Аннотация

Автоматизированная система тестирования файловой системы JFFS2

Дементьев Даниил Викторович

Современные операционные системы состоят из множества модулей, реализующих основной функционал для пользователя. Одними из основных являются модули файловых систем, отвечающие за хранение и организацию доступа к пользовательским данным. По оценкам разработчиков, на подгружаемые модули приходится большинство ошибок, приводящих к некорректной работе всей ОС. По этой причине активно разрабатываются системы тестирования модулей ядра Linux с использованием различных подходов к тестированию и анализу кода. В данной работе рассмотрена проблема тестирования файловой системы ядра Linux JFFS2 и описан процесс разработки системы автоматизированного тестирования исследуемой ФС на базе набора регрессионных тестов для различных файловых систем xfstests. В работе рассмотрены основные подходы к тестированию модулей ядра, а также использованы соответствующие различным подходам инструменты. Разработанная система покрывает 79.4% строк исходного кода JFFS2 и 91.4% функций. С целью дальнейшего развития разработанной системы в работе проведен анализ непокрытого кода.

Содержание

1 Введение **3**
1.1 Файловая система . . . 3
1.2 Устройства долговременного хранения информации . . . 3
1.3 Взаимодействие ФС с флэш-памятью . . . 4
1.4 Устройства на основе технологии памяти . . . 4
1.5 Устройства с несортированными блоками . . . 5
1.6 Файловая система JFFS2 . . . 6
1.7 Тестирования файловых систем . . . 7
1.8 Симуляция сбоев . . . 8
1.9 Фаззинг-тестирование . . . 8
1.10 Характеристика систем тестирования . . . 8

2 Постановка задачи **9**

3 Обзор существующих решений **10**
3.1 Xfstests . . . 10
3.2 Fsfuzz . . . 10
3.3 Syzkaller . . . 10
3.4 Тесты из пакета инструментов MTD . . . 11

4 Исследование и построение решения задачи **12**
4.1 Тестовое покрытие . . . 12
4.2 Расширение тестового покрытия ФС . . . 12
4.3 Функционал JFFS2 . . . 12
4.4 Окружение файловой системы . . . 14
4.5 Создание системы автоматизированного тестирования . . . 15

5 Описание практической части **16**
5.1 Использованные инструменты . . . 16
5.2 Тестирование с использованием готовых решений . . . 19
5.3 Результаты первого эксперимента . . . 19
5.4 Расширение тестового покрытия . . . 20
5.5 Расширенное тестовое покрытие . . . 21
5.6 Фаззинг-тестирование . . . 22
5.7 Тестирование с использованием симуляции сбоев . . . 24
5.8 Анализ непокрытого кода . . . 26

6 Заключение **29**

Приложение А **33**

Приложение Б **34**

1 Введение

1.1 Файловая система

Одна из основных функций операционной системы (ОС) - управление устройствами долговременного хранения информации и предоставление пользовательским приложениям интерфейса для взаимодействия с ними с целью хранения и организации доступа к данным. Элемент операционной системы, отвечающий за эти функции - **файловая система** (ФС) [1].

Для интеграции ряда файловых систем в единую структуру с пользовательской точки зрения в ядре ОС Linux используется концепция виртуальной файловой системы, позволяющая обращаться к различным файловым системам, используя единый интерфейс системных вызовов. Для оптимизации своей работы ФС использует множество внутренних структур данных. Для хранения метаданных пользовательских файлов и каталогов ФС использует структуру данных, называемую **индексным дескриптором**. Также для хранения информации о свободном пространстве на устройстве файловые системы используют особые структуры данных, например, битовую карту данных или список свободных блоков устройства. Для оптимизации процесса монтирования вся информация о файловой системе необходимая ОС в процессе монтирования хранится в структуре, называемой **суперблоком**.

1.2 Устройства долговременного хранения информации

Большинство файловых систем ядра Linux были разработаны для взаимодействия с накопителями на жестком магнитном диске, особенности строения которых формировали алгоритмы хранения данных, используемые файловыми системами. Подобные устройства хранения состоят из вращающегося диска с магнитными дорожками и считывающей головки, способной перемещаться между дорожками. Из-за подобного строения доступ к произвольному блоку данных на жестком диске имел задержку позиционирования считывающей головки над нужной дорожкой и задержку вращения диска, обусловленную скоростью вращения магнитного диска. Для оптимизации задержки вращения в файловых системах использовались различные алгоритмы размещения данных на диске, например, размещение файлов из одного каталога и их индексных дескрипторов поблизости.

С развитием технологий на рынке устройств долговременного хранения

информации начали появляться аналоги жестким дискам в виде **устройств флэш-памяти** [2]. Устройства флэш-памяти состоят из транзисторов, за счет чего скорость доступа к произвольному блоку имеет тот же порядок, что и скорость последовательного доступа к данным. Ячейки флэш-памяти организованы в так называемые **стираемые блоки**, которые обычно имеют размер 128 килобайт. Интерфейс взаимодействия с устройствами флэш-памяти отличается от интерфейса жесткого диска и составляет три основные команды: чтение данных из блока, стирание всего блока, то есть установка всех его бит в единицу, и программирование памяти, то есть замена части единичных бит на нули для записи нужных данных, что может быть осуществлено после стирания блока.

Главная проблема флэш-памяти - износ. В процессе стирания и перезаписи блока на транзисторах накапливается избыточный заряд, из-за чего становится невозможно определить значение на транзисторе, и блок становится непригодным к использованию. Время жизни блоков флэш-памяти измеряется в циклах стирания и имеет значение порядка 100000 циклов. Для предотвращения излишнего использования определенных блоков и продления срока службы всей микросхемы используется технология **выравнивания износа** блоков, что означает равномерное использование всех блоков.

1.3 Взаимодействие ФС с флэш-памятью

Для эффективного использования флэш-памяти в качестве запоминающего устройства необходимо разрешить несколько основных проблем: выравнивание износа, обеспечение надежности в случае отключения питания, сборка мусора и необходимость программной обработки дефектных блоков. Эти проблемы решаются **уровнем флэш-преобразования** (FTL).

Уровень флэш-преобразования может быть размещен в контроллере устройства памяти, в драйвере устройства или в файловой системе, как это реализовано в JFFS2 и других ФС, взаимодействующих с неуправляемой флэш-памятью, например, UBIFS и YAFFS2.

1.4 Устройства на основе технологии памяти

Для доступа к неуправляемой флэш-памяти, то есть не имеющей контроллера выполняющего функции FTL, в ядре Linux используется система устройств на основе **технологии памяти** (MTD - Memory Technology

Device) [3], предоставляющая базовые интерфейсы для взаимодействия с устройствами, а также обработки и идентификации дефектных блоков. Структура подсистемы MTD проиллюстрирована на рис. 1.

