# Использование симуляции сбоев при тестировании компонентов ядра ОС Linux

*[1]А.В. Цыварев <tsyvarev@ispras.ru>*
*[1,2,3,4]А.В. Хорошилов <khoroshilov@ispras.ru>*
*[1] Институт системного программирования РАН,*
*109004, Россия, г. Москва, ул. А. Солженицына, дом 25*
*[2] Московский государственный университет имени М.В. Ломоносова,*
*119991, Россия, Москва, Ленинские горы, д. 1.*
*[3]Московский физико-технический институт (государственный университет),*
*141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9*
*[4] Национальный исследовательский университет «Высшая школа экономики»*
*101000, Россия, Москва, ул. Мясницкая, д.20*

**Аннотация**. В статье рассматриваются методы тестирования компонентов ядра ОС Linux с использованием симуляции сбоев. Основная цель таких методов – проверка поведения модуля при возникновении нештатных ситуаций, таких как сбои в аппаратуре или нехватка ресурсов.

Эти ситуации на практике встречаются достаточно редко и непредсказуемо, что существенно затрудняет их обнаружение и локализацию.

Единственным распространённым подходом к поиску таких проблем является случайное внесение сбоев в ходе выполнения обычных тестов.

Но из-за недетерминированной природы такой способ тестирования не обеспечивает достаточной воспроизводимости тестовых сценариев, что не позволяет давать никаких гарантий корректности поведения драйвера даже в случае успешного выполнения всех тестов.

В статье предлагаются новые методы систематического тестирования устойчивости к сбоям, решающие проблему воспроизводимости тестирования.

В основе этих методов лежит систематический перебор точек для симуляции сбоев с отбрасыванием потенциально "неинтересных" тестов, которые не проверяют ничего нового по сравнению с другими тестами. Применимость и эффективность конкретного систематического метода зависит от способа определения "неинтересных" тестов: это может быть как ручное выделение уникальных участков базовых тестов, ограничивающих множество точек для симуляции сбоев, так и автоматическое определение дублирующих точек по стеку вызовов функций тестируемого модуля. Была проведена апробация предложенных систематических методов на одном из драйверов Linux и двух тестовых наборах. Результаты апробации показывают, что систематические методы не уступают по эффективности методу со случайным внесением сбоев, а в чем-то и превосходят его.

**Ключевые слова:** Linux; компоненты ядра Linux; драйвер; тестирование; симуляция сбоев.

**DOI:** 10.15514/ISPRAS-2015-27(5)-9

**Для цитирования:** А.В. Цыварев, А.В. Хорошилов. Использование симуляции сбоев при тестировании компонентов ядра ОС Linux. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 157-174. DOI: 10.15514/ISPRAS-2015-27(5)-9.

## *1. Введение*

Операционные системы, основанные на ядре Linux, широко используются в мире. Они обеспечивают работу более 90% мощнейших суперкомпьютеров из международного рейтинга TOP500 и преобладают на рынке операционных систем для мобильных устройств в составе ОС Android. Существенная часть компонентов ядра Linux является драйверами устройств - одними из ключевых компонентов Linux, отвечающих за организацию взаимодействия пользователя с различными внешними устройствами компьютеров: жесткими дисками, съемными накопителями, сетевыми адаптерами, и т. д. Кроме того, в Linux компонентами ядра реализованы и некоторые программные сущности, такие как сетевые протоколы и файловые системы. В некотором роде такие компоненты - "драйвера программных сущностей". Далее в статье будет использоваться название *драйвер* как общее обозначение для драйверов устройств и "драйверов программных сущностей".

В статье [1] на примере драйверов файловых систем были описаны основные требования к системе тестирования, позволяющей качественно проверять работоспособность драйверов. В качестве одного из требований приводилось тестирование работы драйвера при нехватке памяти и в случае сбоев устройства. Как эффективный способ такого тестирования предлагалось использовать симуляцию таких условий в функциях, вызываемых из тестируемого драйвера. Для драйвера это выглядит как будто вызванная им функция не смогла выделить память, не смогла записать данные на диск или считать данные с диска. Для обобщения такую симуляцию будем называть симуляцией сбоев.

В данной статье исследуются существующие методы тестирования с использованием такого рода симуляции, а также предлагаются новые методы.

Во втором разделе статьи будет рассмотрен наиболее прямолинейный способ тестирования с использованием симуляции сбоев – целенаправленная разработка каждого теста вместе с сценарием симуляции сбоев. Из-за трудоемкости разработки таких тестов, они практически не применяются. В третьем разделе будет описан уже существующий метод с использованием рандомизированного сценария симуляции сбоев. Хотя качественные показатели метода неплохи, недетерминированность сценария симуляции сбоев не позволяет использовать его результаты тестирования в качестве обоснования корректности драйвера. Поэтому в четвертом разделе

предлагаются методы систематической симуляции сбоев, которые могли бы преодолеть недостатки рандомизированного метода. В пятом разделе будут приведены результаты испытания описанных методов с существующими тестовыми наборами на одном из драйверов ОС Linux. В шестом разделе будет проведено сравнение описанных методов тестирования с симуляцией сбоев, и будут сделаны выводы об их применимости на практике.

