Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.М07-мм

Организация автоматического тестирования встраиваемого программного обеспечения

Кижнеров Павел Александрович

Отчёт по преддипломной практике
в форме «Производственное задание»

Научный руководитель:
заведующий кафедрой системного программирования, д. ф.-м. н., А. Н. Терехов

Санкт-Петербург
2023

Оглавление

Введение

Иллюзия – искажение восприятия реальности, которое возникает у человека ввиду различных факторов: ограниченности сенсорных систем, индивидуальных убеждений, стереотипов и тд. С данным феноменом люди на протяжении всей своей жизни сталкиваются постоянно. Иллюзии важны и необходимы человеческой психике для снижения уровня тревожности, однако зачастую они препятствуют достижению поставленных целей. Самой распространенной и деструктивной иллюзией является ощущение твердой уверенности в чем-то. С одной стороны, оно дает чувство спокойствия, с другой – вводит в заблуждение. По этой причине при решении задач необходимо контролировать корректность промежуточных результатов.

В области разработки программного обеспечения существует практика управления качеством продукта. Контроль корректности результатов, коими являются функции, модули или сам программный продукт, традиционно осуществляется с помощью тестирования. Оно и есть способ устранения иллюзий работоспособности. Эффективной практикой разработки является автоматизация тестирования средствами системы непрерывной интеграции и доставки. Такой подход обеспечивает быструю обратную связь, если возникли проблемы. Существенную часть несовершенств удается ликвидировать еще на этапе их зарождения.

Чем больше в системе объектов и связей между ними, тем выше сложность обнаружения источника дефектов. В прикладном программировании в роли таких объектов выступают логические сущности, то есть сложность обнаружения проблем связана исключительно с требованиями к программному продукту. Однако существуют области, которые сложны сами по себе ввиду специфики, например разработка встраиваемого программного обеспечения. Появляются дополнительные физические сущности: целевая платформа, инструментальная машина, периферийные устройства, интерфейсы беспроводной связи и тд. Также имеют место следующие свойства и ограничения, присущие данной области: малый объем вычислительных ресурсов, влияние физической

среды на каналы связи, ограниченное время автономной работы, доступность платформы для разработчиков и тд.

Настоящая работа выполняется в рамках разработки встраиваемого ПО для новой серии фитнес-браслетов западной компании. Коммерческая ценность устройств заключается в уникальной технологии, которая позволяет точно и автоматически измерять такие метрики организма как количество поступающий калорий, гидратацию тела, артериальное давление и другие. У актуальных устройств есть ряд несовершенностей, связанных с архаичностью аппаратных компонентов, временем работы, нестабильной работы отдельных компонентов. По этой причине компания приняла решение начать разработку новых устройств, делая акцент на тестировании, чтобы избежать подобных недостатков.

1. Постановка задачи

Целью настоящей работы является создание средств автоматического модульного и интеграционного тестирования встраиваемого программного обеспечения для фитнес-браслета с использованием техники символьного исполнения для верификации критически важных функций.

Для достижения этой цели были поставлены следующие задачи.

- Изучить особенности устройств линейки фитнес-браслетов, подходы тестирования и верификации встраиваемого программного обеспечения.
- Учитывая технические особенности устройств линейки фитнес-браслетов, спроектировать и реализовать систему сборки, которая позволит конфигурировать прошивку в зависимости от аппаратных параметров целевого устройства.
- Проанализировать, выбрать и ввести в эксплуатацию инструменты и подходы для автоматического тестирования, сбора и анализа результатов.
- Изучить возможность применения техники символьного исполнения, и верифицировать критически важные функции, сформулировать набор тестовых данных для них.

2. Обзор

Контекстом данной работы является разработка прошивки для новой серии фитнес-браслетов. Концептуально актуальные устройства состоят из следующих аппаратных компонентов: коммуникационный микроконтроллер ESP32 от Espressif Systems, отвечающий за синхронизацию данных с мобильным устройством и вывод информации на дисплей, если он имеется; измерительный микропроцессор с архитектурой ARM, который собирает, обрабатывает данные и считает метрики тела; дисплей; датчики; тактильное устройство вывода.