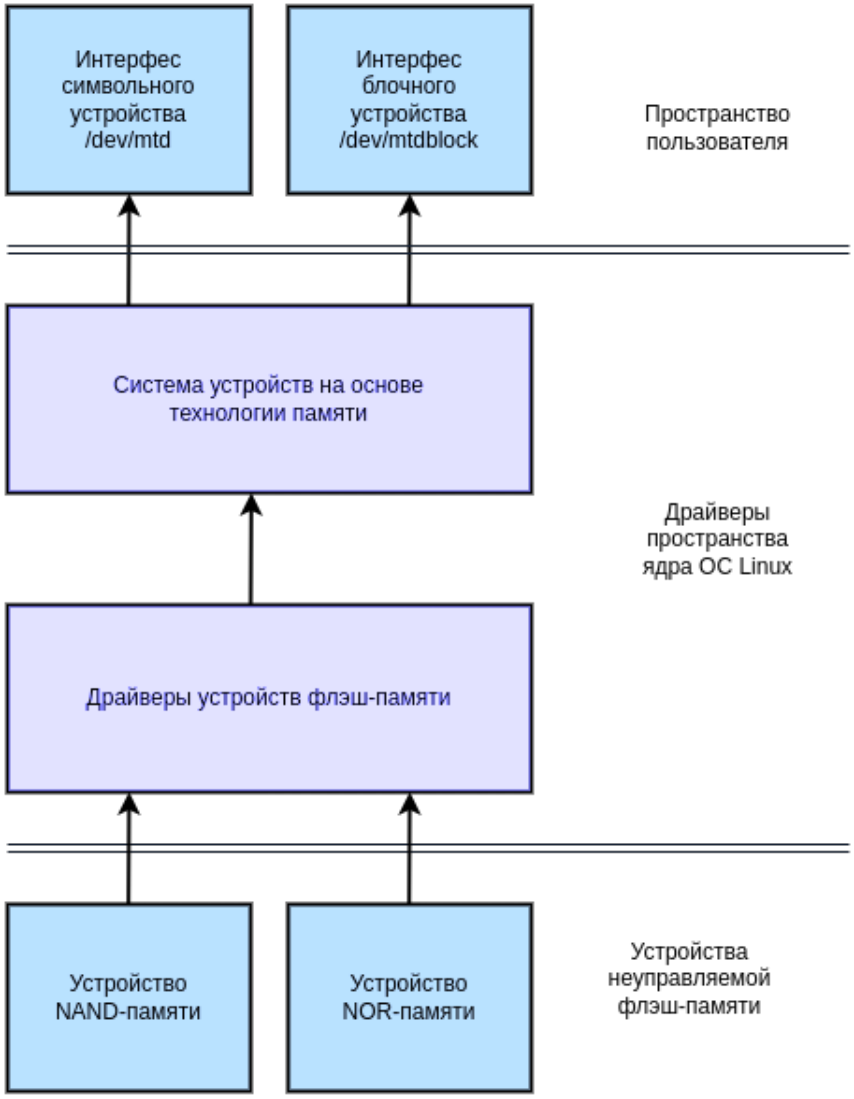

Рис. 1: Подсистема устройств технологии памяти (MTD)

Она состоит из трех уровней: основной набор функций, драйверы микросхем флэш-памяти и драйверы пользовательского уровня, предоставляющие пользователю интерфейс символьного и блочного устройства для взаимодействия с флэш-памятью.

1.5 Устройства с несортированными блоками

Подсистема **образов с несортированными блоками** (UBI - Unsorted Block Images) является частью модуля MTD. UBI предоставляет механизм абстракции над устройствами технологии памяти в виде томов, состоящих из логических стираемых блоков. Каждому логическому блоку соответствует физический стираемый блок устройства, при том соседние логиче-

ские блоки необязательно будут связаны с соседними физическими блоками устройства. Помимо возможности разделения устройства на несколько томов, такая абстракция помогает избавить файловую систему от проблем обработки дефектных блоков устройства. Дефектные блоки находятся подсистемой UBI и не включаются в томы. Механизм работы UBI продемонстрирован на рис. 2.

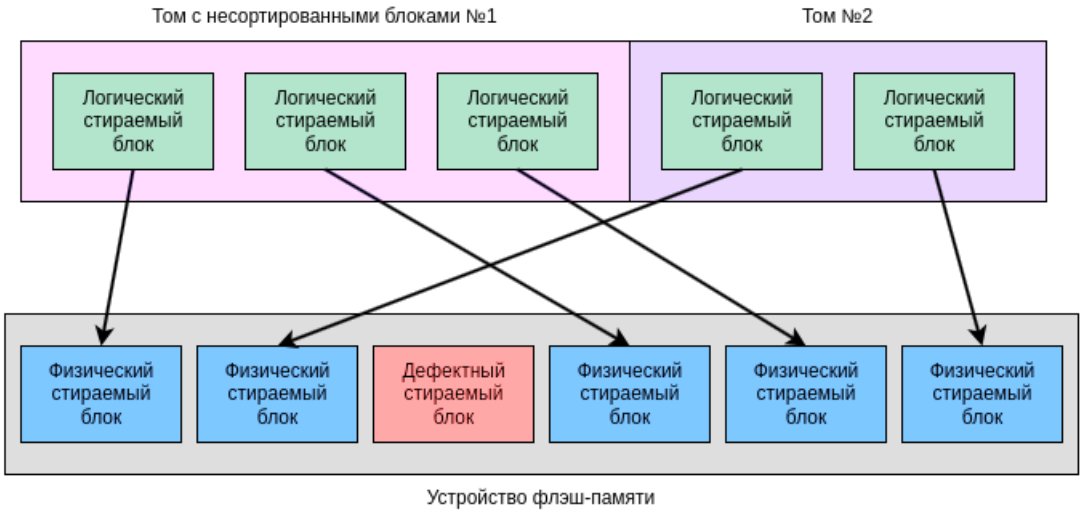

Рис. 2: Томы с несортированными блоками (UBI)

1.6 Файловая система JFFS2

Файловая система JFFS2 (Journaling Flash Filesystem 2) [4] была разработана для использования неуправляемых микросхем флэш-памяти в качестве запоминающего устройства и поддерживает устройства как NOR, так и NAND типа. JFFS2 является исторически первой файловой системой Linux для флэш-накопителей, но используется до сих пор [2]. Она имеет журнальную структуру и использует подсистему устройств технологий памяти MTD для доступа к флэш-памяти.

Для того, чтобы избежать чтения всего журнала для вычисления структуры каталогов на этапе монтирования, JFFS2 использует сводные узлы (summary node). Сводный узел записывается в конец открытого стираемого блока и содержит в себе всю информацию, необходимую на этапе монтирования. Такой подход позволяет уменьшить время монтирования ценой небольшого перерасхода объема используемой памяти. Этот режим включается при установке определенного конфигурационного параметра ядра.

С целью различить, является ли стираемый блок, в котором все биты установлены в 1, очищенным, или он хранит полезную информацию, состоящую из всех единичных бит, JFFS2 использует маркеры очистки (clean

marker), записываемые в начале блока. Если маркер очистки присутствует в блоке, значит блок очищен.

1.7 Тестирования файловых систем

Обеспечение надежности и отказоустойчивости модулей ядра, и файловой системы в частности - важная и непростая задача [5]. Наличие ошибок в исходном коде ФС может привести к нарушениям в основных функциях файловой системы, получению пользователем несанкционированного доступа к привилегированному режиму ядра, а также полному краху или блокировке системы. Подобные последствия ошибок на серверах и суперкомпьютерах могут привести к потерям значимых данных или несанкционированному доступу к ним.

Большинство веб-серверов и суперкомпьютеров используют операционные системы на основе ядра Linux, имеющего открытый исходный код. В ядре Linux имеется множество файловых систем, разработанных для различных типов устройств хранения информации и использующие различные алгоритмы расположения данных на устройстве.

Решением для проверки корректности функционирования операционных систем и их модулей является создание систем автоматизированного тестирования.

Системы автоматизированного тестирования файловых систем должны тестировать код на наличие ошибок разного характера, принимая во внимание особенности архитектуры тестируемого модуля. Например, код файловых систем, и JFFS2 в частности, содержит функции, выполняющиеся параллельно, которые в случае некорректной работы могут привести к конкуренции за данные (data races) или взаимной блокировке параллельно исполняющихся функций (deadlock).

Стандартный подход к тестированию ФС заключается в создании тестов пространства пользователя с детерминированным результатом. Такие тесты реализуются в виде последовательности системных вызовов для взаимодействия с функционалом ФС. Такой подход к тестированию ФС является наиболее популярным и хорошо подходит для обнаружения дефектов в участках ФС, отвечающих за стандартный сценарий работы модуля.

1.8 Симуляция сбоев

Исходный код любого модуля ядра, и в частности ФС, содержит множество строк кода, обрабатывающих редкие сценарии ошибок. Для тестирования подобного кода применяют специальные подходы, например, тестирование с использованием **симуляции сбоев** (fault injection) [6].

С использованием этого подхода, в вызываемых функциях симулируются условия, вызывающие ошибки в процессе их исполнения. С точки зрения вызывающей функции сценарий ее работы выглядит, как если бы вызываемая функция вернула код ошибки в случае нехватки памяти, обнаружении невалидных данных или невозможности доступа к запоминающему устройству.

