DOI: 10.15514/ISPRAS-2019-31(5)-1

Автоматизированное тестирование фронтенда транслятора TCG для Qemu

[1]Д.С. Колтунов, ORCID: 0000-0003-4556-9463 <koltunov@ispras.ru>
[1]В.Ю. Ефимов, ORCID: 0000-0003-3433-6787 <real@ispras.ru>
[1,2]В.А. Падарян, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>

[1]Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2]Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Реализация новой виртуальной процессорной архитектуры в Qemu предполагает создание фронтенда динамического двоичного транслятора TCG для данной процессорной архитектуры. Существующие на сегодняшний день системы тестирования фронтенда TCG используют подход на основе сравнения с эталоном той же процессорной архитектуры. В качестве эталона могут выступать реальный процессор, виртуальная машина с большей точностью эмуляции или другая реализация двоичного транслятора. Однако не всегда такие эталоны доступны, зачастую они могут вообще не существовать. Данная работа нацелена на тестирование реализации процессорной архитектуры в Qemu в условиях отсутствия необходимого эталона для сравнения. Предлагаемый подход основывается на том, что даже для малораспространённой процессорной архитектуры, как правило, доступен пакет `binutils` и компилятор языка Си. Си-программа может одинаково выполняться на различных процессорных архитектурах, если удаётся избегать в ней ситуаций неопределенного или реализационно зависимого поведения. Это позволяет проводить сравнение хода работы двух разных исполняемых файлов на тестируемой виртуальной машине и машине разработчика. Объектами сравнения такого подхода выступают сущности языка Си, на котором разрабатываются тесты. Подход реализован в программном средстве c2t (CPU Testing Tool) и входит в состав программного комплекса автоматизации разработки моделей устройств и вычислительных машин для Qemu, исходный код которого доступен по адресу https://github.com/ispras/qdt. c2t реализовано на языке программирования Python, поддерживает тестирование Qemu в режиме полносистемной эмуляции и в режиме эмуляции уровня пользователя. Данное средство пригодно как для тестирования фронтендов TCG, полученных с использованием системы автоматизации создания фронтендов TCG, так и реализованных классическим способом (вручную).

Ключевые слова: Qemu; автоматизированное тестирование фронтенда TCG; QDT; GDB RSP

Для цитирования: Колтунов Д.С., Ефимов В.Ю., Падарян В.А. Автоматизированное тестирование фронтенда транслятора TCG для Qemu. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 7–24. DOI: 10.15514/ISPRAS–2019–31(5)–1

Благодарности: Работа поддержана грантом РФФИ № 16-29-09632

Automated testing of a TCG frontend for Qemu

[1]D.S. Koltunov, ORCID: 0000-0003-4556-9463 <koltunov@ispras.ru>
[1]V.Y. Efimov, ORCID: 0000-0003-3433-6787 <real@ispras.ru>
[1,2]V.A. Padaryan, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>

[1]Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2]Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation

Abstract. Implementing a new target architecture in Qemu involves creation of a dynamic binary translator TCG front-end for that architecture. Testing is necessary to verify correctness of that translator component. Currently, existing TCG front-end testing systems use an approach based on a comparison with an oracle. Such oracle have the same processor architecture. And an oracle may be a real processor, a high-fidelity emulator or another binary translator. Unfortunately, such oracles are not always available. This paper is devoted to testing a target architecture implementation in Qemu when the necessary oracle is not available. The main idea is following. There is observation, a program written in a high-level programming language is expected to execute equally regardless of processor architecture. In other words, one can use a real processor with a different architecture for comparison. In this paper, it is the processor of a developer AMD64 machine. The comparison objects are the term of a high-level programming language. I.e. tests are written in C. C language was chosen for this purpose, because, on the one hand, it is fairly close to the hardware, and, on the other, it has compilers for many processor architectures. The approach is implemented in CPU Testing Tool (c2t) which is part of QDT. Source code is available at https://github.com/ispras/qdt. The tool is implemented in Python programming language and supports testing of Qemu in both full system and user level emulation modes. c2t is suitable for testing TCG front-ends which are generated by the automatic TCG front-end generation system or implemented in the classical way (manually).

Keywords: Qemu; automated testing of a TCG frontend; QDT; GDB RSP

For citation: Koltunov D.S., Efimov V.Y., Padaryan V.A. Automated testing of a TCG frontend for Qemu. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 5, 2019 г., pp. 7-24 (in Russian). DOI: 10.15514/ISPRAS–2019–31(5)–1

Acknowledgements. This work was supported by RFBR grant № 16-29-09632

1. Введение

Виртуальные вычислительные машины применяются для решения разнообразных задач, включая исследования в рамках информационной безопасности. Одной из таких задач является динамический анализ, который нуждается в контролируемой среде выполнения исследуемого машинного кода. Удобный и распространённый способ реализации контролируемой среды – программный эмулятор, поскольку он обеспечивает изоляцию средств анализа от анализируемого кода. На практике такой способ хорошо подходит для исследования компьютерных вирусов и другого вредоносного ПО.

Эмулятор Qemu наиболее удобен для этой цели, поскольку обладает рядом полезных свойств: полностью открытый исходный код (лицензия GPL), поддержка разнообразных гостевых архитектур (Intel x86, AMD 64, ARM, MIPS, PowerPC, SPARC и др.), реализация важных, с точки зрения динамического анализа, технологий и возможностей [1].

Тем не менее, при необходимости провести динамический анализ для малораспространенной процессорной архитектуры, скорее всего, аналитик столкнется с тем, что готовой виртуальной машины найти не удастся. В этом случае придётся решать задачу разработки виртуальной машины, в которую входит большая подзадача реализации процессорной архитектуры.