## *2. Целенаправленная разработка тестов с симуляцией сбоев*

Самый прямолинейный способ тестирования с симуляцией сбоев – разработка каждого теста вместе со сценарием симуляции сбоев. Под сценарием симуляции сбоев подразумевается выбор конкретных вызовов функций из драйвера для симуляции сбоев в этих вызовах. В дальнейших рассуждениях на такие вызовы функций будем ссылаться как на точки симуляции. Такая разработка тестов может основываться на поведении аналогичных тестов, но без симуляции сбоев. При выборе сценария симуляции можно учитывать исходный код драйвера и его логику, результаты измерения покрытия кода при запуске теста без сбоев и пр.

Теоретически, таким способом можно заставить драйвер работать во всех случаях поведения функций, сбои которых эмулируются. И таким образом исследовать корректность обработки всех возможных ошибок.

На практике же такой метод симуляции сбоев малоэффективен. Во-первых, разработка каждого теста вместе со сценарием симуляции сбоев очень трудоемка: необходимо исследовать исходный код драйвера, исследовать исходный код самого ядра или на основе опытных данных определить его точную логику. Во-вторых, получившиеся тесты будут применимы и обеспечивать необходимое покрытие только для конкретного драйвера и его конкретной версии и для конкретной версии ядра. Для других драйверов, других версий данного драйвера или других версий ядра такие тесты с большой вероятностью будут либо некорректны, либо бессмысленнены, так как их запуск не будет вынуждать драйвер пройти по задуманному пути. Такое ограничение тестов возникает из-за того, что их разработка должна учитывать реализацию драйвера и ядра, с которым драйвер очень тесно взаимодействует.

По причине трудоемкости разработки, целенаправленная разработка тестов со сценариями симуляции практически неприменима для поиска новых ошибок в драйверах. Таким способом можно только проверить наличие уже найденных ошибок, которые плохо выявляются другими методами, и их исправление (регрессионное тестирование). Далее в статье этот метод не рассматривается.

## 3. Рандомизированный метод симуляции сбоев

Наиболее распространенным методом симуляции сбоев при тестировании ядра Linux и его компонентов является использование рандомизированного сценария симуляции сбоев. Такой метод используется, например, в проекте Linux Test Project (LTP)[2]. При рандомизированном сценарии, в процессе выполнения теста каждый вызов функции выделения памяти, работы с диском и др. возвращает ошибку с некоторой вероятностью p ($0<p<1$). В таком окружении каждый тест выполняется несколько раз, что позволяет наблюдать реакцию драйвера на разные ошибки.

Применение такого метода симуляции сбоев не требует каких-либо специальных свойств от базовых тестов, которые запускаются в сбойном окружении. Поэтому в качестве базового набора тестов часто берут уже существующий набор тестов, который работает с данным драйвером.

Теоретически, выполняя конкретный (базовый) тест большое количество раз, с таким методом симуляции сбоев можно наблюдать поведение драйвера при всех последовательностях сбоев в тестовом сценарии. На практике же, количество повторений теста ограничено временем, отведенным на тестирование. Поэтому за одну итерацию тестов будет наблюдаться только часть из возможных последовательностей сбоев.

Вероятностная база сценария симуляции сбоя определяет недостаток такого подхода: итерация тестов не дает гарантии, что то или иное *конкретное* поведение драйвера будет проверено.

С другой стороны, при рандомизированном сценарии симуляции сбоев есть вероятность наблюдать одинаковые последовательности сбоев при разных запусках одного теста. Это ведет к увеличению времени тестирования без повышения качества, то есть уменьшается эффективность тестирования.

Хотя использованием рандомизированного метода симуляции сбоев обеспечивает неплохое качество тестирования (при адекватном базовом наборе тестов), слабая воспроизводимость тестовых сценариев не позволяет давать гарантий корректности поведения драйвера. Основным источником проблем с воспроизводимостью является, понятно, факт недетерминированности сценариев сбоев. Поэтому логичным способом улучшения воспроизводимости результатов тестирования было бы использование детерминированных сценариев симуляции.

## 4. Систематическая симуляция сбоев

Вместо того, чтобы запускать каждый тест несколько раз с одним и тем же рандомизированным сценарием симуляции сбоев, можно систематически перебирать сценарии симуляции из заранее сформированного множества, и запускать тест с каждым из таких сценариев.