В новых браслетах (см. рис. 3) принято решение обновить периферийные устройства ввода/вывода, отказаться от ESP32 и использовать два одинаковых современных микропроцессора Atmel с архитектурой ARM от Microchip. Это привело к необходимости использовать отдельный модуль BLE (Bluetooth Low Energy), так как в актуальных устройствах его предоставляет ESP32.

Двухпроцессорная архитектура обусловлена коммерческой ценностью алгоритмов подсчетов метрик, находящихся на прошивке для измерительного модуля. На заводе в Китае программное обеспечение загружается только для коммуникационного модуля. Измерительный микропроцессор, прошивается при синхронизации со смартфоном конечного потребителя.

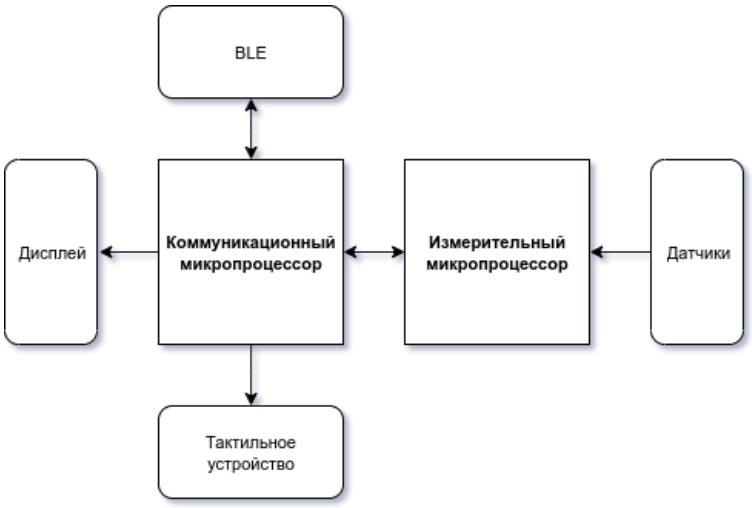

Рис. 1: Концептуальная схема нового устройства с дисплеем

Компания Microchip предоставляет программную платформу ASF (Atmel Software Framework), которая включает в себя набор библиотек и конфигуратор для микропроцессоров Atmel. Также производитель предлагает использовать интегрированную среду разработки Microchip Stud на базе Visual Studio от Microsoft. Разработка ПО предполагается с помощью языков C/C++.

Таким образом, чтобы создать программное обеспечение для новой линейки фитнес браслетов, компании необходимо адаптировать старые версии прошивок для измерительного и коммуникационного модулей под новые микропроцессоры, добавить модуль взаимодействия с BLE и пересмотреть подход к сборке и организации исходного кода.

2.1. Инструменты сборки

Разработка на языках C/C++ предполагает сборку проекта. Используя поиск в сети интернет удалось получить релевантный список, состоящий из следующих инструментов: GNU make, Ninja, Meson, SCons, Build2, Bazel, Ant, Gradle, Maven, MSBuild, CMake, Premake, GNU Autotools, QMake.

GNU make — широко используемое средство для автоматической сборки, стандарт де-факто для разработки под операционными системами семейства UNIX. К основным преимуществам можно отнести переносимость, документированность, количество обучающих материалов и интеграцию с большинством интегрированных сред разработки. Основные недостатки заключаются в трудоёмкости расширяемости, сложно понимаемом синтаксисе и низкой производительности на крупных проектах.

Ninja является разработкой Эвана Мартина, сотрудника Google, и представляет собой легковесную систему автоматической сборки нацеленную на производительность. Обладает высокой скоростью работы, платформной независимостью. Предполагает использование генератора скриптов сборки, что избавляет разработчиков от необходимости читать документацию по этому инструменту.