1.9 Фаззинг-тестирование

Другой популярный подход к тестированию работы модулей ядра в редких сценариях работы — **фаззинг-тестирование**. Этот подход применяется с целью намеренно вызвать сбои и достичь редких сценариев работы тестируемой программы. Суть работы инструментов фаззинг-тестирования заключается в генерации случайных данных, передаваемых в качестве параметров при взаимодействии с интерфейсом тестируемого ПО. В случае тестирования модулей ядра ОС инструменты фаззинг-тестирования генерируют случайные входные данные для системных вызовов.

1.10 Характеристика систем тестирования

Важнейшей метрикой, характеризующей качество подобных систем тестирования является **тестовое покрытие** исходного кода. Полнота тестового покрытия кода является предметом для множества исследований, в том числе в области тестирования файловых систем [7]. В частности, в данной работе будет рассмотрена проблема создания системы автоматизированного тестирования файловой системы JFFS2, а также исследование достигаемого тестового покрытия исходного кода этой файловой системы.

Анализ достигаемого тестового покрытия - неотъемлемая часть создания системы тестов. Разработчику инструмента тестирования важно понимать, какой функционал программы хорошо тестируется разрабатываемыми тестами, и какие участки исходного кода используются малое число раз и требуют дальнейшей разработки инструмента.

2 Постановка задачи

Целью данной работы является разработка системы автоматизированного тестирования файловой системы JFFS2.

Для достижения данной цели были поставлены следующие задачи:

- Изучить особенности внутреннего устройства файловой системы JFFS2.
- Исследовать существующие решения тестирования файловых систем и модулей ядра Linux.
- Расширить тестовое покрытие кода, достигаемое существующими решениями.
- Разработать тесты, предназначенные для проверки особенностей JFFS2.
- Провести анализ достигаемого системой тестового покрытия для дальнейшего ее развития.

Актуальность данной работы обусловлена тем, что инструменты тестирования ядра Linux в лаборатории Linux Verification Center [8] часто обнаруживают в исходном коде JFFS2 ошибки, требующие исправлений. Для проверки корректности внесенных исправлений требуется система тестирования, специализированная под особенности этой ФС и покрывающая своими тестами основной функционал.

JFFS2 в данный момент не имеет тестовых наборов, специализированных под ее архитектурные особенности ввиду небольшой популярности. Однако эта ФС до сих пор используется на множестве устройств [2].

3 Обзор существующих решений

Для большинства файловых систем ядра Linux сообществом были разработаны тестовые наборы, фреймворки стресс-тестирования и инструменты фаззинг-тестирования. В данной главе будут описаны наиболее популярные и подходящие к тестированию JFFS2 решения.

3.1 Xfstests

Xfstests [9] - набор регрессионных тестов, изначально разработанный под файловую систему xfs, но в данный момент поддерживающий множество других файловых систем ядра Linux, среди которых btrfs, reiserfs, f2fs и другие. Этот набор представляет собой множество bash-скриптов, разделенных по каталогам на группы. Среди тестов имеются как нацеленные на тестирование общих для всех ФС функций, так и предназначенные для определенных систем. На данный момент xfstests является наиболее популярным и широко использующимся решением для тестирования файловых систем.

3.2 Fsfuzz

Fsfuzz [10]- это инструмент фаззинг-тестирования ФС, представляющий собой bash-скрипт, суть работы которого заключается в генерации множества случайных образов выбранной файловой системы и дальнейшего монтирования этих образов. Подобный метод перебора не покрывает все основные функции файловой системы, однако может привести к редким сценариям обработчика ошибок, что полезно, так как подобные редкие сценарии сложно покрыть обычными тестовыми наборами.

3.3 Syzkaller

Syzkaller [11] - наиболее популярный инструмент динамического анализа кода ядра ОС. Суть его работы заключается в конструировании случайных программ на основе системных вызовов, запуске их на тестируемой системе и генерации обратной связи в виде покрытия кода ядра. Случайные программы конструируются с использованием различных системных вызовов в различном порядке и со случайно подобранными аргументами с целью расширения покрытия кода. При достижении ошибок syzkaller генерирует программы языке Си повторяющие сценарий достижения ошибки, которые называют программами **репродьюсерами**.

В файле конфигурации syzkaller можно определить список разрешенных к использованию системных вызовов, таким образом ограничить его покрытие, но нацелить на определенные сценарии с выбранными системными вызовами. При должной конфигурации syzkaller может быть использован для тестирования определенного модуля ядра, в том числе и ФС.

3.4 Тесты из пакета инструментов MTD

Пакет **mtd-utils** [12] содержит множество инструментов для работы с устройствами флэш-памяти, в том числе и инструмент создания образа файловой системы JFFS2. Разработчиками этого пакета инструментов были добавлены тесты для файловых систем флэш-памяти в виде четырех bash-скриптов.

Эти тесты должны хорошо покрывать функции, отвечающие особенностям файловой системы JFFS2, однако такого небольшого числа скриптов недостаточно для покрытия всех основных функций ФС.

4 Исследование и построение решения задачи

В данной главе будут подробно рассмотрены идеи, разработанные в ходе исследования и построения решения задачи. Реализации данных идей подробно описывается в следующей главе.

4.1 Тестовое покрытие

Тестовое покрытие является основной метрикой, характеризующей качество системы тестирования любой программы. Однако функционал модулей ядра операционной системы, и ФС в частности, во многом зависит от конфигурации окружения, в котором она работает. То есть для получения полной информации о покрытии достигаемом при помощи тестового набора необходимо провести несколько экспериментов в различном окружении.

4.2 Расширение тестового покрытия ФС

Для расширения тестового покрытия исходного кода ФС, достигаемого при помощи тестового набора, необходимо провести анализ основных функций тестируемой ФС и параметров конфигурации ее окружения, от которых зависит ее функционал.

4.3 Функционал JFFS2

Основные функции файловой системы JFFS2 можно разделить на несколько групп по их предназначению:

1. **Основные операции с файлами.**

   К этой группе относятся функции, отвечающие за основной функционал операций с файлами, каталогами и внутренними структурами ФС. Этот базовый функционал является общим для всех ФС. К таким операциям относятся чтение и запись данных в файл, создание и удаление файла или каталога и соответствующих им индексных дескрипторов, установка расширенных атрибутов файлов. Функции этой группы могут быть покрыты тестовыми наборами других ФС, так как являются частью общего для всей файловых систем функционала.

2. **Функции контроля доступа.**

К этой группе относятся функции, отвечающие за контроль доступа к пользовательским данным и операции с внутренними структурами, отвечающими за это, например, списком контроля доступа.

3. **Функции взаимодействия с устройствами хранения.**

   К этой группе относятся функции, определяющие тип устройства хранения с которым работает ФС и работающие с его блоками, например, обеспечивающие выравнивание износа стираемых блоков.

4. **Сборщик мусора.**

   Эта группа включает в себя функции, обеспечивающие работу алгоритма сборщика мусора JFFS2. Этот алгоритм является специфичным для этой ФС и его функции могут не покрываться при стандартных сценариях.

5. **Алгоритмы сжатия данных.**

   В JFFS2 есть 4 различных алгоритма сжатия данных, состоящих из множества внутренних функций. Помимо функционала самих алгоритмов, в коде присутствуют функции, отвечающие за стратегию выбора алгоритма сжатия. Например, выбор может производиться по приоритету, размеру сжатых данных или вынуждая ФС в любом случае использовать один алгоритм.

6. **Обработчики ошибок.**

   К этой группе относятся функции, обрабатывающие ошибки и редко достигаемые в обычных сценариях работы ФС. Эти ошибки могут быть связаны с некорректным выделением памяти, несоответствием важных параметров образа и устройства, используемого для его подключения или невалидностью данных, по причине наличия ошибок во внутренней логике самой файловой системы. Функции этой группы лучше всего покрываются при помощи **фаззинг-тестирования**, либо тестированием с **симуляцией сбоев**.