Таким способом определяется класс методов т.н. систематической симуляции сбоев. Формально, каждый метод этого класса из набора *нормальных* тестов $N$ (не использующих симуляцию сбоев) автоматически генерирует набор тестов $S$ с симуляцией сбоев, каждый тест которого есть пара $s=(n, c)$, где $n \subset N$, а $c$ выбирается из множества сценариев симуляции сбоев $C$. Как и в случае с рандомизированным сценарием, в качестве набора $N$ выбирается уже существующий набор тестов.

При выборе метода систематической симуляции сбоев должны учитываться следующие критерии:

1. Качество тестирования с использованием набора тестов $S$.
2. Эффективность тестирования с использованием набора тестов $S$.
3. Применимость набора тестов $S$ для большого множества драйверов и их версий.
4. Применимость метода для широкого класса базовых тестов.

Введем две гипотезы, описывающие тестируемый драйвер и исходный набор тестов.

Первая гипотеза предполагает, что драйвер работает по принципу «все или ничего»: если сбой не дает успешно выполнить запрос, то состояние драйвера возвращается к состоянию, которое драйвер имел до выполнения запроса. При этом считается, что в ходе возврата к исходному состоянию драйвера функции, допускающие сбой, не вызываются.

Это гипотеза выполняется для большей части драйверов и запросов к ним; при этом инициатору запроса, обнаружившему сбой, часто возвращается некоторый индикатор ошибки.

Вторая гипотеза уже касается исходных тестовых сценариев: трасса выполнения любого исходного теста не зависит от запуска этого теста. Своего рода, это предположение о детерминированности теста и драйвера.

Как и первая гипотеза, гипотеза о детерминированности выполняется или почти выполняется для большинства драйверов Linux. На тесты же эта гипотеза накладывает существенные ограничения. Которые, впрочем, выполняются для многих существующих тестов.

В условиях гипотезы «все или ничего» достаточно рассматривать сценарии с однократной симуляцией сбоев; тестирование с использованием более сложных сценариев может быть сведено к однократным сбоям. Например, корректность драйвера при запуске исходного теста с симуляцией сбоев в точках $A$ и $B$ можно проверить, выполнив запуск исходного теста с симуляцией с точке $A$, а затем повторив запуск исходного теста, но уже без запроса драйверу, включающего точку $A$, с симуляцией сбоев в точке $B$. (Здесь неявно считается, что эквивалент исходному тесту без запроса с точкой $A$ также присутствует в исходном тестовом наборе.)

Из гипотезы детерминированности следует, что набор точек для однократной симуляции сбоев в исходном тесте устойчив. Этот набор точек можно выделить, запустив исходный тест без симуляции сбоев, а затем ссылаться на

каждую точку с возможной симуляцией сбоев, например, по ее порядковому номеру в тесте.

Все эти рассуждения приводят к следующему методу систематической симуляции сбоев.

## 4.1 Метод перебора всех возможных точек симуляции сбоев

Алгоритм тестирования:

1. Запускаем каждый тест $n$ исходного набора $N$ без симуляции сбоев, но собирая данные о возможных точках симуляции сбоев в процессе выполнения теста. В простейшем случае достаточно узнать общее количество таких точек $K(n)$.
2. Запускаем каждый тест $n$ исходного набора $N$ $K(n)$раз. В запуске номер $k$ используем сценарий симуляции сбоя в точке номер $k$.

Исходя из рассуждений выше, такой метод тестирования с симуляцией сбоев дает *полный* тестовый набор: он проверяет драйвер во всех ситуациях, в каких его можно проверить, запуская тесты исходного набора с различными сценариями симуляции сбоев.

На первый взгляд, получившийся тестовый набор с симуляцией сбоев обладает также свойством минимальности: ни один тест из него нельзя выкинуть без ухудшения качества тестирования. Ведь запуск исходного теста с каждым новым сценарием однократной симуляции сбоев позволяет выполнить код драйвера, отвечающий за выполнение обработки сбоя в новом участке кода драйвера. На самом деле, это рассуждение неверно.

Во-первых, разные точки возможных сбоев в трассе выполнения исходного теста могут соответствовать одному и тому же участку кода драйвера. Это происходит, например, когда исходный тест повторяет запрос драйверу для достижения какого-то особенного состояния драйвера (например, исчерпания свободных блоков в файловой системе).

Во-вторых, разные тесты в исходном тестовом наборе могут выполнять одинаковую последовательность операций с драйвером в качестве подготовительной работы. Например, как для создания файла, так и для создания директории в файловой системе, ее требуется сначала примонтировать. В итоге тесты со сбоями в этих общих последовательностях операций дублируют друг друга.

На практике выходит, что подавляющая часть тестов в получившемся наборе тестов с симуляцией сбоев – «неинтересная», то есть в плане тестирования эти тесты не дают ничего нового по сравнению с остальными. Применимость метода быстро упирается в практическое ограничение по времени выполнения тестов при увеличении сложности исходных тестов и/или их количества.