Реализация новой процессорной архитектуры в Qemu, кроме прочего, предполагает создание фронтенда (англ. frontend) динамического двоичного транслятора TCG для данной процессорной архитектуры. Как правило, фронтенды TCG создаются вручную, но на данный момент уже есть средства, предлагающие некоторую автоматизацию [2]. В любом случае для проверки корректности фронтенда транслятора необходимо тестирование. Оно является неотъемлемым этапом разработки ПО, который своевременно сокращает количество допущенных программистом ошибок, что, в свою очередь, повышает точность эмуляции и качество анализа исследуемого кода.

Самым распространённым способом тестирования реализации процессорной архитектуры в Qemu является сравнение с эталоном («test oracle») [3]. Чтобы провести тестирование таким способом необходима доступность эталона, а также возможность получать от него информацию о состоянии выполняющейся на нем программы. Существующие на сегодняшний день системы тестирования фронтенда TCG используют в качестве эталона реальный процессор, виртуальную машину с большей точностью эмуляции или альтернативную реализацию двоичного транслятора. Оценивается корректность работы отдельной инструкции, которая выполняется в тестируемом и эталонном окружениях, подготовленных специальным образом. После ее выполнения сравниваются состояния заданных регистров процессора и содержимого памяти в эмуляторе и эталоне.

Однако не всегда есть возможность работы с эталоном. Это обусловлено недоступностью необходимого реального оборудования, отсутствием у оборудования отладочных интерфейсов, недоступностью альтернативной виртуальной машины или двоичного транслятора нужной процессорной архитектуры.

Целью данной работы является разработка подхода к тестированию реализации процессорной архитектуры в Qemu в условиях отсутствия необходимого эталона для сравнения и последующая реализация соответствующего программного средства, позволяющего проводить автоматизированное тестирование.

Предлагаемый подход основывается на том, что даже для малораспространенной процессорной архитектуры будет доступен некоторый набор программных инструментов разработки: пакет `binutils` или его аналог, компилятор языка Си. Располагая такими инструментами можно проводить тестирование, не опускаясь на уровень отдельных команд, а выполняя тестовые Си-программы и сравнивая их поведение в терминах языка Си –на уровне значений переменных. Один и тот же тест компилируется для двух различных машин: Qemu (тестируемая процессорная архитектура) и машины (используемой в качестве эталона), на которой ведётся разработка.

В данной работе сделано следующее:

- разработан набор тестов;
- разработан способ оценки достигнутого покрытия кода;
- реализовано средство автоматизации тестирования;
- проведены эксперименты.

2. Обзор родственных работ

Существующие на сегодняшний день системы тестирования реализации процессорной архитектуры в Qemu основываются на сравнении с эталоном данной процессорной архитектуры. Они делятся на следующие типы.

- Системы, которые направлены на обнаружение неточностей реализации процессорной архитектуры x86 с целью повысить прозрачность с точки зрения динамического анализа. В качестве эталона для сравнения выступают реальный

процессор или виртуальная машина с большей точностью эмуляции. Такие системы описаны в работах EmuFuzzer [4], KEmuFuzzer [5] и PokeEMU [6].
- Системы генерации тестов для тестирования реализации не только процессорной архитектуры x86. Эталоном является реальный процессор. К данному типу относится системы RISU [7] и MicroTESK [8].
- Системы, использующие в качестве эталона для сравнения другую реализацию двоичного транслятора. Такой системой является MeanDiff [9].

Далее кратко будет рассмотрена каждая из этих систем.

2.1 EmuFuzzer, KEmuFuzzer и PokeEMU

EmuFuzzer –прототип, в котором реализована полностью автоматизированная методика тестирования для эмуляторов процессора (Qemu, Valgrind, Pin и BOCHS), основанная на фаззинге. Данная методика может быть использована для автоматического обнаружения расхождений конфигурации среды (т.е. состояние регистров процессора и содержимое памяти) в эмулируемом и физическом процессорах. Для тестирования эмулятора создаётся большое количество тестов, которые запускаются на эмулируемом и реальном процессорах. В конце выполнения каждого теста сравниваются конфигурации двух сред. Любое расхождение является признаком неправильного поведения эмулятора.

KEmuFuzzer –прототип, который реализует автоматизированную методику (как и EmuFuzzer) для тестирования четырёх современных виртуальных машин: BOCHS, Qemu, VirtualBox и VMware. Отличие от EmuFuzzer в том, что тестируются полносистемные эмуляторы, и что в качестве эталона выступает KVM.

PokeEMU –инструмент тестирования эмулятора. Он автоматически генерирует наборы тестов с большим покрытием и сравнивает поведение эмулятора с реальной машиной, выполняя тесты на обоих из них. Инструмент генерирует тестовые сценарии с помощью символьного выполнения. Данным инструментом был протестирован Qemu. PokeEMU также использует в качестве эталона KVM.

Имеются ещё работы подобного типа [10] и [11].

2.2 RISU и MicroTESK

Инструмент RISU (Random Instruction Sequences for Userspace) –инструмент для проверки точности реализации набора инструкций процессора в таких виртуальных машинах, как Valgrind и Qemu. Он состоит из двух частей.

- Генератор, который выводит случайный машинный код на основе входного файла, описывающего шаблоны набора инструкций.
- Тестовая программа, которая запускает сгенерированный код, как на тестируемом, так и на реальном оборудовании.

Инструмент может использоваться для тестирования эмуляторов, предназначенных для запуска пользовательских приложений, например, Valgrind и Qemu linux-user. Для архитектуры ARM RISU поддерживает тестирование 32-битных наборов команд A32 (ARM) и T32 (Thumb), а также 64-битного набора команд A64. Инструмент также поддерживает архитектуры PPC и m68k.

Инструмент MicroTESK (Microprocessor TEsting and Specification Kit) –среда генерации тестовых программ на языке ассемблера целевой процессорной архитектуры для функциональной верификации микропроцессоров. Также данный инструмент предоставляет возможность автоматизированного конструирования генераторов тестов на основе формальных спецификаций необходимой процессорной архитектуры. Такие

тесты можно запускать в Qemu и на реальном процессоре с целью выявления расхождений в их поведении.

