DOI: 10.15514/ISPRAS-2026-38(6)-22



SyzAnalyzer: Инструмент анализа трудновоспроизводимых ошибок в ядре Linux

П.А. Никульшин, ORCID: 0009-0008-5801-5922 <p.nikulshin@ispras.ru>
Е.В. Дубич, ORCID: 0009-0009-7215-9946 <e.dubich@ispras.ru>
А.В. Хорошилов, ORCID: 0000-0002-6512-4632 <khoroshilov@ispras.ru>

Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

НИУ Высшая школа экономики,
101978, Россия, г. Москва, ул. Мясницкая, д. 20.

Аннотация. В современном мире операционные системы, использующие ядро Linux, по праву занимают важное место, поскольку широко используются в самых разных сферах: от высоконагруженных серверных систем до бытовой техники. Ядро Linux активно разрабатывается, вследствие чего размер его кодовой базы превысил сорок миллионов строк, что, в свою очередь, не может не приводить к появлению внушительного количества ошибок и уязвимостей. Данный факт является угрозой для безопасности и стабильности работы систем на базе ядра Linux. Одним из широко распространенных средств обнаружения ошибок ядра Linux является фаззинг-тестирование, которое основано на генерации программ, выполняющих большое количество псевдослучайных системных вызовов. Наиболее популярным инструментом фаззинг-тестирования для операционных систем является Syzkaller. В случае выявления аварийных завершений ядра фаззер пытается сгенерировать так называемые программы-репродьюсеры, позволяющие воспроизвести обнаруженную ошибку, однако большое количество найденных ошибок является трудно воспроизводимыми. Syzkaller не имеет достаточного количества инструментов для автоматизации процесса отладки трудно воспроизводимых ошибок и поиска границ их воспроизведения. Это значительно уменьшает шанс того, что ошибка будет исследована и исправлена. В данной работе мы представляем инструмент SyzAnalyzer, который предназначен для автоматизации процесса анализа трудновоспроизводимых ошибок. Инструмент позволяет сократить усилия, необходимые для локализации таких проблем.

Ключевые слова: тестирование; фаззинг; ядро Linux.

Для цитирования: Никульшин П.А., Дубич Е.В., Хорошилов А.В. SyzAnalyzer: Инструмент анализа трудновоспроизводимых ошибок в ядре Linux. Труды ИСП РАН, 2026, том 38, вып. 6, часть 2, стр. 7--20. DOI: 10.15514/ISPRAS-2026-38(6)-22.

SyzAnalyzer: A tool for analysing difficult-to-reproduce errors in the Linux kernel

P.A. Nikulshin, ORCID: 0009-0008-5801-5922 <p.nikulshin@ispras.ru>
E.V. Dubich, ORCID: 0009-0009-7215-9946 <e.dubich@ispras.ru>
A.V. Khoroshilov, ORCID: 0000-0002-6512-4632 <khoroshilov@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

National Research University, Higher School of Economics,
20, Myasnitskaya st., Moscow, 101978, Russia.

Abstract. In today's world, the Linux operating system holds a significant position, being utilized in a wide range of applications, from high-performance server systems to consumer devices. The Linux kernel is actively being developed, and as a result the size of its code base has exceeded forty million lines. This, in turn, can lead to a significant number of bugs and vulnerabilities, which poses a critical risk to the security and stability of Linux-based systems. One of the popular methods of error detection is fuzzing testing, which is based on the generation of programs that perform a large number of pseudorandom system calls. The most popular fuzzing testing tool for operating systems is Syzkaller. When a kernel crash is detected, the fuzzer tries to generate so-called reproducer programs that allow reproducing the detected error. However, a significant number of the bugs found are challenging to reproduce. Syzkaller currently lacks the tools to automate the process of verifying the reproducibility of errors and identifying the commit that triggered them. This significantly reduces the likelihood that errors will be investigated and resolved. In this paper, we present SyzAnalyzer, which is designed to automate the process of analyzing difficult-to-reproduce errors. The tool allows reducing the effort required to localize such errors and has already allowed to debug several difficult-to-reproduce errors.

Keyword: testing; fuzzing; Linux kernel.

For citation: Nikulshin P.A., Dubich E.V., Khoroshilov A.V. SyzAnalyzer: A tool for analysing difficult-to-reproduce errors in the Linux kernel. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 6, part 2, pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-22.

