Организация автоматизированного тестирования встраиваемого программного обеспечения

Список авторов с электронными адресами, должностями/учебными статусами и организациями, например:

Кижнеров П. А., магистр кафедры системного программирования СПбГУ, p.a.kizhnerov@yandex.ru

Аннотация

В докладе приводится исследование способов организации тестирования и верификации встраиваемого программного обеспечения для фитнес-браслетов.

Введение

Разработка встраиваемого программного обеспечения предполагает ряд особенностей: малое количество вычислительных ресурсов, физические параметры каналов связи, время автономной работы, доступность прототипа устройства для разработчиков. Такие ограничения сильно усложняют поиск источников дефекта, повышают ценность выявления источников проблем, поэтому компании традиционно выделяют существенное количество ресурсов для тестирования программного обеспечения.

Дипломная работа выполнялась в контексте разработки прошивки для новой продуктовой линейки фитнес-браслетов западной компании. Коммерческая ценность - набор алгоритмов, позволяющих точно и автоматически измерять поступающие калории и гидратацию тела.

У актуальной линейки браслетов есть проблемы с архаичностью аппаратных компонентов, временем автономной работы и качеством кода некоторых модулей - поэтому было принято решение разрабатывать новую версию, уделяя особое внимание тестированию.

Система сборки проекта

Ввиду того, что конечных устройств несколько, а код для них отличается несущественно, было принято решение организовать фабрику прошивок.

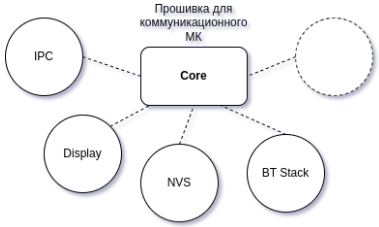

Рис. 1: Артефакт фабрики прошивок

В качестве инструмента сборки проекта был выбран CMake[1].

	Масштабируемость	Переносимость	Понятность/документированность	Опыт
CMake	+	+	+	+
Make	-	+	+	-
Atmel	+	-	+	+
SCons	?	+	+	-
Maven	?	+	-	+
Shake	?	+	-	-
Meson	+	+	+	-

Таблица 1: Сравнительная таблица инструментов сборки

Такая организация системы позволяет определять артефакты фабрики прошивок в зависимости от поставленных целей и задач. Им может быть прошивка для: бездисплейного устройства; тестирования под управлением инструментальной машины; модульного аппаратно-независимого тестирования и тд.

Тестирование

Исходный код имеет смысл разделять на аппаратно-зависимый и аппаратно-независимый. Для модульного и интеграционного тестирования аппаратно-независимого кода был GTest[2]. Для аппаратно-зависимого – Robot Framework[3].

		XML отчет	Переносимость	Опыт	Понятность	Доработки
АНЗ	**GTest**	+	+	+	+	–
	Catch2	+	+	–	+	–
	CxxTest	+	+	–	+	–
АЗ	**Robot FW**	+	+	–	+	+/–
	DejaGNU	+	–	–	–	+
	TETWare RT	+	+	–	–	+

Таблица 2: Сравнительная таблица инструментов тестирования

Управление процессом аппаратно-зависимого тестирования должно происходить с участием инструментальной машины, так как нет возможности установить фреймворк для тестирования непосредственно на саму целевую платформу. Это означает, что в любом случае придется дополнять решение для тестирования аппаратно-зависимого кода протоколом взаимодействия с целевой платформой.

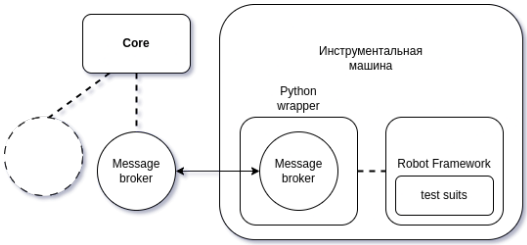

Рис. 2: Схема взаимодействия Robot Framework и прошивки

Традиционно для таких целей используется свободный последовательный порт, однако таких не оказалось, поэтому канал взаимодействия с прошивкой - последовательный порт отладки.

Верификация

Верифицировать программу - значит доказать, что в любой момент времени поток исполнения не достигает некорректного состояния. Техника символьного исполнения гарантирует, что поток исполнения посетит все ветви программы. Таким образом, если некорректное состояние искусственно поместить в отдельную ветвь программы и запустить символьный интерпретатор, то появляется возможность рассуждать о достижимости такого состояния, а значит и о корректности программы.

Для верификации узких мест прошивки необходимо добавить библиотеки символьного интерпретатора в артефакт фабрики прошивок, скомпилировать LLVM-биткод и подать его на вход символьному интерпретатору.

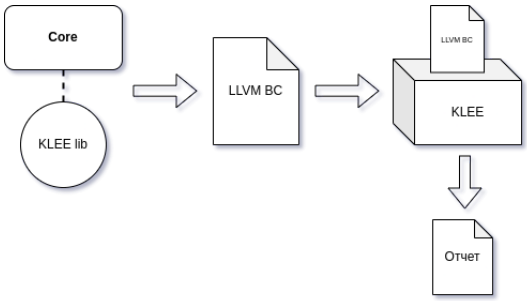

Рис. 3: Порядок использования символьного интерпретатора

Если не учитывать различные модификации, то в настоящий момент существует единственный работающий символьный интерпретатор для C/C++ – KLEE[4], который и используется в данной работе.

Заключение

В докладе был представлен подход, который позволил ввести в эксплуатацию систему автоматического тестирования. Также была задействована техника символьного исполнения, с помощью которой удалось обнаружить и устранить порядка пяти критических ошибок исполнения.

Список литературы

[1] CMake Tutorial // CMake URL: https://robotframework.org/ (дата https://medium.com/@onur.dundar1/cmake-tutorial-585dd180109b) (дата обращения: 27.08.2021)

[2] Тестирующий фреймворк от Google // GoogleTest URL: https://google.github.io/googletest/ (дата обращения: 15.09.2021)

[3] Фреймворк для интеграционных тестов Robot Framework // Robot Framework URL: https://robotframework.org/ (дата обращения: 20.10.2021)

[4] Символьный интерпретатор для C/C++ // KLEE URL: https://klee.github.io/ (дата обращения: 05.08.2022)