1. Введение

Бинарные трансляторы являются инструментами для эмуляции, бинарного анализа и профиляции, важным классом которых являются динамические бинарные трансляторы, которые преобразуют двоичный ход на лету и генерируют нужные инструкции по мере требования. Примером такого транслятора является Rosetta 2 от Apple. Часто, исходный двоичный код преобразуется в какое-либо промежуточное представление, где такое преобразование осуществляет лифтер. Примером промежуточного представления может быть TCG для QEMU. Но в качестве промежуточного представления хочется использовать LLVM IR, так как тулчейн LLVM может делать качественные оптимизации кода. В данной работе хотим рассмотреть динамические бинарные трансляторы с поднятием двоичного кода в LLVM IR для RISC-V.

2. Обзор

Для начала необходимо рассмотреть такие канонические ДБТ, как

- QEMU-user, который является стандартом эмуляции userland программ;
- Rosetta 2, который является эмулятором с закрытым исходным кодом от Apple;
- Valgrind, который является инструментом динамического бинарного анализа в состав которого входит Memcheck и Callgrind.

Также довольно популярным является Box64, который позволяет запускать программы, в частности игры, скомпилированные под linux x86/x64, на ARM64 и RISC-V. Важно упомянуть, что Box64 нацелен на производительность. Из ДБТ с поднятием в LLVM IR, можно выделить

- Instrew в связке с лифтером Rellume;
- BinRec (не обновлялся с 2022го года);
- HQEMU, который использует два промежуточных представления: TCG и LLVM IR (не обновлялся с 2018го года);
- DBILL, LnQ, Lasagne (есть только статьи, либо код закрыт).

В результате обзора оказалось, что Instrew + Rellume производительнее, чем QEMU и Valgrind, благодаря оптимизациям

LLVM (на основе бенчмарков представленных в кандидатской). Также, в сравнение с QEMU, Valgrind и Box64, является не только эмулятором, но и инструментом для динамического бинарного анализа (что позволяет менять семантику транслируемых программ), так как из коробки имеет API для добавления анализаторов, например, подсчет инструкций. По этим причинам, неплохо было бы портировать Instrew на RISC-V. Также в сравнение с другими ДБТ с поднятием в LLVM IR поддерживается автором и не заброшен, а также имеет поддержку архитектуры RISC-V, как целевой.

3. Постановка задачи

Целью работы является добавление поддержки host-архитектуры RISC-V. Для ее выполнения были поставлены следующие задачи:

- выполнить обзор архитектуры Instrew;
- реализовать минимальную функциональность для запуска Instrew на RISC-V, а именно:
  - скомпилировать, добавив процессорно-специфические патчи и проставив константы (макросы riscv, размеры функций, адреса и номера и аргументы системных вызовов);
  - реализовать функции, связанные с RISC-V ABI (эмуляция системных вызовов, трамплины PLT таблицы, релокации).
- протестировать на простых примерах.

4. Архитектура Instrew

Instrew реализует архитектуру клиент-сервер. Клиент написан на языке Си и из его основных функций можно выделить:

- отправка функций транслируемой программы серверу;
- получение от сервера ELF объекта, проведение релокаций, разрешение символов и эмуляция системных вызовов.

Таким образом клиент выполняет диспетчеризацию исходных инструкций и является динамическим линкером. Также из деталей реализации важно отметить, что для клиента написана своя реализация libc (так называемый minilibc).

Сервер написан на C++ с помощью LLVM, и его основными функциями являются:

- поднятие двоичного кода в LLVM IR с помощью Rellume;
- применение оптимизаций и кодогенерация.

В данной практике была проведена работа именно над клиентской частью Instrew.

5. Реализация (1/2)

Первый шаг реализации — это скомпилировать Instrew. Для этого необходимо проставить константы, а именно макросы RISC-V, размеры функций, адреса, номера и аргументы системных вызовов. Далее необходимо написать на ассемблере точку входа в программу в нашем minilibc (в glibc — это файл start.S, в musl — crt_arch.S). Из сложностей можно отметить, что при написании точки входа был просмотрен код канонических реализаций libc и там была обнаружена плохо задокументированная псевдоинструкция, которая, грубо говоря, не понятно, что делала, и при подставлении которой в свой код, все магически работало.

6. Особенности Instrew (1/2)

После того, как мы успешно написали точку входа в программу, необходимо попробовать запустить клиент Instrew без аргументов. И здесь хочется рассказать, почему Instrew сложно дебажить:

- Во-первых, из-за специфических флагов сборки. Пытаемся запустить программу — получаем ошибку сегментации. Asan и gprof здесь не работают:
  - первый — по причине того, что система сборки автоматически его выключает;
  - второй — потому что при попытке поставить флаг -pg, система сборки выдает ошибку — конфиликтующие флаги.

  После локализации проблемы, оказалось, что динамически подключался линкер ld-linux, а такого быть не должено. Оказалось проблема в том, что –static-pie флаг в gcc одной и той же версии 13, интерпретируется по-разному на архитектурах x86_64 и RISC-V.

- Во-вторых, из-за атрибутов и релокаций. Так как клиент Instrew самостоятельно разрешает свои релокации, было неочевидно, почему программа выходит с ошибкой. После проверки правильности подсчета всех релокаций, оказалось, что есть функция с атрибутом, для которой тип релокации отличается от всех остальных функций (или про-

сто не матчится). То есть атрибут меняет тип релокации, что было неочевидно.

- В-третьих, сервер запускает клиент с помощью fork() и fexecve(). Поэтому когда пытаемся запустить клиент Instrew в gdb, ничего не выходит. Точку остановки поставить нельзя, символы не подгружаются. Спустя некоторое время удалось научиться запускать Instrew сервер с правильными дебаг флагами, чтобы можно было запустить gdb на клиенте.

- Вывод:

  – из-за некоторых флагов сборки и линковки, а также особенностей реализации сложно дебажить Instrew;

  – по сути из инструментов есть только printf, gdb и Valgrind.