1. Введение

Операционные системы являются ключевым элементом современного программного обеспечения. Ядро Linux используется в ряде наиболее активно разрабатываемых и широко используемых операционных систем, которые применяются в различных областях, включая серверные системы, мобильные устройства, интернет вещей и автомобили. Размер кодовой базы ядра Linux превышает 40 миллионов строк кода, в написании которых участвует более 29 тысяч разных авторов по состоянию на конец 2024 года [1]. Такие масштабы неизбежно приводят к возникновению большого количества ошибок, для обнаружения которых применяются различные методы, включая фаззинг-тестирование.

Фаззинг – это метод тестирования, основанный на автоматической генерации большого количества псевдослучайных входных данных. Этот метод позволяет выявлять неочевидные ошибки, и потому применяется в различных областях разработки и исследованиях. Для тестирования ядер операционных систем наиболее распространенным фаззером на данный момент является Syzkaller [2]. Инструмент генерирует поток программ. Если происходит падение, то Syzkaller берёт журнал выполненных программ и пытается их минимизировать, сгенерировав специальную программу, позволяющую воспроизвести найденное срабатывание ошибки. В рамках данной статьи для обозначения таких программ будет использоваться термин репродьюсер. Если репродьюсер не удалось получить, то журнал выполненных программ можно также использовать для воспроизведения ошибки. С помощью Syzkaller удалось обнаружить и исправить более 7 тысяч уязвимостей [3].

Зачастую из собранной при фаззинге информации об ошибке эксперту понятна причина ее возникновения, поэтому исправление таких ошибок происходит достаточно быстро. Однако значительная часть обнаруживаемых ошибок остается трудной для анализа. В одних случаях репродьюсер не удается получить вовсе, и тогда в распоряжении специалиста остаются журналы ядра и фаззера, содержащие большое количество программ, не обязательно связанных с анализируемой проблемой. В других случаях даже при наличии репродьюсера ошибка может воспроизводиться крайне редко. Подобные ошибки требуют значительных временных затрат эксперта на проверку воспроизводимости и локализации проблемы и зачастую остаются без исправлений.

Практическая сложность возрастает ввиду того, что специалисту приходится работать не с одной конкретной ошибкой, а сразу с большим набором срабатываний. Для каждой такой ошибки успех анализа не гарантирован: часть случаев может требовать слишком большого числа запусков для воспроизведения, часть может зависеть от конкретной версии или конфигурации ядра, а часть – не воспроизводиться вовсе. В этих условиях становится критически важным выделить те срабатывания, дальнейшее исследование которых с наибольшей вероятностью приведет к исправлениям этих ошибок. Такой отбор вручную очень времязатратен. Для анализа перспективности каждой из ошибок специалисту приходится вручную настраивать виртуальные машины, запускать на них инструмент воспроизведения, передавать ему репродьюсеры и многократно перезапускать в надежде получить информацию об искомом срабатывании. Если репродьюсер отсутствует, к этому алгоритму добавляется необходимость вручную разбирать журнал выполнения и осуществлять ручную проверку всех содержащихся в нем программ. При этом редкие срабатывания фаззера могут оказаться не связанными с исследуемой ошибкой и быть вызваны сторонними факторами. В этом случае время специалиста будет потрачено впустую.

В данной работе предлагается метод автоматизации анализа трудновоспроизводимых ошибок ядра Linux, ориентированный на работу с большим набором срабатываний. Предлагаемый подход включает в себя предварительный анализ воспроизводимости ошибки и последующую локализацию границ воспроизводимости для оценки перспективности ее исследования.

2. Текущее состояние проблемы

На практике работа с инфраструктурой фаззера Syzkaller зачастую начинается с отчетов о найденных ошибках, формируемых в ходе непрерывного фаззинг-тестирования различных веток ядра Linux (рис. 1).

Для обнаруженных в ходе фаззинга ошибок сохраняется следующий набор данных: версия ядра, параметры сборки, отчет об ошибке, журнал выполненных программ и в ряде случаев программа-репродьюсер, вызвавшая срабатывание ошибки, на языке C или специально разработанном для Syzkaller языке системных вызовов syz-lang [4].

В некоторых случаях информации из отчета бывает достаточно для понимания причины срабатывания ошибки и поиска ее источника. Однако для значительной части срабатываний дальнейший анализ затруднен без дополнительных данных.