Meson является относительно новым инструментом. Первая версия вышла в 2014 году и активно развивается. Многие проекты уже перешли именно на эту систему сборки ввиду высокой производительности, выразительности и простоты синтаксиса сборочных скриптов. Для описания сценариев сборки используется собственный синтаксис.

SCons — средство автоматической сборки, использующее Python в качестве языка описания конфигураций. Более простая и производительная альтернатива GNU make. Недостатки данного инструмента заключаются в необходимости писать много кода для простейших задач и в сильном снижении производительности на больших проектах.

Build2 был создан специально для крупных проектов на C/C++. Обладает высокой производительностью и переносимостью, масштабируемый и не имеет сторонних зависимостей. Не позволяет генерировать проектные файлы, требует постоянного обращения к документации.

Bazel активно использует Google. Данный инструмент хорошо поддается масштабированию, обладает высокой производительностью, поддерживается основными операционными системами. Способен собирать проекты на разных языках программирования. К основным недостаткам можно отнести жесткие ограничения на структуру проекта, большое количество внешних зависимостей и малое количество обучающих материалов в интернете.

MSBuild является разработкой Microsoft и может быть использован только в контексте Visual Studio.

Ant, Maven, Gradle — системы сборки для проектов на Java, однако позволяют работать и с C/C++ с помощью сторонних плагинов. Для описания сценариев сборки используется достаточно простой язык XML. На момент написания данного текста, Ant редко используется даже для целевого языка программирования. К недостаткам данного инструмента можно отнести архаичность и слабо развитую поддержку. Maven и Gradle актуальны, обладают мощными средствами управления зависимостями, хорошей поддержкой и большим количеством обучающих материалов, однако сборка проектов на C/C++ не является их приоритетным направлением развития.

Далее последует описание специальных средств, генераторов, предназначенных для автоматического порождения скриптов сборки. Такой подход повышает масштабируемость проектов.

CMake — переносимая утилита для автоматической генерации файлов сборки Ninja и GNU make согласно файлу сценария на собственном языке. Имеет большую аудиторию, в интернете легко найти обучающие материалы, позволяет создавать проектные файлы для большинства сред разработки. Обладает понятным синтаксисом, перенимает все достоинства и недостатки выбранной системы сборки.

Premake позиционирует себя как простой и быстрый инструмент генерации скриптов управления сборкой, используя синтаксис языка Lua. Позволяет генерировать проектные файлы для Visual Studio, XCode и CodeLite. На практике данное средство сложно поддается пониманию ввиду сочетания процедурного языка и декларативности описания сценариев сборки. К прочим недостаткам можно отнести низкую информативность сообщений об ошибках.

GNU Autotools также позволяет собирать проекты на C/C++, но основная цель использования данного инструмента — поддержка переносимости между *NIX системами. Состоит из следующих утилит: Automake, Autoconf, Libtool. Automake предполагает генерацию make-файлов, поэтому наследует свойства GNU make. Пользователи отмечают высокий порог входа и не советуют использовать данное средство без острой необходимости.

QMake — устаревшее платформно-независимое средство, позволяющее автоматизировать генерацию make-файлов. Перенимает свойства GNU make. Может быть использован только для проектов Qt C++.

Ниже приведена сравнительная таблица описанных систем сборки. Масштабируемым считается инструмент, если с помощью него был реализован крупный проект. Популярным, если в сети удалось найти актуальные обсуждения в контексте решения прикладных задач. Система сборки считается малопроизводительной, если существуют жалобы пользователей на соответствующих ресурсах.