2.3 MeanDiff

MeanDiff –инструмент тестирования промежуточных представлений для двоичных трансляторов. Он реализует идею поиска семантических ошибок во фронтендах (binary lifters) существующих двоичных трансляторов.

Подход к тестированию основывается на сравнении семантики промежуточных представлений трансляторов. Разработано некоторое специальное унифицированное промежуточное представление, которое используется для единообразного описания семантики разных промежуточных представлений.

Процесс тестирования заключается в том, что на вход сравниваемым двоичным трансляторам подаётся некоторый машинный код. Этот код переводится трансляторами в своё промежуточное представление. Затем все полученные промежуточные представления транслируются в унифицированное промежуточное представление. После этого результаты сравниваются между собой, и производится проверка на наличие семантических расхождений. Данным инструментом можно протестировать правильность реализаций фронтендов двоичных трансляторов, которые имеются в Qemu, Valgrind, BINSEC и т.д.

2.4 Выводы

В рассмотренных работах при тестировании реализации процессорной архитектуры в Qemu используются разные виды эталонов: реальный процессор, виртуальная машина с большей точностью эмуляции, а также другая реализация двоичного динамического транслятора. Но все они той же процессорной архитектуры, что и тестируемая реализация. Однако не всегда есть доступ к таким эталонам. Для решения этой проблемы в данной работе предлагается подход, опирающийся на альтернативный, но всегда доступный эталон.

3. Предлагаемый подход к тестированию

Предлагаемый в данной работе подход к тестированию фронтендов TCG различных процессорных, подобно другим подходам, использует сравнение с эталоном. Но эталоном будет выступать процессор *любой* архитектуры (к какому-нибудь имеется доступ всегда). Главная проблема заключается в том, у процессора другой архитектуры отличаются регистры (длина, номенклатура, количество, и пр.), варианты типовых команд, длина адреса и прочие архитектурные особенности. Но обычно процессоры используются для выполнения алгоритмов, логика которых не привязана к конкретной архитектуре. Например, вычисление арифметического среднего, сортировка, поиск в массиве и т.д.

Независимо от архитектуры процессора такой алгоритм на каждом своём шаге должен находиться в состоянии, зависящем только от входных данных. Суть предлагаемого подхода заключается в повышения уровня представления состояния регистров процессора и памяти до абстрактных понятий, используемых в алгоритме. Затем предлагается сравнивать путь выполнения и состояния высокоуровневых абстракций (таких как переменные).

Этого предлагается достичь за счет написания тестовых сценариев на языке программирования высокого уровня, что дает следующие возможности для тестирования.

- Возможность использовать в качестве эталона для сравнения –персональный компьютер.

- Повторное использование тестов. Уход от машинной зависимости тестов, которые в существующих решениях пишутся на языке ассемблера или генерируются в виде машинного кода, предоставляет возможность использовать один раз написанный тест для тестирования фронтендов TCG различных архитектур.

Очевидно, что у данного подхода есть много ограничений, которые рассмотрены ниже. Тем не менее показано, что так можно протестировать значительную часть реализации фронтенда.

Наиболее подходящим для решения данной задачи является использование высокоуровневого языка программирования Си.

Объектами сравнения являются значения переменных и номера строк программы. Стоит отметить, что для проверки корректности функционирования фронтенда TCG сравнение значений переменных и позиции выполнения является практически достаточным для большинства случаев. Архитектурные особенности основной и целевой системы отличаются, но логика работы с переменными у них одинаковая. Немаловажным является то, что предложенный подход к тестированию автоматизируется.

Таким образом, предлагаемый подход заключается в компиляции программы (теста), написанной на языке Си, под тестируемую архитектуру и архитектуру основной машины с последующим запуском в Qemu и ОС пользователя под контролем отладчика.

3.1 Ограничения и тонкости

В стандарте языка Си говорится о ситуациях, когда поведение некоторых конструкций языка может быть различным в зависимости от платформы и реализации компилятора (undefined behaviour и implementation-defined behavior). Если аккуратно избежать попадания в данные ситуации, то скомпилированная программа на Си должна одинаково выполняться независимо от процессорной архитектуры и компилятора. При этом будем считать, что процессор и компилятор реализованы без ошибок, так как вероятность наличия в них ошибок много меньше, чем в тестируемой реализации процессорной архитектуры.

При составлении тестовых программ необходимо учитывать и другие тонкости. Для того, чтобы компилятор не устранил переменную как избыточную при оптимизации, нужно использовать для данной переменной квалификатор типа `volatile`. Другим примером является то, что часто при программировании микроконтроллеров тип `int` языка Си имеет диапазон значений (размер), который отличается от того диапазона, к которому привыкли Си-программисты под IA32 и AMD64. В данном случае рекомендуется использовать подходящий тип из `<stdint.h>`. Например, `int32_t` или `int16_t` – в зависимости от того, какой диапазон значений на самом деле нужен. При этом, явное указание диапазона значений переменной может влиять на выбор компилятором определённых вариантов инструкций, а также поможет проверить правильность реализации в эмуляторе поведения при переполнении.

Не всегда возможно добиться от компилятора использования некоторых инструкций. В некоторых случаях от разработчика тестов требуется определённая изобретательность. В крайнем случае, интересующую инструкцию можно внедрить ассемблерной вставкой, а при помощи препроцессора добиться, чтобы для эталона вместо этой инструкции выполнялся эквивалентный Си-код. Другими словами, подход не исключает классический способ тестирования с написанием тестов на ассемблере. Он позволяет сэкономить время в тех случаях, когда требуемого покрытия можно добиться, используя Си. Такая гибридизация является предметом дальнейших исследований.

Кроме того, процессоры часто обладают инструкциями, семантика которых не описывается языком Си. Например, инструкция остановки процессора до следующего

аппаратного прерывания (`hlt` в i386) или векторные инструкции (расширение SSE для x86). Тестирование таких инструкций выходит за рамки данной работы.