7. **Обработка входных параметров команд.**

   К этой группе относятся функции, обрабатывающие суперблок ФС и входные параметры при выполнении команды монтирования. Эти функции являются частью основного функционала любой ФС, однако могут быть выделены в отдельную группу, так как лучше покрывают-

ся фаззинг-тестированием ввиду большой вариативности возможных входных параметров и образа ФС.

8. **Функции работы с внутренними структурами.**

   К этой группе относятся функции, работающие со внутренними структурами, особенными для исследуемой ФС. Для JFFS2 такими структурами являются, например, сводные узлы и маркеры очистки блоков. Ввиду своего назначения, эти функции могут не покрываться обычными тестовыми наборами и требовать написания специализированных тестов.

4.4 Окружение файловой системы

Функционал файловой системы зависит от множества параметров ее окружения. В данном разделе будут выделены факторы, от которых зависит функционал JFFS2.

Во-первых, для различных типов запоминающих устройств файловая система может использовать разные функции для работы с данными. Файловая система jffs2 по разному работает с устройствами технологии памяти типа NOR и NAND, а также имеет особый интерфейс взаимодействия с образами с несортированными блоками (UBI) [13] и устройствами флэш-памяти dataflash, подключаемыми по шине SPI.

Во-вторых, при тестировании заранее созданного образа ФС, тестируемый образ может быть создан с различными опциями. Для файловой системы JFFS2 опции команды mkfs, создающей образ ФС, могут быть использованы для задания размера выходного образа, размера стираемого блока, размера страницы, используемого алгоритма сжатия данных, используемой стратегии автоматического выбора алгоритма сжатия, используемого порядка бит в выходном образе и подключения, либо отключения функционала расширенных атрибутов файлов Linux.

В-третьих, при тестировании команды монтирования, используемой для подключения файловой системы к единому дереву каталогов, могут быть указаны различные опции. При помощи опций общих для всех файловых систем можно устанавливать режим доступа к данным, содержащимся на подключаемом устройстве, например, доступ только для чтения, либо для чтения и записи. С использованием опций специфичных для JFFS2 можно, например, выбрать используемый алгоритм сжатия данных.

В-четвертых, функционал собранного ядра операционной системы во

многом определяется составленным на этапе сборки файлом конфигурации. Опции, определяемые в этом файле могут отвечать за подключение определенных модулей ядра и указание параметров для этих модулей. Для модуля устройств на основе технологии памяти в конфигурационном файле могут быть указаны опции, отвечающие за подключение различных эмуляторов и определение их параметров, поддержку драйверов нужных для взаимодействия с различными чипами и контроллерами устройств памяти. Для файловой системы JFFS2 при помощи опций конфигурационного файла подключается поддержка расширенных атрибутов Linux, поддержка буфера записи, использование структуры сводных узлов и маркеров очистки и поддержка различных алгоритмов сжатия. Также при помощи конфигурационных опций может быть выбрана стратегия автоматического выбора алгоритма сжатия. Например, строка "CONFIG_JFFS2_CMODE_PRIORITY=y" в файле конфигурации ядра указывает файловой системе выбирать алгоритм сжатия на основе их приоритета, а строка "CONFIG_JFFS2_CMODE_SIZE=y" указывает на выбор алгоритма сжатия на основе размера сжатых данных.

4.5 Создание системы автоматизированного тестирования

В соответствии с целью и поставленными в работе задачами, для создания системы автоматизированного тестирования файловой системы JFFS2 необходимо протестировать исследуемую ФС при помощи имеющихся решений, в процессе доработать или адаптировать необходимые инструменты, расширить покрытие, достигаемое при использовании имеющихся решений, произведя запуск в различных конфигурациях окружения, протестировать функции обработки ошибок при помощи инструментов фаззинг-тестирования, и создать тесты, покрывающие редкие сценарии, с использованием технологии внедрения ошибок.

5 Описание практической части

5.1 Использованные инструменты

В данном разделе будет приведен список инструментов, используемых в ходе практической части работы.

В ходе всей работы по тестированию JFFS2 используется код ядра Linux версии 5.10 из репозитория [14] лаборатории Linux Verification Center. Процесс тестирования происходит на виртуальной машине Qemu, эмулирующей машину архитектуры x86-64 под гостевой системой Ubuntu Linux.

Для тестирования сценариев работы с различными типами устройств флэш-памяти без использования реальных микросхем памяти, необходимо использовать эмуляторы флэш-устройств. На данный момент в модуле устройств на основе технологии памяти (MTD) [3] ядра Linux реализованы 3 эмулятора: mtdram, эмулирующий NOR-память в оперативной памяти системы, block2mtd, эмулирующий устройство NOR-памяти поверх блочного устройства и nandsim, эмулирующий устройство NAND-памяти в оперативной памяти. Поддержка эмуляторов подключается в файле конфигурации ядра, например, параметр "CONFIG_MTD_MTDRAM" отвечает за использование эмулятор mtdram, а параметры "CONFIG_MTDRAM _TOTAL_SIZE" и "CONFIG_MTDRAM_ERASE_SIZE" устанавливают размер эмулируемого устройства и размер его стираемого блока.

За основу при создании системы автоматизированного тестирования было решено взять тестовый набор xfstests, так как он содержит в себе множество скриптов, тестирующих базовые функции, общие для всех ФС, а также является наиболее популярным решением для тестирования файловых систем ядра Linux и имеет удобный функционал для добавления новых тестов. Для добавления в набор xfstests своих тестовых сценариев необходимо описать сценарий в виде bash-скрипта, запустить скрипт new для добавления нового теста в набор и создать файл, определяющий выходные данные, которые должны генерироваться в ходе корректной работы теста. При тестировании xfstests использует 2 блочных устройства, называющиеся TEST и SCRATCH с тестируемой ФС на них и два каталога, используемые для монтирования. Для предоставления тестовому набору интерфейса блочного устройства в модуле MTD есть драйвер mtdblock. Для использования этого драйвера необходимо установить соответствующую ему опцию конфигурации ядра, что делается при помощи строки "CONFIG_MTD_BLOCK=y".

Для сбора информации о покрытии исходного кода модулей ядра используется утилита gcov [15]. Для визуализации собранной информации в виде html отчета используется расширение утилиты gcov - lcov [16]. Для сбора информации о покрытии в ядре Linux достаточно установить параметр конфигурации ядра "CONFIG_GCOV_KERNEL" при его сборке, а так же параметр "GCOV_PROFILE" в файле сборки модуля fs/jffs2/Makefile для активации профилирования директории,содержащей исходные файлы тестируемой ФС. Для сбора корректного покрытия в ходе работы тестов перед запуском необходимо сбросить информацию о покрытии. Сброс покрытия можно осуществить командой "echo 0 > /sys/kernel/debug/gcov/reset". После окончания работы тестов, информация о покрытии хранится в директории /sys/kernel/debug/gcov. На основе собранного покрытия утилита lcov генерирует информационный файл, который преобразуется в html-отчет утилитой genhtml. Пример отчета о собранном покрытии представлен на рис. 3

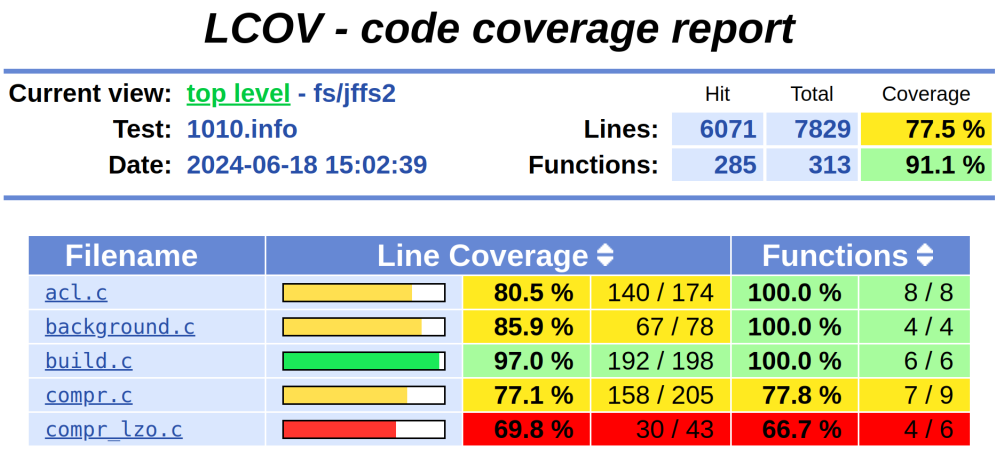

Рис. 3: HTML отчет о покрытии, сгенерированный утилитой genhtml.