Рассмотрим способы уменьшения количества «неинтересных» тестов, порожденных данным методом симуляции сбоев.

## 4.2 Выделение уникальных участков в коде тестов

Количество заведомо «неинтересных» тестов можно сократить, если в каждом исходном тесте выделить участки кода, выполняющие операции с драйвером, характерные только для данного теста, и никаких других. Такие участки кода теста будем называть "уникальными". Используя разметку уникальных участков кода тестов, метод простого перебора можно модифицировать следующим образом:

1. Запускаем каждый тест $n$ исходного набора $N$ без симуляции сбоев, но собирая данные о возможных точках симуляции сбоев в процессе выполнения уникальных участков кода теста. В простейшем случае достаточно узнать общее количество таких точек $U(n)$.
2. Запускаем каждый тест $n$ исходного набора $N$ $U(n)$ раз. В запуске номер $u$ используем сценарий симуляции сбоя в точке номер $u$.

Если разметка выполнена корректно, то набор тестов, порожденный модифицированным методом, сохранит свойство полноты, но за счет значительно сокращенного общего количества тестов метод уже будет применим в реальных условиях.

Выделение уникальных участков кода в тестах должно выполняться человеком. Эта деятельность усложняется необходимостью учитывать информацию обо всех тестах в наборе. Тем не менее, это существенно проще, чем разработка сценариев симуляции для каждого теста вручную (см. главу 2).

Альтернативой ручному выделению "уникальных" участков кода является использование базовых тестов, в которых уже выделены 3 стандартных этапа теста: *подготовка*, *основная часть* и *освобождение ресурсов*. В этом случае логично ограничить симуляцию сбоев *основной частью* теста, которая отличает каждый тестовый сценарий от других. Понятно, что основная часть теста не обязана быть уникальной с точки его воздействия на драйвер, но сокращение времени тестирования по сравнению с неограниченной симуляцией сбоев может оказаться существенным для применимости метода.

## 4.3 Учет стека вызовов для отбрасывания неинтересных сценариев симуляции сбоев

Вместо того, чтобы вручную выделять «уникальные» участки кода тестов, можно использовать различные характеристические функции, использующие информацию о точке возможного сбоя, для автоматического определения дублирующих тестов с симуляцией сбоев. При этом метод простого перебора точек симуляцией сбоев будет обобщенно модифицирован следующим образом(считаем, что множество исходных тестов $N$ упорядочено):

1. Запускаем каждый тест $n$ исходного набора $N$ без симуляции сбоев, но собирая данные о возможных точках симуляции сбоев в процессе

выполнения теста. Помимо общего количества $K(n)$ таких точек, для каждой точки вычисляем значение характеристической функции $h(n, k)$.

2. Множество $Q$ - множество собранных значений характеристической функции $h$. Изначально пустое.
3. Для каждого $n$ из $N$ пробегаем числа $k$ от 1 до $K(n)$. Если $h(n,k) \in Q$ (тест с таким значением характеристической функции уже выполнялся), то ничего не делаем. Иначе выполняем тест $n$ с симуляцией сбоя в точке номер $k$ и добавляем значение $h(n, k)$ в множество $Q$.

Одна из простых, и в тоже время эффективных характеристических функций - стек вызовов(call stack) в возможной точке симуляции:

$CS(n,k)$ = {стек вызовов в возможной точке симуляции номер $k$ в процессе выполнения теста $n$}.

Под стеком вызовов в точке выполнения программы подразумевается рекурсивно генерируемая цепочка(массив) из точек в коде в формате

$$pair\ \{< \text{имя функции} >; < \text{смещение в коде функции} >\}$$

где первый элемент цепочки соответствует заданной точке выполнения, а каждый последующий - точке вызова функции, содержащей предыдущую точку. Генерация цепочки обрывается на точке входа в ядро.

Такая характеристическая функция одинаково классифицирует (возвращает одно и тоже значение) точки возможных сбоев, которые соответствуют одной и той же инструкции исходного кода ядра, цепочки вызовов функций для которых также совпадают.

Модификация метода тестирования с симуляцией сбоев, использующая такую характеристическую функцию, гарантированно выявляет случаи симуляции сбоев в общих участках инициализации различных тестов, и случаи симуляции сбоев в повторяющихся запросах при инициализации одного теста. За счет этого набор тестов с симуляцией сбоев значительно сокращается, и становится пригодным для использования.