Таким образом, подход главным образом ориентирован на проверку правильности реализации:

- арифметики – сложение, вычитание, умножение, деление;
- побитовых операций – умножение, сложение, сложение по модулю 2, инверсия, сдвиги;
- обработки данных и операций с памятью – операции с регистрами, стеком и т.д.;
- операций изменения потока управления – вызовы подпрограмм, условная и безусловная передача управления, возврат управления.

Создание тестов, которые обеспечивают наибольшее покрытие по инструкциям, реализованных в тестируемом фронтенде TCG, является отдельной сложной задачей и выходит за рамки данной работы. Однако, ряд показательных тестов всё-таки был реализован для оценки подхода.

4. Реализация автоматизированного тестирования

Разработанный подход был реализован в программном средстве автоматизированного тестирования фронтендов TCG –c2t (CPU Testing Tool), которое входит в состав программного комплекса автоматизации разработки моделей устройств и вычислительных машин для Qemu [1]. c2t написан на языке Python. Исходный код QDT (в том числе и c2t) является открытым.

Подход предполагает использование отладчика для сравнения значений переменных и пути выполнения (номеров строк).

Получение состояний переменных обеспечивается с помощью модуля отладки. Помимо контроля за выполнением отладчик интерпретирует низкоуровневые данные в терминах языка Си, так как эмулятор и эталон оперируют понятиями памяти и регистра процессора. Подробно об этом рассказано выше.

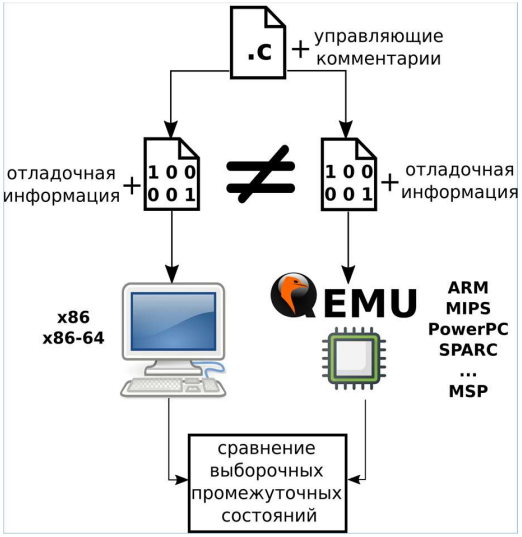

Рис. 1. Схема осуществления тестирования
Fig. 1. Testing approach

При сравнении пути выполнения программы на каждом шаге сравнивается не счётчик команд, как это происходит в EmuFuzzer и KEmuFuzzer, а позиция в коде тестового сценария (номер строки). Разумеется, ни эмулятор, ни эталон не могут предоставить такую высокоуровневую информацию: им доступно только значение своего счётчика команд. Поэтому тут тоже задействован модуль отладки.

Так как сравнение состояний всех переменных после выполнения каждой строки теста в ходе проведения тестирования является избыточным, то необходим механизм указания определённых строк и имён переменных для сравнения. Другими словами: механизм тонкой настройки тестирования. Данный механизм реализован путём добавления специальных управляющих комментариев напротив необходимых строк исходного кода теста (см. рис. 1).

Сам факт наличия управляющего комментария к строке означает, что на данной строке будет приостановлено выполнение теста эталоном и тестируемой виртуальной машиной, и будут получены состояния переменных для дальнейшего сравнения. Другими словами, комментарии указывают c2t те места, на которые необходимо поставить точки останова, а также имена переменных, значения которых необходимо проверить.

В Qemu реализован сервер внешней отладки гостевого кода по протоколу удалённой отладки GDB (GDB Remote Serial Protocol), основными возможностями которого являются:

- чтение/запись значений регистров процессора;
- чтение/запись в оперативную память;
- установка точек останова по управлению (breakpoint);
- установка точек останова по обращению к памяти (watchpoint).

Это означает, что для отладки гостевого кода может быть использован отладчик, поддерживающий данный протокол. Например, отладчик GDB. Однако использование отладчика GDB для проведения тестирования фронтенда Qemu усложняет процесс подготовки к проведению тестирования. Это обусловлено высокой трудоёмкостью процесса модификации отладчика GDB, как и любого другого отладчика, ориентированного на взаимодействие с человеком и не являющегося программной библиотекой. Таким образом, необходимо использовать средство отладки, предоставляющее достаточную гибкость.

В качестве решения данной проблемы выступает отладочный API QDT. QDT является инструментом автоматизации разработки моделей устройств и вычислительных машин для Qemu, написанный на языке программирования Python. Отладочный API QDT использует модуль «pyrsp» [12] с открытым исходным кодом, также написанный на языке Python. Модуль «pyrsp» реализует API для взаимодействия по протоколу RSP. Также отладочный API QDT использует модуль «pyelftools» [13] для работы с отладочной информацией в формате DWARF. На основе данного API и реализовано программное средство автоматизированного тестирования фронтенда транслятора Qemu: c2t.

Средство c2t поддерживает тестирование Qemu в режиме полносистемной эмуляции и в режиме эмуляции уровня пользователя. Обычно при тестировании TCG виртуальные устройства не представляют интереса. Более того, их деятельность может мешать. Т.е. поддержки эмуляции уровня пользователя обычно бывает достаточно. Но бывают случаи, когда для работы алгоритма высокого уровня нужно некоторое устройство.

Например, в микроконтроллерах может отсутствовать инструкция умножения. Вместо неё присутствует устройство-умножитель, не входящее в АЛУ процессора, а работающее как периферийное устройство. Для поддержки реализации таких архитектур применяется полносистемная версия Qemu. Подобный случай, строго говоря, входит за рамки

тестирование фронтенда TCG. Однако необходимость в проверке реализации подобных нюансов работы остаётся.

Ниже представлена схема проведения тестирования с использованием разработанного программного средства (см. рис. 2).