Для создания тестируемого образа файловой системы используется команда mkfs с опцией определяющей тип файловой системы. Команда для создания образа JFFS2 (mkfs.jffs2) является частью пакета инструментов для работы с флэш-памятью mtd-utils [12]. По неизвестной причине, из четырех алгоритмов сжатия, поддерживаемых JFFS2, команда создания образа файловой системы поддерживала только 3: lzo, zlib и rtime. С целью достичь более полного тестирования исходного кода, в процессе работы исходный код инструмента mkfs.jffs2 был доработан, и таким образом была добавлена возможность создания образа JFFS2 с использованием алгоритма сжатия rubin.

Как было описано в прошлой главе, JFFS2 имеет специальные функции

для работы с устройствами dataflash, подключенными по протоколу SPI и устройствами разделенными на томы с несортированными блоками (UBI устройства) [13]. К сожалению в открытом доступе нет эмулятора устройства dataflash, для тестирования сценария монтирования устройства такого типа, однако в модуле MTD есть эмулятор устройств с несортированными блоками - gluebi. Для подключения этого эмулятора в конфигурационном файле ядра используется строка "CONFIG_MTD_UBI_GLUEBI=y". Подсистема UBI, являющаяся частью модуля MTD выполняет функции уровня флэш-преобразования, обеспечивая выравнивание износа блоков устройства, а также управляет томами, являющимися более высокоуровневой абстракцией над устройствами технологии памяти. В процессе практической работы, эмулятор gluebi вызывал панику ядра путем разыменования нулевого указателя. Для корректного использования эмулятора потребовалось исправление его исходного кода.

Для покрытия функций обработки ошибок фаззинг-тестированием, в работе будут использованы инструменты syzkaller [11] и fsfuzz [10]. Не смотря на то, что в документации инструмент фаззинг-тестирования файловых систем fsfuzz заявляется о совместимости с файловой системой JFFS2, прямолинейный запуск инструмента для исследуемой ФС не принес результата, так как bash-скрипт fsfuzz использует при монтировании виртуальные блочные устройства, что является неподходящим для JFFS2. После внесения правок в исходный код инструмента с целью использования MTD устройств при указании в качестве опции тип файловой системы JFFS2, инструмент удалось применить к тестированию исследуемой ФС.

Для симуляции сбоев во внутренних функциях модуля ФС в работе будет применен встроенный в ядро Linux инструмент Linux Fault Injection [17]. Этот инструмент позволяет симулировать сбои почти всех функций ядра, предоставляя пользователю возможность установить в качестве параметров вероятность ошибки, интервал между сбоями, их количество и возвращаемое функцией значение. Для симуляции сбоя функции необходимо разрешить в коде ядра внедрение неисправности в эту функцию при помощи макроса ALLOW_ERROR_INJECTION. Сам инструмент можно использовать при установке параметра конфигурации ядра "CONFIG_FAULT_INJECTION".

5.2 Тестирование с использованием готовых решений

Для тестирования файловой системы с использованием готовых решений необходима предварительная сборка ядра с определенной конфигурацией, наличие тестируемого образа файловой системы и блочных устройств для монтирования. Далее в этом разделе будет описана базовая конфигурация, необходимая для тестирования JFFS2 при помощи готовых решений.

Во первых, необходимо в конфигурации ядра добавить поддержку тестируемой ФС и необходимой для ее работы подсистемы устройств технологии памяти, которая отсутствует в базовой конфигурации ядра. Это может быть осуществлено добавлением следующих строк в файл конфигурации ядра Linux: "CONFIG_MTD=y" - подключение подсистемы MTD, "CONFIG_JFFS2_FS=y" - подключение модуля JFFS2. Во-вторых, после конфигурации и сборки ядра необходимо создать образ тестируемой ФС. Для этого используется команда mkfs.jffs2, являющаяся частью пакета инструментов mtd-utils, который требует предварительной установки, например, при помощи менеджера пакетов apt-get. В третьих, для монтирования тестируемой ФС необходимы блочные устройства, которые в случае JFFS2 создаются драйвером mtdblock поверх MTD устройств эмулируемых одним из эмуляторов, например, block2mtd, который тоже необходимо подключить в конфигурации ядра.

5.3 Результаты первого эксперимента

Прямолинейный запуск набора xfstests в совокупности с тестами из пакета mtd-utils в описаной выше минимальной необходимой конфигурации дал результат в виде тестового покрытия исходного кода в 64% исходного кода.

	Hit	Total	Coverage
Lines:	5009	7822	64.0 %
Functions:	215	313	68.7 %

Рис. 4: Результат первого эксперимента

Следующим важным шагом является расширение тестового покрытия, на основе анализа функционала JFFS2 и параметров окружения, который был приведен в прошлой главе, а также анализа достигнутого в первом эксперименте покрытия.

5.4 Расширение тестового покрытия

Первым шагом на пути к расширению тестового покрытия является добавление в конфигурации ядра всего функционала тестируемой ФС. Опции конфигурации JFFS2 и функционал на который они влияют описаны в файле fs/jffs2/Kconfig. Опции конфигурации и их назначение описаны в таблице на рис. 5.

Параметр конфигурации модуля JFFS2	Описание
CONFIG_JFFS2_FS	Подключение модуля JFFS2
CONFIG_JFFS2_FS_DEBUG	Регулирует количество генерируемых сообщений (используется в процессе исправления кода)
CONFIG_JFFS2_FS_WRITEBUFFER	Подключение поддержки буфера записи
CONFIG_JFFS2_FS_WBUF_VERIFY	Подключение дополнительной проверки после записи в блок данных через буфер записи
CONFIG_JFFS2_SUMMARY	Подключение использования сводных узлов
CONFIG_JFFS2_FS_XATTR	Подключение поддержки расширенных атрибутов Linux
CONFIG_JFFS2_FS_POSIX_ACL	Подключение поддержки списка контроля доступа
CONFIG_JFFS2_FS_SECURITY	Подключение поддержки альтернативных способов контроля доступа, например SELinux

Рис. 5: Опции конфигурации ядра для сборки модуля JFFS2

Анализ достигнутого при первом эксперименте покрытия показал, что не покрыты функции, используемые для работы с устройствами NAND-памяти и UBI томами, так как используемые эмулятор block2mtd эмулирован флэш-память NOR-типа. Для расширения тестового покрытия необходимо использовать эмулятор UBI томов gluebi и NAND-памяти nandsim.

Параметр конфигурации модуля JFFS2	Описание
CONFIG_JFFS2_COMPRESSION_OPTIONS	Подключение возможности выбора алгоритма сжатия данных
CONFIG_JFFS2_ZLIB	Подключение поддержки алгоритма сжатия zlib
CONFIG_JFFS2_LZO	Подключение поддержки алгоритма сжатия lzo
CONFIG_JFFS2_RTIME	Подключение поддержки алгоритма сжатия rtime
CONFIG_JFFS2_RUBIN	Подключение поддержки алгоритма сжатия rubin
CONFIG_JFFS2_CMODE_NONE	Отключение использования сжатия
CONFIG_JFFS2_CMODE_PRIORITY	Алгоритм сжатия будет выбираться по приоритету, определенному в коде
CONFIG_JFFS2_CMODE_SIZE	Алгоритм сжатия будет выбираться по размеру сжатых данных
CONFIG_JFFS2_CMODE_FAVOURLZO	Всегда будет выбираться алгоритм сжатия lzo

Рис. 6: Опции конфигурации ядра для сборки модуля JFFS2 (Продолжение)

Также не были покрыты функции алгоритмов сжатия данных. Для их покрытия необходимо подключить их поддержку в конфигурации ядра, создать несколько тестируемых образов с разными параметрами mkfs, например, команда "mkfs.jffs2 compression-mode=size" создает образ ФС, алгоритм сжатия на котором будет определяться размерам сжатых данных. Для использования каждого конкретного алгоритма сжатия можно установить режим выбора алгоритма на выбор по приоритету и устанавливать приоритет тестируемого алгоритма выше остальных параметрами команды mkfs. Например, команда "mkfs.jffs2 compression-priority=100:zlib" устанавливает максимальный приоритет алгоритму zlib.

