Санкт-Петербургский государственный университет

Кафедра системного программирования

Кижнеров Павел Александрович

Организация автоматизированного тестирования встраиваемого программного обеспечения

Отчет по учебной (технологической) практике

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Санкт-Петербург
2022

Оглавление

Введение

Данная работа выполняется в рамках разработки прошивки для новой версии фитнес-браслетов, разрабатываемых американской компанией-производителем для продажи на американском и мировом рынках. Аппаратная часть состоит из двух процессоров Microchip SAMD51 (ARM v8 Cortex), модуля Bluetooth Low Energy Murata (далее BLE), датчиков и дисплея. Такая связка на протяжении всего времени должна работать как единое целое, потребляя при этом минимум энергии и предоставляя максимальную степень удобства пользователю. Качество конечного продукта в значительной степени определяется качеством встраиваемого программного обеспечения, поэтому важно обеспечить процесс автоматического тестирования и мониторинга результатов. Работа с платформозависимым кодом, исполняемым на контроллере, имеет следующие специфические особенности:

- запуск кода возможен только после загрузки скомпилированного бинарного исполняемого файла в память контроллера; это увеличивает трудоемкость отладки;
- мониторинг результатов осуществляется посредством чтения данных из последовательного порта, через который контроллер подключен к инструментальной машине; это ограничивает возможности отладки.

1. Постановка задачи

Для автоматизации тестирования и обеспечения контроля качества продукта планируется разработка специального инструмента, работающего с тестовым стендом (связка 2 контроллеров - коммуникационного и измерительного, модуля BLE и периферийных датчиков) и обладающего следующей функциональностью (пункты упорядочены по важности):

- автоматический запуск и мониторинг тестов из инструментальной машины;
- поиск некорректных входных данных с помощью символьного исполнения;
- анализ результатов тестов;
- подсчет метрик кода;

2. Обзор существующих подходов

Для модульного и интеграционного тестирования проектов, написанных на языках C/C++, используются такие фреймворки как Gtest [2], Catch [4], Mettle [5], Boost.Test [1]. Все эти решения похожи между собой с точностью до особенностей конфигурации и синтаксиса тестирующих функций. Также в данных фреймворках есть возможность генерировать отчеты в формате XML, с которым работают службы непрерывной интеграции (напр. Jenkins). Связка тестирующего фреймворка и службы непрерывной интеграции обеспечивает автоматизацию процесса запуска тестов и мониторинга результатов, однако это применимо только если на сервере непрерывной интеграции есть возможность собирать и запускать тестируемый проект на целевой архитектуре и нет ограничений, связанных с потреблением вычислительных ресурсов.

Контроллеры обладают ограниченными вычислительными возможностями по сравнению с полноразмерными компьютерами, поэтому при разработке встраиваемого программного обеспечения особое внимание уделяется потреблению ресурсов. Также контроллеры имеют возможность переходить в различные режимы энергопотребления, динамически изменять аппаратные параметры, подключать / отключать периферийные устройства, что может повлиять на работоспособность встраиваемого программного обеспечения. Поэтому тестирование аппаратнозависимых модулей, как правило, производится под управлением инструментальной машины - таким образом достигается приближенность к реальным условиям использования контроллера и расширение спектра сценариев тестирования.

Встраиваемое программное обеспечение обычно пишется на языках C/C++. Ошибки исполнения находятся путем прогона тестов, однако полное покрытие всех ветвей исполнения кода они гарантировать не могут. Для этих целей используется техника символьного исполнения, которая способна эффективно искать входные данные, приводящие к ошибкам.

Каждый проект предъявляет уникальные требования к разраба-

тываемому встраиваемому программному обеспечению, тестирующему окружению, аппаратой конфигурации тестового стенда, поэтому, часто для конкретного проекта не удается подобрать полностью подходящее готовое решение, и приходится полностью лии частично разрабатывать его заново.

Для тестирования платформозависимого кода используются готовые инструменты генерации комплексных отчетов. Популярные решения: TETware RT, OpenTest, autotestnet, DejaGnu, Robot Framework. TETware RT и OpenTest требуют UNIX окружение, в DejaGnu отсутствуют встроенные средства сверки ожидаемого и полученного результата, autotestnet требует windows окружение и имеет проблемы с доступом к документации [3], Robot Framework мультиплатформенный, но требует реализацию интерфейса коммуникации с контроллером (как и все остальные решения).

По результатам проведенного анализа, было принято решение организовать автоматизацию запуска и мониторинга результатов платформозависимых тестов на базе RobotFramework, а также искать ошибки исполнения и генерировать наборы входных данных с помощью KLEE.

3. Реализация

3.1. Структура решения

3.1.1. Автоматический запуск и мониторинг тестов из инструментальной машины

Предполагается что инициировать запуск модульного теста и собирать результат будет программное обеспечение на инструментальной машине. Для этого нужен надежный двунаправленный канал между тестовым стендом и машиной. На тестовом стенде есть четыре внешних интерфейса - один BLE и три UART.