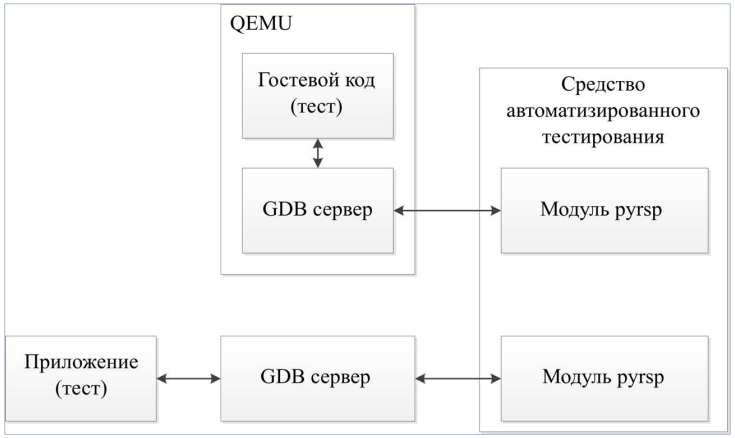

Рис. 2. Схема проведения тестирования
Fig. 2. Testing scheme

Конфигурация тестирования реализуется путём добавления управляющих комментариев в исходном коде теста, которые являются выражениями на языке Python, выполняемыми штатными средствами в окружении, настроенном нужным образом. Данные комментарии пишутся человеком, проводящим тестирование, и содержат команды специального вида, задающие проверки, которые будут выполнены во время тестирования. На данный момент программным средством поддерживаются следующие команды:

- `br` –установка точки останова для однократной проверки соответствия позиций выполнения тестового сценария;
- `brc` –установка точки останова для многократной проверки соответствия позиций выполнения тестового сценария;
- `bre` –завершить выполнение теста;
- `ch` –установка точки останова для однократной проверки соответствия позиций выполнения и равенства значений переменных тестового сценария;
- `chc` –установка точки останова для многократной проверки соответствия позиций выполнения и равенства значений переменных тестового сценарияю.

Пример теста и конфигурации тестирования, с использованием управляющих комментариев, представлен на рис. 3. Для поддержки тестирования в Qemu необходимо добавить код, предоставляющий возможность отладчику читать и писать в регистры процессора, а также отвечающий за поддержку установки отладочных точек останова (вставка транслятором необходимого кода tcg, реализующего точки останова).

Поддержка RSP фронтендом TCG не является обязательным (в техническом смысле) при реализации процессорной архитектуры Qemu. С практической точки зрения, RSP реализуют всегда. При этом её сравнительно просто добавить. Следовательно, будем считать, что поддержка RSP имеется всегда.


```
int main(void) {
    volatile uint32_t a = 0xABCDEF, b = 0x12345678, c = 0, i;

    for(i = 0; i < 20; i++) {
        if(i % 2) {
            // однократная проверка позиций выполнения
            c = b - a; //$br
        }
        else {
            // многократная проверка позиций выполнения
            c = b + a; //$brc
        }

        // однократная проверка значений всех переменных
        c = b & a; //$ch

        // однократная проверка значений переменной `c`
        c = b | a; //$ch.c

        // многократная проверка значений переменной `c`
        c = b ^ a; //$chc.c

        /* Также есть возможность комбинировать команды:
          - однократная проверка позиций выполнения и
            значений переменных `a` и `b`
          - многократная проверка значений переменной `c`
        */
        c = 0; //$br, chc.c, ch.a, ch.b
    }

    // завершить тестирование
    return 0; //$bre
}
```

Рис. 3. Листинг теста с управляющими комментариями
Fig. 3. Test listing

Для поддержки тестирования в Qemu необходимо добавить код, предоставляющий возможность отладчику читать и писать в регистры процессора, а также отвечающий за поддержку установки отладочных точек останова (вставка транслятором необходимого кода tcg, реализующего точки останова). Поддержка RSP фронтендом TCG не является обязательным (в техническом смысле) при реализации процессорной архитектуры Qemu. С практической точки зрения, RSP реализуют всегда. При этом её сравнительно просто добавить. Следовательно, будем считать, что поддержка RSP имеется всегда.

Программное средство c2t поддерживает многопоточный режим. Несколько тестов могут выполняться параллельно. Также оно имеет режим вывода промежуточной информации в ходе проведения тестирования. На момент написания данной работы реализовано 145 тестов. Названия тестов, которые тестировщику необходимо запустить, задаются регулярными выражениями.

5. Описание экспериментального стенда

В качестве эталона использовался персональный компьютер на базе процессора i7-4790 архитектуры AMD64. Версии ПО, использованного в эксперименте приведены в табл. 1. Звёздочкой помечены реализации, находящиеся в официальной ветке «master».

Табл. 1. Версии использованного ПО
Table 1. Versions of used software

фронтенд	Qemu	GCC
ARM	2.5.0*	4.9.3
MIPS	2.5.0*	5.4.0
MSP430_1	2.6.9	7.3.2
MSP430_2	2.12.1	7.3.2

Покрытие было оценено `gcov` версии 5.4.0.

6. Оценка покрытия

Фронтенд транслятора TCG реализован на языке Си. Реализация фронтенда TCG представляет собой набор вложенных конструкций `switch` (синтаксический анализатор), ветки которых содержат семантики соответствующих машинных инструкций. Данные семантики выражаются на промежуточном представлении TCG. Генерация кода TCG осуществляется путём вызова специальных функций.

В данной работе было проведена оценка двух покрытий. Под первым покрытием подразумевается количество выполненных (покрытых) ветвей конструкций `switch` фронтенда TCG, соответствующих уникальным инструкциям гостевого процессора. Оценка данного покрытия была произведена с использованием утилиты `gcov` для исследования покрытия кода. Данная утилита входит в состав пакета GCC. Под вторым покрытием подразумевается количество выполненных Qemu уникальных инструкций гостевого процессора.