При этом Syzkaller нацелен на проведение непрерывного фаззинг-тестирования и хорошо подходит для первичного обнаружения ошибок, в то время как задача более детального анализа срабатываний остается на стороне специалиста. Особую сложность представляют ситуации, когда воспроизвести срабатывание ошибки затруднительно или не удается вовсе.

Данное явление называется недетерминированностью ошибки. У недетерминированности есть несколько причин:

1) Состояния гонки и гонки данных [5]. Гонка данных (data race) возникает, когда два или более потока одновременно обращаются к одной области памяти без синхронизации, и хотя бы одно из обращений является записью. В таком случае

корректность выполнения программы зависит от неконтролируемого порядка выполнения потоков. Поскольку этот порядок может меняться от запуска к запуску, проявление подобных ошибок носит недетерминированный характер, что значительно усложняет их воспроизведение и отладку.

2) Мусорные данные (garbage data). В неинициализированной структуре или переменной могут содержаться «случайные» биты, оставшиеся от предыдущих данных.

3) Тайминги выполнения. Переключения контекста и выполнение процессов на различных физических ядрах могут приводить к различным путям выполнения программы, что в ряде случаев приводит к возникновению ошибки.

4) Остаточные состояния. Возможна ситуация, при которой ошибка произошла из-за совокупности остаточных эффектов от работы предыдущих программ.



Рис. 1. Пример неисправленной ошибки от Syzkaller.
Fig. 1. An example of an uncorrected error from Syzkaller.

Часто для анализа недетерминированных ошибок базовой информации из отчета Syzkaller бывает недостаточно для исправления, и эксперту приходится проводить дополнительные исследования воспроизводимости ошибки. Сложность анализа дополнительно возрастает, когда для срабатывания не удается получить репродьюсер. Тогда специалисту приходится работать не с готовым репродьюсером, а с множеством программ из журнала выполнения, поочередно проверяя каждую. Кроме того, иногда срабатывания Syzkaller могут формально классифицироваться как разные ошибки, хотя на деле являются проявлениями одной и той же ошибки.

На текущий момент при необходимости проверить воспроизводимость ошибки в ядре Linux необходимо вручную организовать инфраструктуру и выполнить ряд рутинных шагов:

1) Подготовить тестовую среду. Требуется выбрать множество версий и конфигураций ядра, собрать образ ОС и настроить виртуальную машину для каждой пары версия–конфигурация.

2) Развернуть инструмент запуска. Чаще всего выполняют анализ ошибок с доступным репродьюсером на C. При этом компилируется код и результирующий исполняемый

файл копируется на каждую виртуальную машину. Альтернативно можно перенести программу syz-execprog и репродьюсер на syz-lang.

3) Выполнять репродьюсеры, фиксируя журналы выполнения. Как правило требуется многократный запуск репродьюсеров в течение от десятков до сотен тысяч раз.
4) Проверять результаты, сопоставив возникшие сообщения об ошибках с искомыми.
5) Выполнить анализ полученных результатов и частоты воспроизведения ошибки.

Также в составе Syzkaller есть инструмент syz-verifier [4], предназначенный для дифференциального фаззинга. Он генерирует общий поток программ и выполняет их на разных версиях ядра, после чего сравнивает возвращаемые системными вызовами коды ошибок. Однако такой подход ориентирован в первую очередь на выявление расхождений в поведении ядер, а не на анализ конкретных срабатываний [6].

В последнее время активно развивается подход к автоматическому анализу и исправлению ошибок с использованием больших языковых моделей. Примером такого инструмента является syz-agent [7], который анализирует найденные сбои, ранжирует их по степени «опасности», генерирует объяснение произошедшей ошибки и предлагает исправление. Это помогает получить дополнительную информацию об ошибке в виде экспертной оценки от искусственного интеллекта. Для решения проблемы ранжирования данный инструмент плохо подходит, ведь позволяет разделить ошибки лишь на две большие группы: достаточно ли имеющейся информации для понимания причины ошибки или нет.

Проверка воспроизводимости для одной версии ядра сама по себе не является финальным результатом анализа. В случае подтверждения воспроизводимости важно также исследовать ее границы: найти версию, внесшую ошибку, и проверить, есть ли у ошибки исправление в новых версиях ядра. В Syzkaller реализована встроенная функция бисекции по версиям и даже по конфигурациям ядра, однако она нацелена на работу с ошибками с подтвержденной воспроизводимостью, требует наличие репродьюсера и не позволяет задать конкретную разновидность ошибки для поиска.