5.5 Расширенное тестовое покрытие

Тестовые наборы были перезапущены несколько раз с различными конфигурациями. Покрытия полученные в этих экспериментах, объединенные с результатом первого эксперимента дали результат в виде тестового покрытия в 75.8%.

	Hit	Total	Coverage
Lines:	6284	8295	**75.8 %**
Functions:	299	338	**88.5 %**

Рис. 7: Расширенное покрытие

По сравнению с первым экспериментом тестовое покрытие увеличилось на 11.8% или на 1275 строк кода.

5.6 Фаззинг-тестирование

Наиболее популярный инструмент фаззинг-тестирования ядра Linux и его модулей - syzkaller. Однако он плохо подходит для задачи тестирования файловой системы, так как суть его работы заключается в генерации случайных программ состоящих из различных системных вызовов. Для достижения типичного сценария работы ФС syzkaller должен сгенерировать программу, состоящую как минимум из монтирования валидного образа файловой системы и дальнейших взаимодействий с ним при помощи, например, системных вызовов open, write и close. Для случайной генерации подходящего сценария потребуется множество перебранных неудачных вариантов последовательностей системных вызовов или монтирования невалидных образов.

Для того, чтобы сузить тестируемые syzkaller области, был применен файл конфигурации, приведенный в приложении (А). Параметры конфигурации syzkaller указывают ему список разрешенных к использованию системных вызовов и запрещают другие, чтобы он быстрее приходил к нужному сценарию. Инструменты описания грамматики системных вызовов, разработанные в syzkaller позволяют определить свою "мутацию"системного вызова заключающуюся, например, в определенных передаваемых параметрах. Таким образом в файле sys/linux/filesystem.txt определен системный вызов монтирования образа файловой системы JFFS2: "syz_mount _image$jffs2". При включении этого вызова в конфигурации и отключении других вариантов монтирования, монтирование всего будет производиться с параметром типа файловой системы jffs2.

При столкновении с ошибкой в виде блокировки, неопределенного поведения, либо просто сообщения об ошибке или предупреждения syzkaller генерирует программу репродьюсер на языке C, при запуске которой будет воспроизведен сценарий достигающий этой ошибки. Для генерации репродьюсеров при достижении определенной строки нужно добавить в коде ядра генерацию предупреждения об ошибке при достижении этой строки, что можно реализовать, например, при помощи макроса "WARN_ON(1)".

При помощи syzkaller удалось достичь покрытия 11 ранее непокрытых строк, отвечающих редким сценариям при обработке суперблока в процессе

монтировании ФС.

Таким образом для расширения покрытия кода ФС, syzkaller был запущен в конфигурации, разрешающей ему производить только системные вызовы для работы с ФС. Достигаемое им тестовое покрытие было проанализировано и в недостижимые ранее строки были помещены предупреждения об ошибках. При повторном запуске syzkaller при достижении этих строк он генерировал программы репродьюсеры, которые необходимо добавить в разрабатываемую систему автоматизированного тестирования.

На рис. 8 продемонстрировано покрытие до запуска программ репродьюсеров, на рис. 9 — покрытие после их запуска.

```
        :
      192 : static int jffs2_parse_param(struct fs_context *fc, struct fs_parameter *param)
        : {
      192 :         struct fs_parse_result result;
      192 :         struct jffs2_sb_info *c = fc->s_fs_info;
      192 :         int opt;
        :
      192 :         opt = fs_parse(fc, jffs2_fs_parameters, param, &result);
      192 :         if (opt < 0)
        :                 return opt;
        :
        0 :         switch (opt) {
        0 :         case Opt_override_compr:
        0 :                 c->mount_opts.compr = result.uint_32;
        0 :                 c->mount_opts.override_compr = true;
        0 :                 break;
        0 :         case Opt_rp_size:
        0 :                 if (result.uint_32 > UINT_MAX / 1024)
        0 :                         return invalf(fc, "jffs2: rp_size unrepresentable");
        0 :                 c->mount_opts.rp_size = result.uint_32 * 1024;
        0 :                 c->mount_opts.set_rp_size = true;
        0 :                 break;
```

Рис. 8: Покрытие без фаззинга

```
        :
      292 : static int jffs2_parse_param(struct fs_context *fc, struct fs_parameter *param)
        : {
      292 :         struct fs_parse_result result;
      292 :         struct jffs2_sb_info *c = fc->s_fs_info;
      292 :         int opt;
        :
      292 :         opt = fs_parse(fc, jffs2_fs_parameters, param, &result);
      292 :         if (opt < 0)
        :                 return opt;
        :
       90 :         switch (opt) {
       80 :         case Opt_override_compr:
       80 :                 c->mount_opts.compr = result.uint_32;
       80 :                 c->mount_opts.override_compr = true;
       80 :                 break;
       10 :         case Opt_rp_size:
       10 :                 if (result.uint_32 > UINT_MAX / 1024)
        3 :                         return invalf(fc, "jffs2: rp_size unrepresentable");
       10 :                 c->mount_opts.rp_size = result.uint_32 * 1024;
       10 :                 c->mount_opts.set_rp_size = true;
       10 :                 break;
```

Рис. 9: Покрытие с применением фаззинга

При тестировании JFFS2 при помощи доработанного инструмента фаззинга файловых систем fsfuzz было также достигнуто покрытие 13 ранее непокрытых строк, составляющих две ранее непокрытые функции обновления опций монтирования при реконфигурации суперблока ФС.

Таким образом, фаззинг-тестирование помогло покрыть 24 ранее непокрытых строки кода, достижимые только при редких случаях работы ФС.

5.7 Тестирование с использованием симуляции сбоев

Исходном код модулей ядра Linux и ФС в частности содержат множество строк кода, отвечающих за обработку крайне редко происходящих сценариев, которые однако в случае возникновения должны быть корректно обработаны. Наглядным примером подобных обработчиков ошибок, редко встречаемых в типичном сценарии работы модуля, является проверка значения, возвращаемого функциями выделения памяти. Для тестирования корректной работы ядра в подобном сценарии используется технология симуляции сбоев [6].

В данной работе применялся метод целенаправленной разработки тестов с симуляцией сбоев, который малоэффективен при создании универсальной системы тестирования нескольких модулей или драйверов ядра, однако подходит для создания тестов направленных на работу с определенной файловой системой.

При анализе тестового покрытия был обнаружен достаточно большой участок кода, обрабатывающий сценарий сбоя в функции записи данных на флэш-память jffs2_flash_writev. Соседние этому обработчику ошибки строки покрываются тестом номер 3 из группы generic из набора xfstests.

Для симуляции сбоя в этой функции был использован инструмент Linux Fault Injection, функционал которого можно подключить, определив в файле конфигурации ядра параметр "CONFIG_FAULT_INJECTION". В параметрах внедрения ошибок было указано имя функции, в которой по тестовому сценарию должен произойти сбой, единичный интервал между сбоями, вероятность сбоя 100%, количество сбоев равное количеству запуска тестируемой функции в рассматриваемом тесте набора и возвращаемое значение функции -12, соответствующее коду ошибки в случае отсутствия свободной памяти (-ENOMEM). Покрытие участка кода обработчика ошибок до тестирования с использованием симуляции сбоя приведено на рис. 10.