Для оценки данного покрытия Qemu запускалась в режиме пошагового выполнения (`-singlestep`), с включенной отладочной трассировкой и выключенными сцеплением блоков транслированного кода (`-d exec,nochain`).

Перед пояснением обозначенных параметров напомним, что TCG транслирует гостевой блок блоками, получая подпрограммы для процессора основной машины. Выполнение подпрограммы приводит к изменениям в гостевой машине, которые бы произошли в ней от выполнения блока исходного гостевого кода, исполняй она его непосредственно. Посоле выполнения каждой подпрограммы происходит возврат в служебный код эмулятора с целью поиска/трансляции следующего блока (а именно, соответствующей ему подпрограммы). Если для данного блока известен следующий блок, то Qemu добавляет в конец подпрограммы переход на точку входа в следующую подпрограмму, **минуя возврат в служебный код**. Эта оптимизация, ускоряющая работу эмулятора, называется *сцеплением* блоков.

В режиме пошагового выполнения в блок всегда попадает одна гостевая инструкция. Перед выполнением подпрограммы **служебный код** эмулятора пишет в трассу **гостевой** адрес точки входа в блок. Из-за сцепления блоков, предотвращающим возврата в служебный код, в трассу не попадают гостевые адреса инструкций в прицепленных блоков. Это мешает учёту покрытых инструкций, поэтому сцепление блоков было отключено при оценке покрытия.

Гостевой адрес в трассе позволяет автоматически с помощью дизассемблера определить множество покрытых гостевых инструкций.

Далее приводится разъяснение того, какие инструкции при оценке покрытия считаются разными. Очевидно, что инструкции сложения и умножения всегда можно назвать разными. Но часто инструкции, выполняющие одну операцию, поддерживают разные режимы адресации операндов:

- непосредственное значение;
- регистр (так называемая «прямая адресация»);
- ячейка памяти, адресуемая значением в регистре (так называемая «косвенная адресация») и др.

Архитектура MSP430 имеет 7 режимов адресации. Даже если семантика инструкции не меняется при изменении режима адресации одного из операндов, то всё равно используется другой код из реализации фронтенда. А разработчику важно покрыть весь код фронтенда. Вариативность не ограничивается режимами адресации и всегда зависит от конкретной архитектуры процессора. Таким образом, критерий одинаковости инструкций зависит как от архитектуры, так и от решаемой разработчиком задачи. Далее описываются критерии, которых придерживались авторы для протестированных в данной работе гостевых архитектур.

6.1 Классификация инструкций

Инструкции получаются из 145 тестов на Си, имеющихся на момент написания работы (они входят в состав c2t). Данные тесты содержат следующие операции, предоставляемые языком Си:

1. присваивание (=);
2. арифметические: сложение (+), вычитание (−), умножение (*) и деление (/);
3. побитовые умножение (&), сложение (|), сложение по модулю 2 (^), отрицание (~), сдвиги вправо (>>)/влево (<<);
4. сравнения: равенство (==), неравенство (!=), больше (>), меньше (<), больше или равно (>=), меньше или равно (<=).

Также имеются тесты, которые содержат условные конструкции, циклы, вызовы функций. Данными тестами можно проверить реализации инструкций передачи и возврата управления, а также работу со стеком. Для последнего используются тесты, содержащие вызовы функций с большим количеством параметров. Это гарантируется размещением входных параметров в стеке, т.к. многие соглашения о вызове стараются передавать параметры через регистры.

Чтобы проверить реализации различных вариаций одной и той же инструкции процессора, тесты выполнены в вариантах, содержащих 8, 16, 32 и 64 разрядные знаковые и беззнаковые переменные.

Для подтверждения работоспособности (применимости) предложенного в рамках данной работы подхода было произведено тестирование существующих в официальном репозитории Qemu реализаций процессорных архитектур ARM32 и MIPS32, а также ещё двух реализаций процессорной архитектуры MSP430 (далее MSP430_1 и MSP430_2) , не находящихся в официальном репозитории. Тесты для данных архитектур были скомпилированы соответствующим компилятором `gcc` с выключенными оптимизациями (`-O0`). Реализация MSP430_1 доступна на github [14]. Однако для того, чтобы протестировать данную реализацию, потребовалось внести в неё поддержку RSP, согласно требованиям, обозначенным выше. Также потребовалось сократить количество тестов, пропустив проверку умножения. Это сделано по причине того, что в архитектуре MSP430 умножение реализуется аппаратно: на микроконтроллере присутствует специальное периферийное устройство. Модель же соответствующего устройства в эмуляторе пока отсутствует.

Реализация MSP430_2 является собственной реализацией, в которой были умышленно допущены ошибки с целью выявления их программным средством тестирования. Все эти ошибки были успешно обнаружены. Также были обнаружены и неумышленные ошибки.

Инструкции процессора можно разбить на следующие классы:

1. обработка данных и операции с памятью;
2. арифметические и логические операции;
3. операции изменения потока управления;
4. операции, специфичные для процессора.

Как было сказано выше, тестирование реализации инструкций 4 класса выходит за рамки данной работы и, следовательно, их нецелесообразно включать в оценку покрытия. Инструкции классов с 1 по 3 входят в оценку покрытия.

6.2 Покрытые инструкции

В табл. 2 содержатся полученные результаты по покрытию тестированием ветвей фронтенда транслятора TCG. Представлено отношение количества покрытых ветвей конструкций `switch` от общего количества ветвей, содержащих семантику уникальных инструкций. Данный результат также соответствует отношению количества выполненных Qemu инструкций от общего количества инструкций тестируемой гостевой процессорной архитектуры, реализованных в Qemu. В подсчёт общего количества инструкций для каждой из таких процессорных архитектур вошли уникальные по семантике инструкции 1, 2 и 3 классов. Те инструкции, которые возможно покрыть тестами, имеющимися на момент написания работы. Из полученного результата исключено число инструкций, которые предназначены для удобства написания программ и которые реализуются ассемблером.