Однако, при сравнении стеков вызовов в точках симуляции сбоев не учитываются возможные различия в классификации этих сбоев в коде их *обработки*. Одна из часто встречающихся ситуаций, когда такое различие существенно, это выделение памяти в цикле:

```
for(i = 0; i < 10; i++) {
   p[i] = kmalloc(…);
   if(!p[i]) {
      for(i--;i >= 0; i--) {
         kfree(p[i]);
   }
}
```

*Рис. 1. Выделение памяти с цикле.*

Хотя стеки вызовов при выделении памяти в первой(i = 0) и второй (i = 1) итерациях не отличаются, трасса обработки сбоев в этих выделениях памяти отличается: вложенный цикл выполняется только во втором случае.

Другая ситуация - зависимость кода обработки сбоев от переменных состояния:

```
p = kmalloc(…);
if(!p) {
   if(panic_on_error) panic(…);
   …
}
```

*Рис 2. Зависимость обработки сбоя от переменной состояния.*

В этом примере стек вызовов в выделении памяти не зависит от значения переменной *panic_on_error*. Но код обработки сбоя в этом выделения памяти зависит от этой переменной.

Как видно их этих примеров, использование $CS(n,k)$ в качестве характеристической функции для метода простого перебора точек симуляцией сбоев приведет к потере качества тестирования. А именно, теряются тесты, проверяющие код обработки сбоя в разных условиях.

Для того, чтобы смягчить эти потери, можно использовать следующие вариации характеристической функции $CS(n,k)$:

1) $CS\_1(n,k)$ = pair{$CS(n,k)$; 1, если $\exists m<k$: $CS(n,m) = CS(n,k)$, 0 иначе}. Такая функция отличает точку возможной симуляции сбоев, в которой заданный стек вызовов наблюдался впервые в тесте, от точки, стек вызовов которой наблюдался в какой-то предыдущей точке теста.
2) $CS\_set(n,k)$ = pair{$CS(n,k)$; $\cup CS(n,m)$, $m<k$}.

Характеристическая функция *CS_1* отличает (возвращает различные значения) точку возможной симуляции сбоев, в которой заданный стек вызовов наблюдался впервые в тесте, от точки, стек вызовов которой наблюдался в какой-то предыдущей точке теста. Таким образом, характеристика сценария симуляции сбоев в первой итерации цикла отлична от характеристики следующей итерации (пример на рис. 1).

Характеристическая функция *CS_set* также позволяет отличать сбои в первой и последующих итерациях циклов. Но помимо этого, эта функция позволяет отличать сбои с разными переменными состояниями (пример на рис.2), если эти переменные влияли на классификацию предыдущих запросов драйверу, в которых возможны сбои.

## *5. Апробация методов и ее результаты*

Описанные методы тестирования с симуляцией сбоев (кроме целенаправленной разработки таких тестов, описанной в разделе 2) были применены для тестирования драйвера файловой системы ext4.

Для реализации сценариев симуляции сбоев была выбрана платформа KEDR[3] и основанный на ней инструмент KEDR Fault Simulation. Как описано в статье[4], этот инструмент предназначен для симуляции сбоев в функциях, вызываемых драйвером, и он уже содержит все необходимое для сценария однократной симуляции в каждой точке, в том числе с ограничением по уникальному коду тестов. Сценарий для рандомизированной симуляции сбоев был также реализован с использованием уже существующей функциональности этого инструмента. Реализация сценариев, учитывающих стек вызовов, потребовала написания новых модулей ядра (т.н. индикаторы для KEDR Fault Simulation).

В качестве меры качества того или иного метода тестирования с симуляцией сбоев использовался прирост покрытия по коду драйвера ext4. А именно, количество строчек кода драйвера, покрытых в результате тестирования с использованием симуляции сбоев, но не покрытых при простом запуске тестов (без симуляции).

В приведенных ниже таблицах и их анализе используются следующие обозначения методов симуляции сбоев:

- *fsim_rnd* – рандомизированный сценарий (вероятность сбоя и количество повторений указываются дополнительно)
- *fsim_all* – однократная симуляция сбоев в каждой возможной точке
- *fsim_restricted* - однократная симуляция сбоев в каждой точке из заранее выбранного участка теста
- *fsim_stack_1* – однократная симуляция сбоев в каждой точке с отбрасыванием дублирующих сценариев, используя характеристическую функцию *CS_1*.

- *fsim_stack_set* – однократная симуляция сбоев в каждой точке с отбрасыванием дублирующих сценариев, используя характеристическую функцию *CS_set*.

В качестве исходных тестовых наборов были взяты:

1. Linux File System Verification Tests[5].
2. 10 тестов из набора Xfstests[6]. Тесты были отобраны с учетом применимости предположения о детерминированности тестов.

## 5.1 Linux File System Verification Tests (LFSVT)

Этот тестовый набор включает модульные тесты(unit tests), использующие достаточно простые сценарии. Эти особенности позволили в каждом таком тесте выделить участки кода, присущие именно этому тесту. Используя такие участки кода как «уникальные» (см. раздел 3.2), можно применять различные методы симуляции сбоев с ограничением области симуляции этими участками.

