes a critical consideration.

In the context of bare-metal programming, developers often use a minimalistic version of libc tailored for their specific target platform. This lightweight libc implementation is designed to provide essential functions and features without relying on an underlying operating system. The primary purpose of this bare-metal libc is to offer basic runtime support for C programs in an environment where traditional operating system services are absent.

There are some lightweight libc implementations such as picolibc but due to license restrictions we decided to use libc (Newlib) provided with riscv-gnu-toolchain.

Bare-metal projects typically need to handle low-level details such as memory management, hardware initialization, and interrupt handling. A bare-metal libc may include implementations of essential functions like memory allocation, input/output operations, and basic math functions, all customized for the specific hardware architecture in our case 64-bit version of RISC-V.

We were able to compile a sample application that used functions from the C standard library. Some of them require subroutines that are not present in a bare-metal environment. The steps to be taken to provide definitions for OS interface are described in newlib documentation.

4.3 Makefile

This project's makefile is responsible for building a binary that runs on RISC-V bare-metal platform. CROSS_PREFIX parameter should be set to *CROSS_PREFIX=riscv64-unknown-elf-* when using riscv-gnu-toolchain. *-static* compiler flag is important because it allows to link C standard library.

```
CC=$(CROSS_PREFIX)gcc

CFLAGS=-v -Og -Wall -std=c99 -g -mcmodel=medany -static -nostartfiles
LFLAGS=-Tlink.ld

all: snake64.elf

snake64.elf: snake.c uart.c start.S
    $(CC) $(CFLAGS) $(LFLAGS) -march=rv64gc -mabi=lp64 -o $@ $^
```

4.4 Linker script

Linker scripts are configuration files used by the linker during the process of building an executable or a shared library from multiple object files. The linker script defines the layout and attributes of the sections in the final binary, specifying how different parts of the code and data are organized in memory. Linker scripts are particularly important in embedded systems and other scenarios where precise control over memory layout is crucial.

Linker scripts organize the various sections of code and data generated during compilation into specific memory regions. Sections include .text for code, .data for initialized data, .bss for uninitialized data, and others.

In embedded systems programming, linker scripts are frequently used to tailor the memory layout to the specific requirements of the target hardware. This may involve placing certain sections in specific memory regions or defining custom memory regions for peripherals or external devices.

4.5 Sanitizers on bare-metal

The purpose of sanitizers is to identify various issues like memory errors, data races, and undefined behavior. While they are commonly associated with operating system-based applications, the application of sanitizers in a bare-metal programming environment poses unique challenges and considerations.

In the context of bare-metal development, where software runs directly on hardware without an operating system, the absence of certain runtime services complicates the use of traditional sanitizers. However, efforts have been made to adapt and implement sanitizers for bare-metal environments, allowing developers to benefit from runtime analysis and bug detection.

However, it's crucial to acknowledge that the application of sanitizers in bare-metal programming may require careful configuration and customization to suit the target platform's constraints and characteristics. We've come across some limitations imposed by the absence of an operating system and tailor the sanitizer usage accordingly. Sanitizer are available in the riscv-gnu-toolchain but running them on bare-metal might need further configuration.

4.6 UBSAN

UBSAN, short for Undefined Behavior Sanitizer, is a dynamic code analysis tool that is part of the GCC compiler infrastructure. UBSAN is designed to detect and report undefined behavior in C and C++ programs at runtime. Undefined behavior in a program refers to situations where the language specifications do not define the expected behavior, leaving the outcome to be unpredictable.

UBSAN aims to identify undefined behavior in a program by instrumenting the code during compilation. It inserts runtime checks to detect issues such as accessing uninitialized variables, integer overflows, division by zero, and other constructs that lead to undefined behavior according to the language standards.

UBSAN can catch a variety of common programming mistakes, including pointer-related issues, type mismatches, and violations of language standards. This helps developers identify potential sources of bugs early in the development process.

In order to check a application with UBSAN, it should be compiled and linked with *-fsanitize=undefined*.

4.7 malloc

malloc is a function in the C programming language, used for dynamic memory allocation. It uses systems calls like mmap (in modern systems) or brk/sbrk to allocate memory.

Sbrk is a system call that adjusts the program's data space, effectively increasing or decreasing the amount of memory available to the application. Its usage typically involves incrementing the program's data space by a specified number of bytes and returning the previous end of the data space (the "break" point).

Brk is another system call related to memory management. It is used to set the end of the data segment of the calling process to the specified value. By adjusting the program break with brk, the heap could be expanded or contracted, providing the necessary memory for dynamic allocation.

These two system calls use references to bss section (section for uninitialized data) defined in linker script. Further attempts may be made to integrate these calls with linker script.

5 Further works

On the basis of studies and experiments carried out, we can draw the following further steps:

- finding out how to get 'malloc()' to work in bare-metal environment, following our approach (libc implementation) or finding different ways (e.g. custom 'malloc' implementation),
- implementation of test application that exhibits undefined behaviour - we recommend approach related to memory management,
- testing the functioning of code sanitizers - either provided with the toolchain or custom made.

6 Knowledge

During the project we learned to work together as a team on a project supervised by a person from big-tech company. The theme of our project was primarily to test the possibility of utilizing code sanitizers on bare-metal platforms. At the start of the work, we were informed that no one from the company had tried to do such things before. For this reason, our work was mainly research and experimental. Above all, we learned to seek information about the technology we were developing, especially using technical documentation. An important aspect was to consult our findings with a mentor, who shared his thoughts, knowledge and experience.

In terms of technical aspects, we learned, among other things, how memory management works in environments without an operating system. We also acquired knowledge of the dependencies of some code sanitizers. We learned about the purpose and characteristics of linker scripts. We had the opportunity to use the Intel Simics simulator during the project and learned how to operate it, as well as the basics of defining hardware platforms.