	Масштабируемость	Популярность	Производительность на крупных проектах	Платформы	Генераторы
GNU make	Да - с генератором Нет - без генератора	Да	Нет	*NIX	CMake, Premake, Automake, QMake
Ninja	Да	Да	Да	*NIX, Windows	CMake
Meson	Да	Да	Да	*NIX, Windows	Нет
SCons	Да	Да	Нет	*NIX, Windows	Нет
Build2	Да	Нет	Да	*NIX, Windows	Нет
Bazel	Да	Нет	Да	*NIX, Windows	Нет
MSBuild	Да	Нет	Нет	Windows	Нет
Ant	Да	Нет	Нет	*NIX, Windows	Нет
Maven	Да	Да	Да	*NIX, Windows	Нет
Gradle	Да	Да	Да	*NIX, Windows	Нет

Таблица 1: Сравнительная таблица систем сборки

2.2. Средства тестирования

В прикладных проектах на C/C++ для целей тестирования классическими инструментами являются: Google Test, Catch, Boost.Test.

Первая публичная версия Google Test была выпущена в 2018 году, и является одним из самых популярных инструментов. Обладает переносимостью, позволяет автоматически определять наборы тестирующих функций и генерировать отчеты в формате XML. Присутствует емкая и понятная документация.

Catch впервые был опубликован в 2016 году. Не имеет внешних зависимостей, поставляется одним заголовочным файлом. Также позволяет генерировать XML отчеты.

Boost.Test — фреймворк, развивающийся с 2001 года. Актуален на момент написания данного текста, однако не пользуется большой популярностью среди разработчиков из-за наличия зависимостей от библиотек Boost и неинформативностью сообщений об ошибках ввиду использования макросов в качестве основы инструмента.

	Генерация XML	Зависимости	Переносимость
Google Test	Да	Нет	Да
Catch	Да	Нет	Да
Boost.Test	Да	Да	Да

Таблица 2: Сравнительная таблица фреймворков тестирования для C++

Тестирование встраиваемого ПО несколько отличается от прикладного, так как появляется платформная зависимость кода. Также ввиду ограниченности вычислительных ресурсов аппаратных платформ, их архитектуры и прочих факторов, зачастую невозможно организовать интеграционное тестирование вышеописанными средствами, так как оно предполагает управление модулями системы за ее пределами. По этим причинам используется иной набор средств, который позволяет коммуницировать с целевой платформой посредством инструментальной машины. Самыми релевантными оказались: Robot Framework, Deja GNU, TETware RT, Autotestnet, OpenTest.

Robot Framework относительно новый, но набирающий популярность мультиплатформенный инструмент. Реализован на языке Python. Для коммуникации с целевой платформой требует наличие соответствующего модуля.

Deja GNU — средство тестирования, основанный на Expect, инструменте автоматизации интерактивных приложений, например telnet, ftp и другие. Требует командный интерпретатор на стороне целевой платформы.

Autotestnet также основан на Expect и выделяется простотой использования, однако доступен исключительно в виде приложения с GUI, что лишает возможности интегрировать данное средство с CI/CD.

TETware RT — фреймворк для тестирования, предоставляющий возможность удаленного вызова функций на целевой платформе. Необходимы доработки в виде реализации предоставляемого интерфейса на стороне встраиваемого ПО.

OpenTest поставляет готовое решение для исполнения тестов на целевой платформе в виде Test Execution Engine, одновременно накладывая ограничения на платформу инструментальной машины. Не может быть использован для целевой платформы без операционной системы.

	Платформы	Коммуникация с платформой	Генерация XML	Интеграция с CI/CD
Robot Framework	*NIX, Windows	Реализуется самостоятельно	Да	Да
Deja GNU	*NIX	Expect	Да	Да
Tetware RT	*NIX	Представлен интерфейс	Нет	Нет
Autotestnet	Windows	Expect	Нет	Нет
OpenTest	*NIX	Test Execution Engine	Да	Да

Таблица 3: Сравнительная таблица инструментов для интеграционного тестирования встраиваемых систем.

2.3. Подход к верификации

Верифицировать программу — значит доказать, что она в произвольный момент времени находится в тех состояниях, в которых от нее ожидается. На практике проще решать немного иную задачу — исследовать на предмет достижения ошибочных состояний. Техника символьного исполнения гарантирует, что все ветви потока исполнения будут посещены. Таким образом, если в код добавить условие, проверяющее ошибочное состояние и запустить программу на символьном интерпретаторе, то становится возможным рассуждать о достижимости этого состояния. Более того, в качестве побочного продукта символьного исполнения получается набор исходных данных, который гарантированно покрывает все ветви кода.