Примерами таких инструкций для архитектуры ARM32 являются `adr` (загрузить адрес в регистр, реализуется инструкцией `add` или `sub`), `push` (положить в стек, реализуется инструкцией `stm`) и `pop` (достать из стека, реализуется инструкцией `ldm`). Для архитектуры MIPS32: `li` (загрузить «непосредственное» значение в регистр, реализуется инструкциями `lui` и `ori`) и `move` (копировать значение регистра в регистр, реализуется инструкцией `or`). Для архитектуры MSP430: `pop` (достать из стека, реализуется инструкцией `mov` со специальным режимом адресации), `inc` (инкрементировать, реализуется инструкцией `add`) и `ret` (возврат из подпрограммы, реализуется инструкцией `mov`).

Кроме того, все различные вариации отдельно взятой инструкции при подсчёте рассматриваются как одна инструкция. Например, для архитектуры ARM32 в качестве таких инструкций выступают инструкции с условным выполнением (например, `addge`), а также обновляющие значения флагов (например, `adds`). Не вошли в подсчет и специфичные для конкретного процессора инструкции. Информация для подсчёта общего количества инструкций была взята из [15], [16] и [17].

Табл. 2. Количество покрытых инструкций
Table 2. Number of covered instructions

Фронтенд	Покрыто	Всего	%
ARM32	38	61	62
MIPS32	42	67	63
MSP430_1	21	27	78
MSP430_2	12	19	63

В соответствии с табл. 2, в среднем покрывается около 60% инструкций. Такое покрытие не является выдающимся результатом. Однако данное покрытие возможно достичь, приложив небольшие усилия. Необходимо добавить поддержку процессорной архитектуры в c2t и установить компилятор для данной архитектуры.

Тем не менее разработанный инструмент предоставляет возможность покрыть инструкции, о которых было сказано выше. Например, инструкцию `addge` (сложить, если истинно условие «больше или равно») можно покрыть кодом, показанным на рис. 4.

```
void main(void)
{
    volatile int32_t a = 0xa, b = 0xb, c;
    if (a >= b)
        c = a + b;
    return;
}
```

Рис. 4. Листинг теста, покрывающего инструкцию `addge`
Fig. 4. Test listing covering `addge` instruction

При этом необходимо скомпилировать этот код, включив оптимизацию размера кода (параметр `-Os` для компилятора `gcc`). Это также актуально и для некоторых других архитектур, имеющих инструкции, которые выполняются, если истинно условие. Создание тестов, покрывающих различные вариации отдельных инструкций, является архитектурно зависимой задачей и выходит за рамки данной работы.

Табл. 3. Список покрытых/не покрытых инструкций
Table 3. List of covered/non-covered instructions

Фронтенд	Класс	Покрытые инструкции	Не покрытые инструкции
ARM32	1	ldr, mov, str, stm, ldm, mvn	—
	2	sub, lsl, lsr, eor, cmp, rsb, asr, add, adc, and, mul, orr, tst, mla, sbc, rrx, rsc, umull, ror	smull, smlal, umlal, cmn, teq, bic
	3	b, bl, bmi, beq, bcs, ble, blt, bhi, bne, bls, bcc, bgt, bge	bpl, bvs, bvc, bl(cc)
MIPS32	1	movn, lb, lw, slt, sltiu, slti, sltu, lbu, mflo, mfhi, lui, sw, lhu, sh, sb	movz, mthi, mtlo, ll, lh, lwl, lwr, swl, swr, sc
	2	addu, addiu, subu, mul, multu, mult, div, divu, nor, sllv, srlv, and, sll, xor, srav, ori, andi, sra, srl, or	add, addi, sub, clo, clz, madd, maddu, msub, msubu, xori
	3	bltz, bgtz, jal, bne, beq, jr, bgezal	blez, bgez, bltzal, j, jalr
MSP430_1	1	mov, push, sxt	swpb
	2	add, addc, sub, subc, cmp, and, bis, xor, rra, rrc, bit bic	dadd
	3	call, jnz, jge, jc, jeq, jl	reti, jmp, jn, jnc
MSP430_2	1	mov	—
	2	add, sub, cmp, and, bis, xor	bit, addc, bic
	3	call, jnz, jeq, jge, jl	jc, jmp, jn, jnc

Реализация архитектуры ARM32 была протестирована путём запуска тестов на виртуальных процессорах cortex-m3 и arm926. Реализация архитектуры MIPS32 была протестирована путём запуска тестов на виртуальных процессорах r4000 и 4kc.

Для тестирования ARM32 и MIPS32 были использованы все 145 тестов, для MSP430_1 – 69 тестов, а для MSP430_2 –49 тестов.

В табл. 3 представлены все инструкции, покрытые и непокрытые тестированием, с разбиением их по классам. Для процессорной архитектуры ARM32 запись `bl(cc)` в непокрытых инструкциях 3 класса является сокращённой записью 14 инструкций. Она подразумевает все условные вариации инструкции `bl`.

Приведенные в табл. 3 данные позволяют сделать вывод, что для всех протестированных в данной работе архитектур покрыты основные арифметические и логические инструкции, инструкции обработки данных и работы с памятью, инструкции изменения потока управления. Не покрыты некоторые инструкции условного перехода. Это обусловлено тем, что в языке Си нет выражений, имеющих семантику соответствующих инструкций. Некоторые инструкции не покрыты по причине отсутствия необходимых тестов. Например, для архитектуры ARM32 не покрыта инструкция `bic` (очистка определённых битов).

7. Направления дальнейших исследований

Перспективными направлениями дальнейших исследований являются следующие.

- Исследование влияния применения оптимизаций компилятора на покрытие инструкций и сам подход. Необходимо оценить, как будет улучшаться покрытие кода. Также необходимо использовать при написании тестов различные приемы, снижающие негативное влияние оптимизаций. Например, оптимизации могут «перемешивать» отображения строк исходного кода на строки машинного кода, в таком случае необходимо использовать `asm volatile`.
- Создание тестов с наибольшим покрытием инструкций, а также автоматизация данного процесса.
- Способы принуждения компилятора к использованию определённых гостевых инструкций.