Рис. 10: Покрытие без симуляции сбоев

При запуске выбранного теста с использованием симуляции сбоя функции jffs2_flash_writev удалось покрыть 24 ранее непокрытые строки обработки ошибки в запускающей ее функции jffs2_write_dnode, а также ранее непокрытую функцию проверки корректности суперблока ФС в случае подобной ошибки, состоящую из 5 строк кода. Кроме того дополнительно были покрыты 7 строк обрабатывающие ошибку произошедшую в функции jffs2_write_dnode. Таким образом тест с использованием симуляции сбоя во внутренней функции JFFS2 покрывает 36 ранее непокрытых строк. Покрытие того же участка кода, после включения в систему нового теста приведено на рис. 11.

```
654089 :        if (ret || (retlen != sizeof(*ri) + datalen)) {
    10 :                pr_notice("Write of %zd bytes at 0x%08x failed. returned %d, retlen %zd\n",
       :                          sizeof(*ri) + datalen, flash_ofs, ret, retlen);
       :
       :                /* Mark the space as dirtied */
    10 :                if (retlen) {
       :                        /* Don't change raw->size to match retlen. We may have
       :                           written the node header already, and only the data will
       :                           seem corrupted, in which case the scan would skip over
       :                           any node we write before the original intended end of
       :                           this node */
     6 :                        jffs2_add_physical_node_ref(c, flash_ofs | REF_OBSOLETE, PAD(sizeof(*ri)+datalen), NULL);
       :                } else {
     4 :                        pr_notice("Not marking the space at 0x%08x as dirty because the flash driver returned retlen zero\n",
       :                                  flash_ofs);
       :                }
    10 :                if (!retried && alloc_mode != ALLOC_NORETRY) {
       :                        /* Try to reallocate space and retry */
     4 :                        uint32_t dummy;
     4 :                        struct jffs2_eraseblock *jeb = &c->blocks[flash_ofs / c->sector_size];
       :
     4 :                        retried = 1;
       :
     4 :                        jffs2_dbg(1, "Retrying failed write.\n");
       :
     4 :                        jffs2_dbg_acct_sanity_check(c,jeb);
     4 :                        jffs2_dbg_acct_paranoia_check(c, jeb);
       :
     4 :                        if (alloc_mode == ALLOC_GC) {
     0 :                                ret = jffs2_reserve_space_gc(c, sizeof(*ri) + datalen, &dummy,
       :                                                             JFFS2_SUMMARY_INODE_SIZE);
       :                        } else {
       :                                /* Locking pain */
     4 :                                mutex_unlock(&f->sem);
     4 :                                jffs2_complete_reservation(c);
       :
     4 :                                ret = jffs2_reserve_space(c, sizeof(*ri) + datalen, &dummy,
       :                                                          alloc_mode, JFFS2_SUMMARY_INODE_SIZE);
     4 :                                mutex_lock(&f->sem);
       :                        }
       :
     4 :                        if (!ret) {
     4 :                                flash_ofs = write_ofs(c);
     4 :                                jffs2_dbg(1, "Allocated space at 0x%08x to retry failed write.\n",
       :                                          flash_ofs);
       :
     4 :                                jffs2_dbg_acct_sanity_check(c,jeb);
     4 :                                jffs2_dbg_acct_paranoia_check(c, jeb);
       :
     4 :                                goto retry;
       :                        }
     0 :                        jffs2_dbg(1, "Failed to allocate space to retry failed write: %d!\n",
       :                                  ret);
       :                }
       :                /* Release the full dnode which is now useless, and return */
     6 :                jffs2_free_full_dnode(fn);
    12 :                return ERR_PTR(ret?ret:-EIO);
```

Рис. 11: Покрытие с применением симуляции сбоев

Аналогичным образом был создан второй тест, на основе того же теста из набора xfstests, покрывающий при помощи симуляции ошибки в функции создания нового индексного дескриптора по причине нехватки памяти. Данный тест покрывает 3 новые строки кода функции jffs2_create.

5.8 Анализ непокрытого кода

Суммарное покрытие, достигаемое тестами из набора xfstests и пакета mtd-utils, тестами, созданными из программ репродьюсеров сгенерированных инструментом syzkaller и фаззинг-тестирования при помощи fsfuzz составляет 6216 строк кода из 7829, что соответствует покрытию 79.4% строк исходного кода. Функциональное покрытие составляет 91.4% или 286 из 313 функций. Визуализация итоговой информации о покрытии, собранной утилитой lcov представлена на рис. 12.

	Hit	Total	Coverage
Lines:	6216	7829	79.4 %
Functions:	286	313	91.4 %

Рис. 12: Покрытие достигаемое разработанной системой

При достижении подобного результата дальнейшее увеличение тестового покрытия кода является крайне трудоёмкой задачей и покрытие 79.4% строк кода можно считать достойным результатом. Однако каждая функция добавляется в код ФС для выполнения особой задачи, и факт наличия непокрытых функций является проблемой, так как тестовые скрипты не соответствуют всем возможным сценариям работы файловой системы.

Для дальнейшего развития системы тестирования исследуемой файловой системы и подведения итогов проделанной работы необходимо провести анализ непокрытого кода.

Оставшиеся непокрытыми 27 функций были классифицированы на 4 группы. Результат представлен на рис. 13.

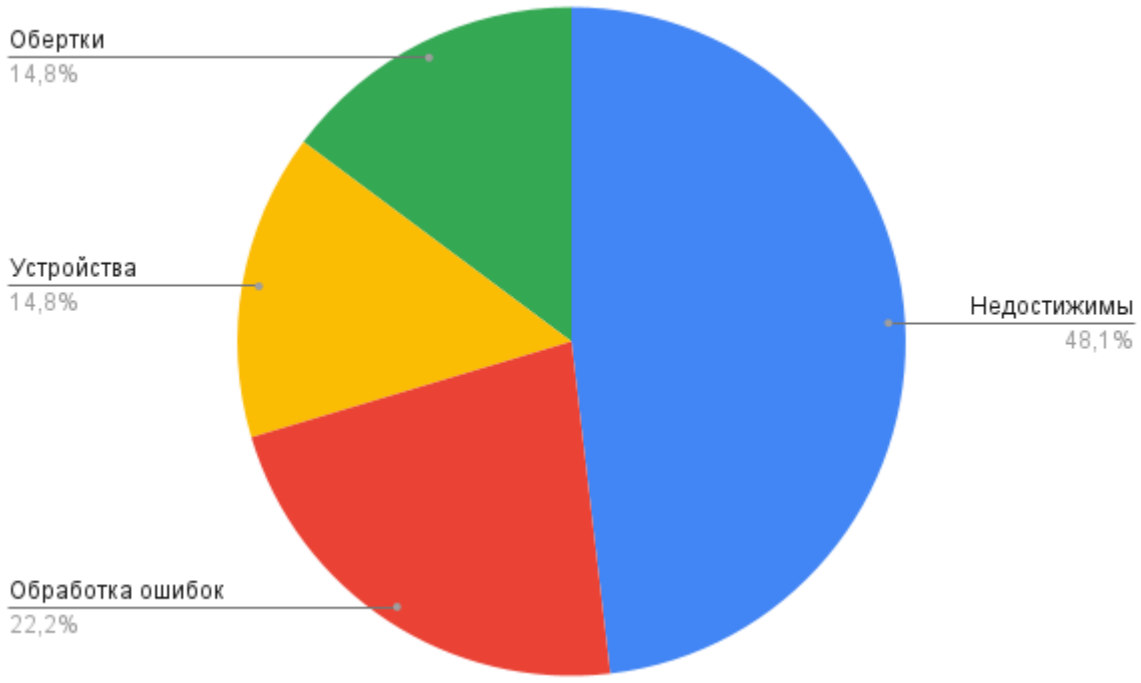

Рис. 13: Категории непокрытых функций

Во-первых, 13 из этих функций можно считать недостижимыми при подходе тестирования скриптами пространства пользователя. Наглядным примером являются функции деинициализации алгоритмов сжатия. Деинициализация структур, хранящих данные об алгоритмах сжатия, про-

исходит в случае возникновении ошибки при инициализации модуля ФС, что происходит на этапе загрузки ядра ОС. Внедрить ошибки для тестирования подобного сценария невозможно используя скрипты пространства пользователя.

Во-вторых, 6 функций отвечают за обработку ошибок в редких сценариях и могут быть покрыты при помощи специальных тестов с использованием симуляции сбоев, аналогично тестам, описанным в прошлом разделе.