Данный подход обладает высокой фундаментальной вычислительной сложностью. Символьные интерпретаторы строятся на базе SMT решателей, которые в свою очередь используют SAT в качестве основы. Как известно, SAT является NP-полной задачей, поэтому техника символьного исполнения на практике может быть применена только для отдельных частей программ ввиду колоссальных вычислительных затрат.

Для C/C++ существует единственный работающий и открытый символьный интерпретатор, KLEE.

3. Сборка и организация кода

При разработке ПО для актуальной версии фитнес-браслетов, команда использовала по отдельному репозиторию для хранения кода коммуникационной и измерительной прошивок ввиду разнородности архитектур целевых платформ. В новой версии микропроцессоры идентичны, более того, обновление аппаратных компонентов существенно повлияло на кодовую базу.

Учитывая вышеописанные факты, а также то, что устройства одной линейки отличаются только периферийными устройствами, было принято решение организовать код как для измерительного, так и для коммуникационного модулей в одном репозитории, фабрике прошивок.

Естественным образом исходный код можно разделить на зависящий от платформы и независимый. Платформно-зависимый код имеет смысл размещать в отдельных функциональных архитектурных единицах таким образом, чтобы их можно было по некоторому условию при сборке подключать к аппаратно-независимому коду. Платформно-независимая часть состоит из библиотек с типами, алгоритмами и структурами данных.

Конечные устройства удобно описывать наборами аппаратных компонентов, входящих в состав фитнес-браслета. По этой причине было введено понятие «продукт сборки» — минимально необходимый для корректной работы устройства набор платформно-зависимых архитектурных единиц. В процессе работы оказалось, что продуктом сборки можно описывать вспомогательные устройства, а также тестируемые сущности.

Например, с помощью продукта сборки без каких-либо аппаратных зависимостей логично тестировать алгоритмы и структуры данных, использующиеся в прошивке. Для тестирования связи с мобильным приложением нет смысла подключать дисплей, измерительный модуль с датчиками и прочие компоненты — нужен только коммуникационный модуль и BLE.

Ниже приведен пример вспомогательного продукта сборки для коммуникационного микропроцессора, который использовался для целей отладки и тестирования. С его помощью проверялась доставка данных из измерительного процессора в мобильное приложение.

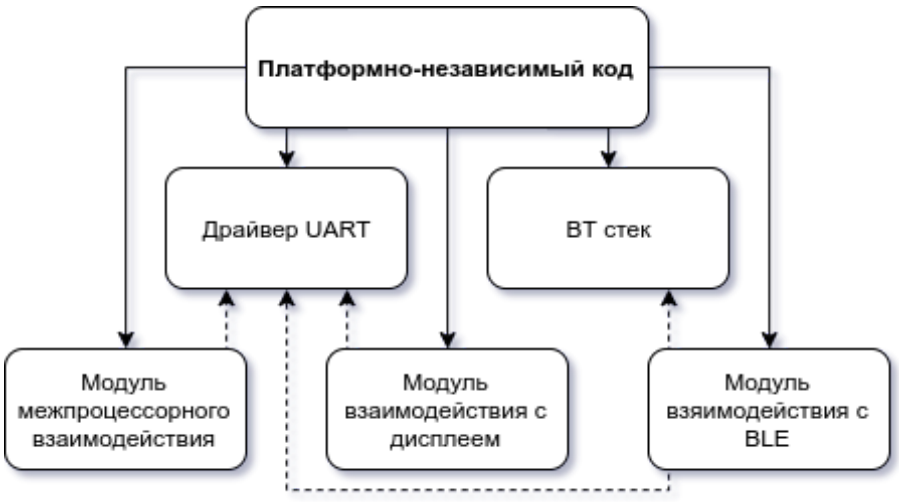

Рис. 2: Продукт сборки для коммуникационного микропроцессора.