С другой стороны, количество тестов LFSVT и время их запуска достаточно велико, что делает неактуальным применение рандомизированных сценариев симуляции сбоев. Например, при вероятности сбоев 1% и повторении каждого базового теста 100 раз общее количество тестов с симуляцией сбоев будет больше 1 млн., а оценка времени их выполнения – 7 дней.

*Табл. 1. Применение методов симуляции сбоев для тестов LFSVT*

| Метод симуляции сбоев | Ограничение симуляции уникальными участками кода теста | Прирост покрытия, строк кода | Время тестирования, мин | Стоимость прироста покрытия, мин/строк |
|---|---|---|---|---|
| (без симуляции) | - | - | 110 | - |
| *fsim_restricted* | да | 311 | 92 | 0,30 |
| *fsim_stack_1* | да | 266 | 2 | 0,0075 |
| *fsim_stack_set* | да | 266 | 3 | 0,011 |
| *fsim_all* (теор. оценка) | нет | - | 5000 | - |
| *fsim_stack_1* | нет | 333 | 4 | 0,012 |
| *fsim_stack_set* | нет | 354 | 9 | 0,025 |

Как видно из таблицы, при учете разметки уникальных участков кода тестов автоматическое отбрасывание «похожих» сценариев симуляции сбоев действительно ухудшает качество тестирования: прирост покрытия от методов *fsim_stack_1* и *fsim_stack_set* меньше, чем у *fsim_restricted*.

С другой стороны, такое отбрасывание делает возможным тестирование с симуляцией сбоев без учета уникальных участков кода тестов: запуск всех (~500 тыс.) тестов с однократной симуляцией потребовал бы несколько дней, в то время как тестирование с методами *fsim_stack_1* и *fsim_stack_set* выполняется меньше, чем за 10 минут.

Более того, использование методов *fsim_stack_1* и *fsim_stack_set* без ограничения области сбоев дает больший прирост покрытия (при значительно меньшей стоимости), чем использование метода *fsim_restricted* с учетом размеченных «уникальных» участков кода тестов. Это говорит о неточности разметки. Детальный анализ покрытия и кода тестов подтвердил недостатки в разметке.

## 5.2 Тестовый набор xfstests

Набор xfstests состоит из тестов системного уровня, которые используют более сложные сценарии, чем тесты Spruce. Не все тесты из этого набора обеспечивают детерминированное воздействие на драйвер ФС ext4, поэтому для экспериментов использовалась только часть из них (10 тестов), воздействие которых на драйвер ФС ext4 близко к детерминированному.

Так как разметки «уникальных» участков кода в этих тестах нет, то метод простого перебора точек симуляции сбоев для таких тестов потребовал бы недопустимо много времени. Однако систематические методы, использующие фильтрацию сценариев на основе стека вызовов, вполне применимы.

Сложные сценарии тестов и небольшое количество этих тестов позволяют использовать рандомизированные сценарии для симуляции сбоев в этих тестах. Количество повторений тестов с рандомизированными сценариями было выбрано 200, что обеспечило сопоставимость суммарного времени такого тестирования с детерминированными методами.

Несмотря на то, что тестовые сценарии в наборе Xfstests вполне детерминированные, алгоритм работы драйвера ext4 под воздействием этих тестов не является в точности детерминированным. Для более точного сравнения методов, тестовые наборы, основанные на этих методах, выполнялись несколько раз, а результаты усреднялись. Для этого использовалась следующая методика испытаний:

1. Базовый набор (без использования симуляции сбоев) запускался 10 раз. В базовое покрытие включались строки, покрытые хотя бы в одной из запусков.
2. Каждый набор тестов с симуляцией сбоев выполнялся 5 раз. В качестве усредненного прироста покрытия учитывались строки, покрытые как минимум в половине запусков этого набора (3 из 5), но не включенные в базовое покрытие. В качестве времени тестирования бралось среднее время по 5 запускам.

Так как качество тестирования с использованием рандомизированного метода зависит от вероятности сбоя, то вначале теоретико-экспериментальным образом был выделен промежуток вероятностей (в процентах), на котором достигается максимум покрытия по коду драйвера. Этим промежутком оказался [0.5; 2.0]. Затем с шагом 0.1% на этом промежутке каждая вероятность испытывалась в соответствии с п.2 приведенной выше методики. Следующий график отражает результат этих испытаний.