В-третьих, 4 непокрытых функций нужны для работы ФС с особыми видами устройств флэш-памяти. Например, функция jffs2_dataflash_setup и jffs2_dataflash_cleanup выполняют соответственно инициализацию и деинициализацию буфера записи, для работы с устройством dataflash. Еще 2 функции имеют такое же назначение, но для работы с NOR-памятью с контроллером.

В-четвертых, 4 функции отвечают за операции с суперблоком файловой системы, как, например, поиск родительского каталога. Эти функции могут быть покрыты тестами пользовательского пространства, однако они фактически являются обертками над функциями, которые являются общими для всех ФС и определены вне директории модуля JFFS2. Как, например, функция jffs2_fh_to_parent просто ссылается на функцию generic_fh_to_parent, определенную в файле fs/libfs.c.

6 Заключение

В результате проделанной работы была разработана система автоматизированного тестирования файловой системы JFFS2 на основе существующих решений для тестирования файловых систем. В процессе разработки системы тестирования был доработан инструмент создания образа JFFS2, путем добавления в него поддержки алгоритма сжатия данных rubin, с использованием драйвера mtdblock был адаптирован под особенности JFFS2 инструмент фаззинг-тестирования файловых систем fsfuzz.

Набор xfstests, являющийся наиболее популярным и поддерживаемым сообществом пользователей и разработчиков Linux решением для тестирования файловых систем, был адаптирован с целью тестирования JFFS2. К существующим тестовым скриптам набора были добавлены тесты пакета mtd-utils, тесты созданные из программ сгенерированных инструментом syzkaller и тесты, использующие симуляцию сбоев внутренних функций JFFS2.

Разработанная система тестирования была экспериментально проверена. В результате проведённого тестирования всеми полученными скриптами было достигнуто покрытие в 79,4% строк исходного кода и 91,4% функций. Это является достойным результатом и дальнейшее расширение тестового покрытия является крайне сложной задачей.

С целью дальнейшего развития разработанной системы был произведен анализ достигаемого тестового покрытия исходного кода, в результате которого были классифицированы непокрытые тестами функции ФС. По итогу проведенного анализа выявлено, что 48% непокрытых функций являются недостижимыми при тестировании скриптами пользовательского пространства. Для тестирования еще 15% непокрытых функций необходимо использование реальных устройств флэш-памяти определенного типа, ввиду отсутствия эмуляторов. В тестирование еще 15% непокрытых функций нет острой необходимости, так как они фактически являются обертками над функциями общей для всех ФС библиотеки, которые могут быть протестированы другими тестовыми наборами. И оставшиеся 22% непокрытых функций, что эквивалентно 2% от общего числа функций файловой системы могут быть покрыты при дальнейшем развитии тестовой системы.

Дальнейшее увеличение тестового покрытия является трудоемкой задачей, но важно продолжать работу над улучшением качества тестирова-

ния, чтобы обеспечить более полное покрытие кода и повысить надежность файловой системы.

Список литературы

[1] *Э., Таненбаум.* // Современные операционные системы 4-е издание. — СПб Питер. — 2022.

[2] *К., Симмондс.* // Встраиваемые системы на основе Linux. — ДМК Пресс. — 2017.

[3] Документация к MTD (generic Linux subsystem for memory devices). [Электронный ресурс]. — URL: `http://www.linux-mtd.infradead.org/index.html` (Дата обращения 20.06.2024).

[4] *D., Woodhouse.* JFFS : The Journalling Flash File System. / Woodhouse D. — 2001.

[5] *Цыварев А.В., Мартиросян В.А.* Тестирование драйверов файловых систем в ОС Linux. — Труды Института системного программирования РАН том 23. — 2012.

[6] *Цыварев А.В., Хорошилов А.В.* Использование симуляции сбоев при тестировании компонентов ядра ОС Linux. — Труды Института системного программирования РАН том 27. — 2015.

[7] *Aota N., Kono K.* File Systems are Hard to Test – Learning from Xfstests. / Kono K. Aota N. // *IEICE TRANSACTIONS on Information and Systems.* — 2019.

[8] Миссия Центра верификации ОС Linux Института Системного Программирования РАН. [Электронный ресурс]. — URL: `http://www.linuxtesting.ru/` (Дата обращения 20.06.2024).

[9] Xfstests [Электронный ресурс]. — URL: `https://github.com/kdave/xfstests` (Дата обращения 28.06.2024).

[10] Fsfuzz [Электронный ресурс]. — URL: `https://github.com/sughodke/fsfuzzer` (Дата обращения 15.06.2024).

[11] Syzkaller [Электронный ресурс]. — URL: `https://github.com/google/syzkaller` (Дата обращения 15.05.2024).

[12] Пакет инструментов mtd-utils [Электронный ресурс]. — URL: `http://git.infradead.org/mtd-utils.git` (Дата обращения 20.06.2024).

[13] Документация к UBI [Электронный ресурс]. — URL: http://www.linux-mtd.infradead.org/faq/ubi.html (Дата обращения 28.02.2024).

[14] Linux Verification Center - Linux kernel stable [Электронный ресурс]. — URL: https://git.linuxtesting.ru/pub/scm/linux/kernel/git/lvc/linux-stable.git (Дата обращения 20.06.2024).

[15] Документация - Using gcov with Linux kernel [Электронный ресурс]. — URL: https://www.kernel.org/doc/html/v5.10/dev-tools/gcov.html (Дата обращения 28.02.2024).

[16] lcov [Электронный ресурс]. — URL: https://github.com/linux-test-project/lcov (Дата обращения 28.02.2024).

[17] Документация Linux Fault Injection [Электронный ресурс]. — URL: https://docs.kernel.org/fault-injection/fault-injection.html (Дата обращения 20.06.2024).

Приложение А

Полная конфигурация ядра

```
CONFIG_MTD=y
CONFIG_MTD_BLKDEVS=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_MTDRAM=y
CONFIG_MTDRAM_TOTAL_SIZE=8192
CONFIG_MTDRAM_ERASE_SIZE=128
CONFIG_MTD_BLOCK2MTD=y
CONFIG_MTD_NAND_CORE=y
CONFIG_MTD_NAND_NANDSIM=y
CONFIG_MISC_FILESYSTEMS=y
CONFIG_JFFS2_FS=y
CONFIG_JFFS2_FS_WRITEBUFFER=y
CONFIG_JFFS2_FS_WBUF_VERIFY=y
CONFIG_JFFS2_SUMMARY=y
CONFIG_JFFS2_FS_XATTR=y
CONFIG_JFFS2_FS_POSIX_ACL=y
CONFIG_JFFS2_FS_SECURITY=y
CONFIG_JFFS2_COMPRESSION_OPTIONS=y
CONFIG_JFFS2_ZLIB=y
CONFIG_JFFS2_LZO=y
CONFIG_JFFS2_RTIME=y
CONFIG_JFFS2_RUBIN=y
# CONFIG_JFFS2_CMODE_NONE is not set
CONFIG_JFFS2_CMODE_PRIORITY=y
# CONFIG_JFFS2_CMODE_SIZE is not set
# CONFIG_JFFS2_CMODE_FAVOURLZO is not set
```

Приложение Б

Конфигурация syzkaller


```
{
"target" : "linux/amd64" ,
"http" : "127.0.2.1:56741" ,
"workdir" : "syzkaller/workdir" ,
"kernel_obj" : "linux-stable" ,
"image" : "syzkaller/tools/bullseye.img" ,
"sshkey" : "syzkaller/tools/bullseye.id_rsa" ,
"syzkaller" : "syzkaller" ,
"procs" : 2,
"type" : "qemu" ,
"vm" : {
"count" : 1,
"cpu" : 2,
"mem" : 8096,
"kernel" : "linux-stable/arch/x86/boot/bzImage" ,
"cmdline" : "net.ifnames=0"
},
"interests" : ["fs/jffs2/" ],
"enable_syscalls" : ["syz_mount_image$jffs2" , "getuid" , "geteuid" ,
"fstat" , "getresuid" , "open" , "write" , "read" ]
}
```