Практическая сложность дополнительно возрастает из-за того, что эксперту приходится работать не с одной конкретной ошибкой, а сразу с большим набором срабатываний. Для каждой ошибки успех анализа не гарантирован: часть из них может требовать слишком большого числа запусков для воспроизведения, часть – не иметь репродьюсеров, часть – зависеть от конкретной версии или конфигурации ядра, часть – проявляться с другим именем, а часть – не воспроизводиться вовсе. В условиях, когда набор исследуемых ошибок растет, становится крайне важным выделить те срабатывания, дальнейшее исследование которых с большим шансом позволит найти причины ошибки и исправить ее.

Таким образом, возникает потребность в разработке инструмента, который позволит автоматически анализировать большое количество срабатываний и помогать эксперту находить наиболее перспективных кандидатов для дальнейшего исследования. Такой инструмент должен автоматизировать выполнение детерминации (проверки воспроизводимости ошибки) и бисекции (бинарного поиска по истории репозитория) ошибок, а также их ранжирования по перспективности. Кроме того, инструмент должен позволять выявлять различные проявления одной и той же ошибки, которые Syzkaller может классифицировать как разные, и предоставлять возможность работы напрямую с файлами журналов даже при отсутствии репродьюсера.

3. Предлагаемое решение

В работе представлен метод, основанный на комбинировании автоматической детерминации и бисекции множества целевых ошибок. На первом этапе каждая из ошибок проходит процесс детерминации путём множественных запусков репродьюсеров на разных сборках ядра. При подтверждении воспроизводимости ошибки инструмент переходит к бисекции. На

заданном диапазоне версий ядра осуществляется поиск границ, в которых данная ошибка воспроизводится. По результатам работы инструмента проводится ранжирование ошибок по перспективности их дальнейшего анализа.

В качестве входных данных передаются ошибки, их репродьюсеры или журналы и одна или несколько веток ядра Linux, для которых указывается конфигурация сборки и три версии: изначальная для детерминации, более старая и новая для поиска границ воспроизводимости.

Для каждой ошибки в первую очередь проверяется её воспроизводимость, то есть факт того, что ошибку удаётся воспроизводить на изначальной версии (см. Алгоритм 1). Для этого запускается её детерминация на изначальной версии. После сборки ядра указанной конфигурации, репродьюсер многократно выполняется на множестве виртуальных машин. Если удалось хотя бы один раз получить искомую ошибку за выбранное количество запусков, значит ошибка воспроизводится.

Алгоритм 1: Алгоритм Детерминации

Input: список ядер K, список репродьюсеров R, количество запусков T, целевые ошибки F

Output: среднее количество запусков до воспроизведения R, найденные целевые ошибки E, остальные найденные ошибки Eo

```
foreach k ∈ K do
    if not k.is_build() then        // Собрать ядро, если ещё не собрано
        k.build()
    end
    for t ← 1 to T do
        e ← k.run(R)                // Получить результат выполнения
        if e.is_target(F) then
            E ← E ∪ {e}             // Сохранить целевые ошибки
        else then
            Eo ← Eo ∪ {e}           // Сохранить остальные ошибки
        end
    end
end
R ← T / |E|
return R, E, Eo
```

Рис. 2. Алгоритм Детерминации.
Fig. 2. The Algorithm of Determination.

Для количественной оценки трудности воспроизведения используется среднее количество запусков для воспроизведения ($\bar{R}$) – отношение общего числа запусков репродьюсера к количеству запусков, воспроизводящих ошибку. Помимо этого, выполнение репродьюсера может вызвать иную ошибку, не идентичную исследуемой. В таком случае после проверки экспертом данная ошибка может быть помечена как иное проявление исследуемой ошибки, и результаты детерминации могут быть автоматически пересчитаны.