BLE нельзя использовать по нескольким причинам: инициализация BLE требует времени, а требуется тестировать модули, которые инициализируются гораздо раньше BLE; BLE подключен только к коммуникационному контроллеру, поэтому нельзя тестировать измерительный контроллер изолированно; закрываются возможности тестирования самого BLE; в офисе много устройств, работающих на одной частоте с BLE, а значит канал связи подвержен интерференциям.

Для коммуникации подходит только UART. Был выбран тот, который сопряжен с USB разъемом на плате, так как с такой конфигурацией требуется меньше проводных подключений к инструментальной машине.

Выбранный интерфейс также является отладочным, то есть в поток вывода могут попадать побочные данные. Чтобы организовать общение с платой, необходим механизм надежной двусторонней доставки и разбора данных - брокер сообщений, который в дальнейшем будет использоваться фреймворком для тестирования.

В качестве каркаса ПО для инструментальной машины можно использовать один из распространенных фреймворков для тестирования, позволяющих генерировать комплексные отчеты по результатам модульных и/или интеграционных тестов. TETware RT, OpenTest, autotestnet, DejaGnu, Robot Framework - основные фреймворки для подобных задач. Вне конкуренции оказался Robot Framework ввиду простоты написания

(язык - python), кроссплатформенности, автоматического документирования и богатой инфраструктуры. Некоторые из других рассмотренных решений либо требуют unix-окружение (TETware RT, OpenTest), либо в них отсутствует встроенное средство проверки результатов (DejaGnu), либо имеют слабую документацию.

3.1.2. Генерация входных даных с помощью символьного исполнения

в качестве символьного интерпретатора был выбран KLEE, так как на момент написания настоящей работы, он является единственным бесперебойно работающим на реальных проектах решением.

В первую очередь необходимо реализовать скрипты настройки инфраструктуры: shell-скрипт и Dockerfile для установки символьного интерпретатора. Затем требуется настроить скрипты компиляции для генерации llvm-биткода, который необходим для работы выбранного символьного движка, KLEE. Финальным этапом является непосредственно внедрение API KLEE в исходный код и запуск символьного интерпретатора.

3.2. Реализация

3.2.1. Автоматический запуск и мониторинг тестов из инструментальной машины

Для двунаправленной коммуникации используется интерфейс UART. По нему пересылаются пакеты определенного формата, которые легко определить в потоке по последовательности байтов в начале пакета.

```
typedef struct message
{
    uint8_t sig[4];              /**< Message signature */
    int32_t id;                  /**< Message id*/
    uint32_t payload_size;       /**< Payload size */
    uint32_t checksum;           /**< Crc32 checksum */
    uint32_t domain_receiver;    /**< Receiver domain number */
    uint32_t action_receiver;    /**< Receiver domain action */
    uint32_t domain_sender;      /**< Sender domain number */
    uint32_t action_sender;      /**< Sender domain action */
    uint8_t payload[0];          /**< Payload buffer */
} message;
```

В языке Си поля структуры располагаются в памяти последовательно, поэтому в структуре пакета первым полем добавлена преамбула (поле sig) - по ней пакет определяется среди потока байтов. Далее идут поля с метаинформацией для корректной обработки и маршрутизации пакетов. В самом конце непосредственно сама полезная информация.

Приемом, передачей и маршрутизацией пакетов управляет брокер сообщений - сущность, реализованная как на стороне тестового стенда так и на инструментальной машине, единственной задачей которой является детектирование и обработка и пересыл пакетов с данными.

Для минимизации количества ошибок в процессе разработки такая сущность должна быть идентичной на обоих концах коммуникации. Единственный язык реализации на стороне тестового стенда - Си.

На стороне инструментальной машины язык реализации определяется использованием Robot Framework - python. Поэтому был реализован модуль на языке python, который посредством библиотеки ctypes оборачивает реализацию брокера на Си. Также был реализован еще один модуль, предоставляющий высокоуровневый интерфейс коммуникации с тестовым стендом и использующий модуль-обертку.

Другими участниками проекта реализован скрипт генерации таблицы соответствий тестируемых функций на стенде и вызывающих функций на стороне инструментальной машины.

Брокер сообщений был добавлен в Robot Framework, в итоге появилась возможность инициировать запуск тестов полностью на инструментальной машине. Также выбранный фреймворк имеет возможность генерировать XML отчеты тестирования, что решает проблему интеграции с CI/CD сервисом Jenkins.

3.2.2. Генерация входных данных с помощью символьного исполнения

Для успешного использования техники символьного исполнения с помощью KLEE проект должен соответствовать определенным требованиям.