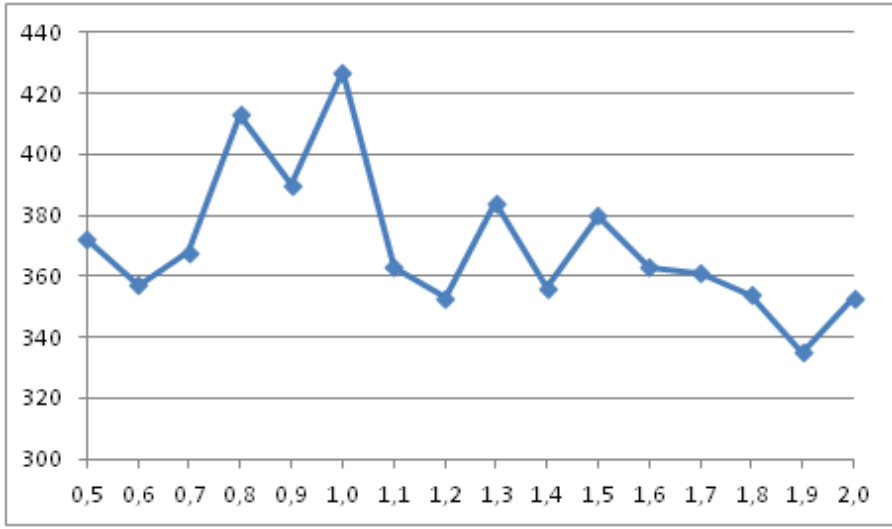


*Рис.3 Прирост покрытия по коду в зависимости от вероятности рандомизированного метода*

Как видно из графика, наибольшее покрытие по коду обеспечивается при вероятности сбоя 1%. Результаты запуска с такой вероятностью сбоя и сравнивались с результатами испытаний остальных методов.

*Табл. 2. Применение методов симуляции сбоев для тестов Xfstests*

| Метод симуляции сбоев | Прирост покрытия, строк кода | Время тестирования, мин | Стоимость прироста покрытия, мин/строк | Дисперсия прироста покрытия, строк кода |
|---|---|---|---|---|
| (без симуляции) | - | 2 | - | - |
| *fsim_all* (теор. оценка) | - | 10 000 | - | - |
| *fsim_rnd* (p=1%) | 427 | 137 | 0,32 | 47,2 |
| *fsim_stack_1* | 397 | 64 | 0,16 | 7,3 |
| *fsim_stack_set* | 439 | 209 | 0,48 | 13,6 |

Как видно из таблицы, прирост покрытия с использованием детерминированных методов симуляции сбоев не сильно отличается от

прироста покрытия при использовании рандомизированного метода. Но дисперсия - показатель воспроизводимости тестирования - у детерминированных методов значительно меньше (в 4 раза у метода *fsim_stack_set*). Различия между детерминированными методами, *fsim_stack_1* и *fsim_stack_set*, заключаются в существенно ускоренном тестировании (в 3 раза) первым из них, но в немного большем приросте покрытия от второго (30 строк кода или 8%).

## 6. Сравнение методов тестирования с симуляцией сбоев

Объединяя результаты испытаний с нашими рассуждениями, получим следующую таблицу, отражающую сравнительные характеристики каждого из метода.

*Табл. 3. Сравнение методов тестирования с симуляцией сбоев*

| Критерий | *fsim_rnd* | *fsim_restricted* | *fsim_stack_1* | *fsim_stack_set* |
|---|---|---|---|---|
| Простота разработки и поддержки тестов | + | ± | + | + |
| Полнота тестирования (прирост покрытия) | + | ± | ± | + |
| Стоимость прироста покрытия | ± | ∓ | + | ± |
| Стабильная воспроизводимость тестовых сценариев | ∓ | ± | ± | ± |

Хотя стоимость прироста покрытия для метода *fsim_restricted* по абсолютной величине (0,30. см. табл. 1) примерно совпадает с соответствующей характеристикой метода *fsim_rnd* (0,32, см. табл. 2), эти величины получены для разных тестовых наборов (Spruce и Xfstests соответственно). А характеристики методов *fsim_stack_1* и *fsim_stack_set* в применении к этим наборам сильно отличаются. Поэтому отличаются и сравнительные оценки методов *fsim_restricted* и *fsim_rnd* по этой характеристике.

Как видно из сравнительной таблицы, метод однократной симуляции сбоев с различными модификациями является хорошей альтернативой рандомизированному методу, обеспечивая гораздо большую воспроизводимость тестовых сценариев. Из минусов этого метода существенным является его требование детерминированности базовых тестов. Хотя в случае небольших отклонений от этого требования метод не теряет своей эффективности (см. табл. 2).

Различия между модификациями метода однократной симуляции сбоев более тонкие и прослеживаются из численных результатов апробации (табл. 1):

- *fsim_restricted* обеспечивает лучшее качество тестирования(полноту), требуя взамен (ручное) выделение «уникальных» участков кода в базовых тестах, что довольно трудоемко.
- Даже в случае небольших ошибок при выделении "уникальных" участков кода тестов в методе *fsim_restricted*, этот метод может проиграть по качеству тестирования методам *fsim_stack_1* и *fsim_stack_set*, которые вообще не требуют ручной предобработки базовых тестов.