8. Заключение

В ходе данной работы был проведён анализ существующих решений тестирования фронтенда транслятора TCG эмулятора Qemu. Было установлено, что рассмотренные решения используют для сравнения эталон той же процессорной архитектуры, что и у тестируемой реализации. Данное требование делает неприменимыми эти решения при отсутствии такого эталона.

В работе предложен подход к тестированию, который направлен на решение указанной ы Идея предлагаемого подхода заключается в том, что можно написать программу на языке высокого уровня так, что её получится скомпилировать под разные процессорные архитектуры. Тем самым достигается возможность использовать фиксированный эталон – процессор машины разработчика. Однако накладывается ограничение на покрытие реализаций инструкций, специфичных для конкретной процессорной архитектуры. На основе данного подхода реализовано программное средство автоматизированного тестирования.

Применимость подхода была подтверждена тестированием реализаций двух процессорных архитектур (ARM32 и MIPS32), находящихся в официальном репозитории Qemu, а также ещё двух реализаций (MSP430_1 и MSP430_2), не находящихся в официальном репозитории. В собственной реализации MSP430_2 были обнаружены как умышленно оставленные ошибки, так и не умышленные ошибки.

Также в работе приведена оценка покрытия инструкций процессорных архитектур для протестированных реализаций. Согласно полученному результату в среднем покрывается около 60% инструкций. Такое покрытие не является выдающимся результатом. Однако данное покрытие возможно достичь, приложив небольшие усилия. Предложенным подходом удалось покрыть основные арифметические и логические инструкции, инструкции обработки данных и работы с памятью, инструкции изменения потока управления.

Список литературы / References

[1]. Ефимов В.Ю., Беззубиков А.А., Богомолов Д.А., Горемыкин О.В., Падарян В.А. Автоматизация разработки моделей устройств и вычислительных машин для QEMU. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 77-104 / Efimov V.Yu., Bezzubikov A.A., Bogomolov D.A., Goremykin O.V., Padaryan V.A. Automation of device and machine development for QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 77-104 (In Russian). DOI: 10.15514/ISPRAS-2017-29(6)-4.

[2]. Bezzubikov A., Belov N., Batuzov K. Automatic dynamic binary translator generation from instruction set description. In Proc. of the 2017 Ivannikov ISPRAS Open Conference, 2017, pp. 27-33. DOI: 10.1109/ISPRAS.2017.00012.

[3]. W.E. Howden, Theoretical and empirical studies of program testing. In Proc. of the 3rd international conference on Software engineering, 1978, pp. 305-311.

[4]. Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, Danilo Bruschi. Testing CPU emulators. In Proc. of the 18th international symposium on Software testing and analysis, 2009, pp. 261-272.

[5]. Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, Danilo Bruschi. Testing system virtual machines. In Proc. of the 19th international symposium on Software testing and analysis, 2010, pp. 171-182.

[6]. Qiuchen Yan and Stephen McCamant. Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining. In Proc. of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2018, 13 p..

[7]. Risu: random instruction sequence tester for userspace [online] Available at: https://git.linaro.org/people/pmaydell/risu.git/about/, accessed: 09.08.2019.

[8]. A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. Extensible environment for test program generation for microprocessors. Programming and Computer Software, vol. 40, issue 1, 2014, pp 1-9.

[9]. Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, Sang Kil Cha. Testing intermediate representations for binary analysis. In Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 353-364.

[10]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2012, pp. 337-348.

[11]. Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. Cardinal pill testing of system virtual machines. In Proceedings of the 23rd USENIX Security Symposium, 2014, pp. 271-285.

[12]. pyrsp. Available at: https://github.com/stef/pyrsp, accessed: 02.08.2019.

[13]. pyelftools. Available at: https://github.com/eliben/pyelftools, accessed:23.09.2018.

[14]. Qemu MSP430. Available at: https://github.com/draperlaboratory/qemu-msp, accessed: 16.07.2019.

[15]. ARM and Thumb-2 Instruction Set Quick Reference Card. Available at: http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf, accessed: 16.07.2019.

[16]. MIPS Instruction Reference. Available at: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-Q RC-01.01.pdf, accessed: 16.07.2019.

[17]. MSP430x2xx Family User's Guide. Available at: http://www.ti.com/lit/ug/slau144j/slau144j.pdf, accessed: 16.07.2019.

Информация об авторах / Information about authors

Дмитрий Сергеевич КОЛТУНОВ – стажер-исследователь ИСП РАН. Научные интересы: анализ бинарного кода, эмуляция и виртуализация.

Dmitry Sergeevich KOLTUNOV – a research trainee at ISP RAS. Research interests: binary code analysis, emulation and virtualization.

Василий Юрьевич ЕФИМОВ является младшим научным сотрудником ИСП РАН. Его научные интересы включают компиляторные технологии, безопасность ПО, анализ бинарного кода, параллельное программирование, эмуляция и виртуализация.

Vasily Yuryevich EFIMOV is a junior researcher at ISP RAS. His research interests include compiler technologies, software security, binary code analysis, parallel programming, emulation and virtualization.

Вартан Андроникович ПАДАРЯН является кандидатом физико-математических наук, ведущим научным сотрудником ИСП РАН, доцентом кафедры системного программирования ф-та ВМК МГУ. Его научные интересы включают компиляторные технологии, безопасность ПО, анализ бинарного кода, параллельное программирование, эмуляция и виртуализация.

Vartan Andronikovich PADARYAN is Candidate of Physical and Mathematical Sciences, Leading Researcher at ISP RAS, Associate Professor of the System Programming Department of the The faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. His research interests include compiler technologies, software security, binary code analysis, parallel programming, emulation and virtualization.