Основными критериями выбора инструмента сборки были: простота масштабируемости, возможность использования как разработчиками, так и на сервере CI/CD, производительность, легкость поиска инструкций и опыт ключевых членов команды. Опираясь на таблицу 1, релевантными для целей проекта оказались: Meson, SCons и связка CMake + GNU make/Ninja.

Выбор был сделан в пользу CMake + Ninja по нескольким причинам. Во-первых, сборка прошивки для актуальных устройств занимает порядка 10 минут, это достаточно много, поэтому производительность важна. Во-вторых, для заказчика принципиальна сопровождаемость проекта своими силами, поэтому использование Meson или SCons было нецелесообразно в его случае. В-третьих, по экономическим соображениям было критично как можно быстрее наладить процесс, а CMake владели все ключевые разработчики. Ninja не требует времени на его изучение, так как работа с ним происходит исключительно через генератор, при этом он существенно производительней GNU make.

Для достижения конечной цели использовалась условная компиля-

ция и был введен ряд правил сопровождения кода для разработчиков.

Все использования аппаратно-зависимых функций оборачивались в директивы `#ifdef A... #endif`. Для каждого модуля, зависящего от платформы, создавался отдельный файл с расширением `*.cmake`, в котором указывается какое условие необходимо включить для сборки. Для таких файлов было введено понятие «feature». Такие файлы перечислялись списком в других `*.cmake` файлах, продуктах сборки или «product». Продукты сборки в свою очередь явно указывались в аргументах командной строки перед компиляцией.

4. Тестирующая система

В начале разработки прошивки наблюдался дефицит отладочных плат, поэтому не было возможности выделить целевую платформу только для тестирования — нужда разработчиков была в приоритете. По этой причине было важно начать налаживать тестирование без привлечения макетов конечных устройств.

Организация системы сборки тривиальным образом позволяла порождать вспомогательный продукт сборки, запускаемый на произвольных платформах, в частности на серверах компании. Было принято решение интегрировать одну из указанных в таблице 2 программных платформ.

Выбор был сделать в пользу Google Test, так как он широко применяется в других проектах компании и уже на тот момент имелся многолетний опыт работы с ним. Более того, аудитория данного инструмента в разы больше, нежели у остальных фреймворков, а значит вероятность наличия критических несовершенностей минимальна. Сильные стороны аналогов по сравнению с вышеизложенными фактами сочлись несущественными.

Рис. 3: Продукт сборки для тестирования платформно-независимого кода

Автоматизация процесса тестирования была достигнута с помощью средств Jenkins и GitLab. При загружении изменений в репозиторий с кодом, Jenkins проверяет компилируемость продуктов сборки, порождает соответствующие исполняемые файлы, запускает тестирование, уведомляет о проблемах.

5. Поиск некорректных состояний

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.

В результате экспериментов с символьным исполнением удалось обнаружить порядка пяти ошибок исполнения, связанных с неопределенным поведением, делением на ноль, переполнением типов.

Заключение

В рамках настоящей работы были достигнуты следующие результаты.

- В рамках обзора были рассмотрены основные подходы для тестирования и верификации встраиваемого ПО и выбраны инструменты, учитывающие особенности конечных устройств.
- Для фабрики прошивок, состоящей из нескольких сот тысяч строчек кода, на базе CMake спроектирована и реализована система сборки, которая позволяет гибко настраивать комплектацию итогового продукта в зависимости от аппаратных параметров целевого устройства.
- В контексте настройки платформно-зависимого тестирования на базе Robot Framework был разработан протокол взаимодействия с прошивкой на целевой платформе через отладочный последовательный порт. Введено в эксплуатацию платформно-независимое тестирование на базе фреймворка GTest.
- С помощью символьного интерпретатора KLEE были локализованы и устранены порядка пяти критических ошибок исполнения, связанных с переполнением типов и делением на ноль.

Список литературы