Если воспроизводимость ошибки подтверждена, следующим шагом является определение границ воспроизводимости, то есть в каком диапазоне версий ядра Linux ошибка воспроизводится. Нижняя граница указывает на версию, на которой начала воспроизводиться

ошибка. Это позволяет эксперту проанализировать внесенные в этой версии изменения и оценить их связь с возникновением ошибки. Поиск же верхней границы показывает была ли ошибка потенциально исправлена в новых версиях. Однако отсутствие воспроизведения после некоторой версии нельзя автоматически считать исправлением, так как оно может быть вызвано побочными изменениями в ядре. Для поиска границ применяется алгоритм бинарного поиска по истории репозитория, реализованный в команде git bisect [8]. Бисекция проводится в двух направлениях от изначальной версии: в сторону наиболее свежей версии ветки, чтобы проверить наличие потенциального исправления ошибки, и в сторону старой версии ветки, чтобы найти момент, когда ошибка начала проявляться. На каждой итерации бинарного поиска выполняется сборка ядра для проверяемой версии, после чего проводится детерминация путём многократного запуска репродьюсеров. Количество запусков определяется средним количеством запусков для воспроизведения ошибки ($\bar{R}$). В зависимости от того, удалось ли воспроизвести ошибку на данной версии, бинарный поиск продолжается в соответствующей половине диапазона, как схематично показано на рис. 3.

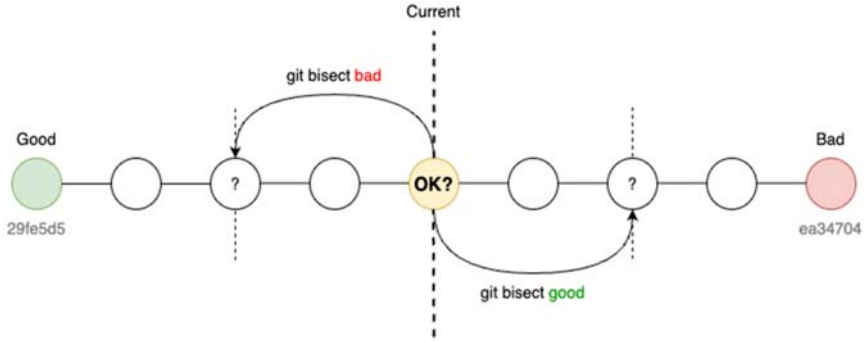


Рис. 3. Принцип работы git bisect.
Fig. 3. How git bisect works.

После завершения анализа ошибки вычисляется оценка приоритетности её исследования (W), которая позволяет эксперту упорядочить ошибки по порядку дальнейшего исследования. Показатель основывается на частоте срабатывания ошибки и найденных в ходе бисекции границ воспроизводимости:

$$W = B + \frac{1}{\bar{R}},$$

где $\bar{R}$ – среднее количество запусков для воспроизведения ошибки, B – показатель, отражающий результаты поиска границы воспроизводимости:

$B = 0$ – ошибка воспроизводится на всём заданном диапазоне;

$B = 1$ – найдена версия, на которой ошибка начала воспроизводиться (нижняя граница);

Все исследуемые ошибки упорядочиваются по оценке приоритетности (W), что позволяет эксперту определить очерёдность их дальнейшего изучения.

Полученные результаты требуют дальнейшей обработки экспертом. В частности, если для ошибки обнаружена версия с вероятным исправлением, эксперту рекомендуется дополнительно проверить корректность этого исправления.

В процессе многократного воспроизведения ошибки могут быть зафиксированы и иные её проявления – срабатывания с текстами сообщений, отличными от исходного, которые стандартными средствами Syzkaller были бы классифицированы как самостоятельные ошибки. Разработанный инструмент регистрирует все обнаруженные в ходе запусков сообщения об ошибках. Если эксперт подтверждает, что некоторое из них действительно

является иными проявлением той же самой исследуемой ошибки, то инструмент автоматически пересчитывает статистику воспроизводимости, включая эти срабатывания в число целевых. Поскольку расширение множества проявлений ошибки может повлиять на вывод о её воспроизводимости на других версиях ядра, инструмент повторно выполняет бисекцию с учётом обновлённых данных.

4. Реализация

Итоговое решение реализовано на языке программирования Go [9] с частичным заимствованием компонентов из кодовой базы Syzkaller. Архитектура построена по модульному принципу и включает следующие основные компоненты (рис. 4):

- **Analyzer** – главная точка входа, через которую пользователь взаимодействует с инструментом. Получает от пользователя набор исследуемых ошибок, параметры сборки ядер, количество запусков и другие настройки, после чего делегирует выполнение соответствующим подсистемам.