Выбор между методами *fsim_stack_set* и *fsim_stack_1* – это выбор между немного лучшим покрытием по коду, обеспечивающимся первым из них, и существенно лучшей эффективностью второго.

## *7. Заключение*

В статье были рассмотрены методы тестирования драйверов ОС Linux с симуляцией сбоев, позволяющие проверять работу драйвера в ситуациях, редко встречающихся в реальной работе. Существующий метод, использующий рандомизированные сценарии сбоев, обеспечивает достаточно качественное тестирование, но его недетерминированная природа не позволяет делать выводы о корректности протестированного драйвера. Для решения этих проблем в статье предлагаются методы, основанные на систематическом переборе точек возможной симуляции. Хотя полный перебор таких точек в тестах является неэффективным, методы, основанные на несложных ограничениях этих точек симуляции, вполне применимы на практике и по качеству сравнимы с рандомизированным методом. Что и было подтверждено экспериментами, проведенных на двух существующих тестовых наборах.

## Список литературы

[1]. А.В. Цыварев, В.А. Мартиросян. Тестирование драйверов файловых систем в ОС Linux. Труды Института системного программирования РАН, том 23, 2012 г. ISSN 2079-8156. Стр. 413-426.

[2]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.

[3]. KEDR Project, http://linuxtesting.org/kedr.

[4]. Е.А. Герлиц, В.В. Кулямин, А.В. Максимов, А.К. Петренко, А.В. Хорошилов, А.В. Цыварев. Тестирование операционных систем. Труды Института системного программирования РАН, том 26, Выпуск 1, 2014 г. ISSN 2079-8156. Стр. 73-108.

[5]. Linux File System Verification Project, http://linuxtesting.org/spruce

[6]. Xfstests source code, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git

# Using Fault Injection for Testing Linux Kernel Components

[1]*A.Tsyvarev <tsyvarev@ispras.ru>*
[1,2,3,4]*A.Khoroshilov <khoroshilov@ispras.ru>*
[1]*Institute for System Programming of the RAS,*
*25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University,*
*GSP-1, Leninskie Gory, Moscow, 119991, Russia.*
[3] *Moscow Institute of Physics and Technology (State University)*
*9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics (HSE)*
*11 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

**Abstract.** The paper considers methods of using fault injection for testing components of Linux kernel. The main goal of the methods is to check how the kernel behaves in abnormal situations such as lack of resources and hardware faults. Such situations happens quite rarely and unpredictably, that makes very difficult to detect and to localize bugs in the code responsible for their handling. The most widely-used approach to find the bugs is random fault injection during execution of normal tests. Random fault injection works much better than absence of abnormal testing at all, but it has a number of drawbacks. Its random nature does not allow to reproduce test scenarios and it does not allow to estimate absence of bugs even if all tests are passed.

The paper presents a method of systematic fault injection for robustness testing that also enables reproducibility of test scenarios out of box.The basic idea of the method is to detect all execution points where a fault can happen, to filter out the points on the base of some equivalence relation and to execute tests bringing code into the chosen execution points with injecting faults at that points.Applicability and efficiency of the approach depends on the equivalence relation. The equivalence relation can be defined using manual markup of important segments of tests or it can be defined completely automatically on the base of a set of stack traces describing a fault injection.Experimental data were collected using test suites for Linux kernel file system implementations augmented with random and systematic fault injection.The experiments demonstrate the efficiency of systematic approach and acknowledge benefits coming from its deterministic nature.

**Keywords:** Linux; Linux kernel; file system; driver; testing; fault simulation.

**DOI:** 10.15514/ISPRAS-2015-27(5)-9

**For citation:** A.Tsyvarev, A.Khoroshilov. Using Fault Injection for Testing Linux Kernel Components. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 157-174 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-9.

## References

[1]. A.V. Tsyvarev, V.A. Martirosyan. Testirovanie drajverov fajlovyh sistem v OS Linux [Testing of Linux File System Drivers]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2012, vol. 23, pp. 413-426 (in Russian). doi: 10.15514/ISPRAS-2012-23-24.

[2]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.

[3]. KEDR Project, http://linuxtesting.org/kedr.

[4]. E.A. Gerlits, V.V. Kuliamin, A.V. Maksimov, A.K. Petrenko, A.V. Khoroshilov, A.V. Tsyvarev. Testirovanie operacionnyh sistem [Testing of Operating Systems]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*], 2014, vol. 26, n.1, pp. 27-72 (in Russian). doi: 10.15514/ISPRAS-2014-26(1)-3.

[5]. Linux File System Verification Project, http://linuxtesting.org/spruce

[6]. Xfstests source code, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git