Для использования выбранного символьного интерпретатора необходимо:

- подключить библиотеку KLEE;
- настроить в качестве компилятора clang;
- настроить в качестве компоновщика lld;
- отключить оптимизацию;
- настроить порождение LLVM IR промежуточных файлов;
- использовать API KLEE;

В текущем проекте сборка осуществляется с помощью утилиты CMake. Согласно приведенному порядку подготовки, в корневой скрипт CMake были добавлены следующие инструкции.

```
...
# настройка компилятора
set(CMAKE_C_COMPILER clang-11)
set(CMAKE_CXX_COMPILER clang++-11)

add_compile_options(
-emit-llvm                  # порождение LLVM IR файлов
-O0                         # отключение оптимизации компилятором
-Xclang                     # передача последующих флагов в clang
-disable-O0-optnone)        # включение оптимизации LLVM

# настройка компоновщика
set(CMAKE_EXE_LINKER_FLAGS
${CMAKE_EXE_LINKER_FLAGS}
''-fuse-ld=lld-11'')

# подключение библиотеки KLEE
add_library(klee SHARED IMPORTED)

set_target_properties(klee PROPERTIES
IMPORTED_LOCATION ''${KLEE_PREFIX}/build/lib/libkleeRuntest.so''
INTERFACE_INCLUDE_DIRECTORIES ''${KLEE_PREFIX}/build/include''

target_link_libraries(${PROJECT_NAME}
PUBLIC
klee)
...
```

Пример использования функций из библиотеки KLEE:

```
#include ''klee/klee.h''
```

```
#include <stdio.h>

int abs(int x) {
    int abs = 0;

    if (x > 0)
    {
        abs = x;
    }
    else
    {
        abs = -x;
    }
    return abs;
}

int main(int argc, const char *argv[])
{
    int x;
    // x помечается как символьная переменная
    klee_make_symbolic(&x, sizeof(x), ”x”);
    return abs(x);
}
```

Порядок использования символьного исполнения для генерации входных данных:

- собрать проект;
- собрать все промежуточные LLVM IR в единый файл;
- запустить символьное исполнение;

На практике в качестве компилятора требуется не только clang, но и gcc-arm-none-eabi, поэтому сборка организована в виде shell-скрипта,

который помимо непосредственных задач позволяет задавать необходимую конфигурацию. При необходимости генерации LLVM IR биткода, после компоновки ELF-файла также создается и BC файл.

Ради удобства использования символьного движка KLEE был также разработан shell-скрипт, позволяющий конфигурировать расположение порождаемых артефактов, содержащих сгенерированные входные данные для тестируемых функций, отчеты об ошибках, трассы и тд.

Результат символьного исполнения - директория с различными файлами, в том числе содержащие сгенерированные входные данные. Такие файлы имеют расширение `.ktest` и их можно использовать для запуска тестируемой функции, заменяя символьные переменные на конкретные значения. Для этого достаточно вызвать ELF файл как обычную программу, передав в переменную среды `KTEST_FILE` путь до файла с расширением `.ktest`. Пример:

```
KTEST_FILE=./klee-last/test00001.ktest ./program
```

Особый интерес представляют те файлы, которые имеют расширение `.err`. В них указана трасса, которая привела к ошибке исполнения. Файлы с таким же названием, но с расширением `.ktest` содержат входные данные, которые приводят к ошибке.

Выполненная к настоящему моменту часть работы позволила заложить основу для автоматической генерации тестов разработана: реализованы скрипты настройки инфраструктуры, сборки и запуска символьного исполнения. Часть ключевой функциональности с помощью данной техники уже протестирована, пока что было найдено несколько ошибок исполнения, связанные с указателями, и 1 ошибки разработки.

4. Результат

В рамках данной работы на текущий момент реализовано:

- автоматический запуск и мониторинг тестов из инструментальной машины:
  - брокер сообщений;
  - высокоуровневая обертка над брокером на python;
  - обертка интегрирована в Robot Framework;
  - настроен CI/CD;
- генерация входных даных с помощью символьного исполнения:
  - созданы скрипты настройки инфраструктуры, сборки проекта, запуска символьного исполнения;
  - настроена сборка проекта;
  - сгенерированы тесты для части ключевой функциональности;
  - найдены критические ошибки исполнения;

Список литературы

[1] Boost.Test // Boost.Test. — 2001. — URL: https://www.boost.org/doc/libs/1_75_0/libs/test/doc/html/index.html (online; accessed: 02.12.2021).

[2] Google. googletest // GoogleTest. — 2017. — URL: https://github.com/google/googletest (online; accessed: 02.12.2021).

[3] alexkalmuk // Тестирование встроенных систем. — 2009. — URL: https://habr.com/ru/company/embox/blog/239387/ (дата обращения: 13.12.2021).

[4] catchorg. A modern, C++-native, test framework for unit-tests // Catch2. — 2017. — URL: https://github.com/catchorg/Catch2 (online; accessed: 02.12.2021).

[5] jimporter. A C++17 unit test framework // Mettle. — 2017. — URL: https://github.com/jimporter/mettle (online; accessed: 02.12.2021).