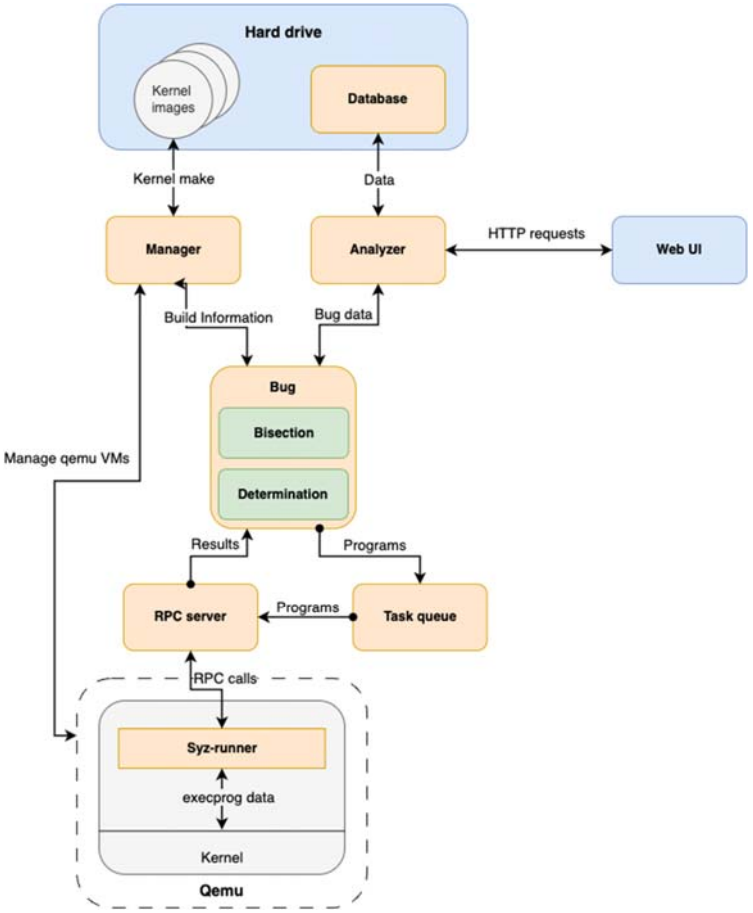


Рис. 4. Архитектура SyzAnalyzer.
Fig. 4. The SyzAnalyzer architecture.

- **Manager** – отвечает за сборку ядер Linux и управление виртуальными машинами. На основе переданных параметров (ссылка на репозиторий, версия, конфигурационный файл) клонирует исходный код и компилирует образ ядра.
- **Database** – обеспечивает долговременное хранение данных, полученных в процессе работы инструмента. Используется SQLite[10] база данных.
- **Bug** – реализует логику исследования отдельной ошибки. Координирует этапы детерминации и бисекции: запрашивает у Manager нужные сборки ядра, формирует задачи для Task Queue, анализирует поступающие отчёты о выполнении и принимает решение о дальнейших действиях.
- **Task Queue** –- реализует очередь репродьюсеров для каждой виртуальной машины.
- **RPC server** – обрабатывает RPC запросы от виртуальных машин.
- **Syz-runner** – запускает программы на виртуальной машине с помощью встроенной в Syzkaller программы syz-execprog и возвращает результат их выполнения.
- **Web UI** – реализует взаимодействие пользователя с инструментом.

В реализации частично переиспользуются компоненты Syzkaller. В частности, модуль Manager использует пакет vm для создания и управления виртуальными машинами, а компонент Syz-runner выполняет репродьюсеры внутри виртуальной машины с помощью утилиты syz-execprog.

5. Эксперимент

5.1 Цель эксперимента и постановка задачи

Целью эксперимента является оценка применимости разработанного инструмента для анализа набора трудновоспроизводимых ошибок ядра Linux и ранжирования по перспективности дальнейшего исследования. В данной работе инструмент рассматривается как средство первичного отбора и упорядочивания большого набора срабатываний.

Для экспериментальной оценки была сформирована выборка из 37 ошибок. Среди них 14 были класса WARNING, general protection fault – 10, KASAN – 6, BUG – 3, RCU stall – 2, NULL dereference – 1, UBSAN – 1. Большинство ошибок относились к файловым системам и сетевой подсистеме ядра: 16 и 13 соответственно. Каждая ошибка исследовалась на трех ветках ядра Linux: 5.10, 6.1, 6.12. Для каждой ветки было выбрано по 3 версии: изначальная версия для изучения, более старая и новая версия для поиска границы воспроизводимости.

Таблица 1. Использованный для тестирования версии ядра.
Table 1. Kernel commits used in testing.

Новая версия	v5.10.250-lvc74	v6.1.163-lvc46	v6.12.73-lvc14
Изначальная версия	v5.10.246-lvc66	v6.1.152-lvc33	v6.12.44
Старая версия	v5.10.176-lvc1	v6.1	v6.12.1

Выборка ошибок для анализа формировалась случайным образом без предварительной фильтрации. Количество первичных запусков на изначальной версии ядра для каждой ошибки было выбрано равным 100 000. Это значение является порогом того, сколько запусков мы были готовы потратить на каждую ошибку, при этом достаточным для получения результатов воспроизводимости. В ходе запусков фиксировались следующие результаты: запуск без ошибки, целевое срабатывание искомой ошибки и расширенное срабатывание целевой ошибки с новым заголовком.

Количество запусков для тех ошибок, воспроизведение которых удавалось получить близко к заданному числу первичных запусков, увеличивалось. Если ошибка показывала устойчивую воспроизводимость, то её обработка могла останавливаться ранее заданного

числа запусков с целью экономии времени эксперта. Для бисекции границ воспроизводимости ошибки репродьюсер запускался трехкратное среднее количество запусков для воспроизведения ($\bar{R}$) на каждой версии.

5.2 Результаты

В ходе эксперимента было выполнено 10615493 запуска репродьюсеров на различных сборках ядра. Выполнение детерминации одной ошибки на одной версии ядра в среднем занимало 3,5 часа. На бисекцию в одну сторону в среднем было затрачено 15 часов, из которых на анализ одной версии уходило порядка 1,5 часов.

Из 37 проанализированных ошибок воспроизводимость хотя бы на одной из 3 веток ядра была подтверждена для 11 ошибок, что составляет 29,7% выборки. Для остальных 26 ошибок не было зафиксировано ни одного срабатывания целевой ошибки за отведенное время ни на одной из веток. Таким образом, детерминация сократила на 71,3% множество кандидатов для дальнейшего углубленного анализа.

Отсутствие воспроизведения для большей части выборки может объясняться как требованием на порядок большего числа запусков для воспроизведения, так и зависимостью ошибок от специфичных условий машины, на которой эти ошибки были получены.

Среди подтвержденных ошибок наблюдался значительный разброс показателя среднего количество запусков для воспроизведения ($\bar{R}$): от 1,02 до 217482 запусков. Это свидетельствует о высокой неоднородности недетерминированных ошибок.

Рис. 5 показывает вероятность целевого срабатывания для каждой подтвержденной ошибки на исходных версиях для каждой из исследованных веток. Для одной части ошибок вероятность воспроизведения близка к 1, что соответствует устойчивой воспроизводимости. Для другой напротив вероятность крайне мала. Кроме того, часть ошибок демонстрирует воспроизводимость на конкретных ветках ядра при полном отсутствии целевых срабатываний на других ветках. Для подобных ошибок становится особенно важным выбор ветки для дальнейшего исследования, включая локализацию границ воспроизводимости.

На рис. 6 приведены данные о распределении $\bar{R}$ для подтвержденных ошибок на каждой из протестированных веток. Кривые демонстрируют различия в сложности детерминации между ветками. Для ветки 6.12 заметно, что среднее число запусков каждой из ее ошибок не превышает 100, тогда как для 5.10 и 6.1 почти половина ошибок расположена на диапазоне от 100 до 10000, и имеются особо трудновоспроизводимые случаи с $\bar{R}$ больше 100000. Таким образом, выбор ветки для анализа может влиять и на скорость детерминации набора.

После детерминации для подтвержденных ошибок для каждой из подтвержденных веток была проведена бисекция с целью поиска границ воспроизводимости (рис. 7). В общей сложности было получено 24 результата бисекции, для 11 из которых были найдены обе границы воспроизводимости, для 7 – только верхняя граница, для 2 – только нижняя, и для 4 границы установить не удалось (табл. 2).

На основе полученных результатов, для каждой воспроизведенной ошибки была рассчитана числовая оценка приоритетности (W), основанная на двух факторах: результате бисекции и трудности воспроизведения. На рис. 8 левой и правой квадратными скобками показаны найденные нижняя и верхняя границы воспроизводимости соответственно, а числовым значением показана сама оценка W. Эта оценка позволяет сортировать ошибки в зависимости от объема важных для исправления ошибки данных. Подобное ранжирование помогает эксперту сконцентрировать усилия на тех срабатываниях, дальнейший анализ которых с наибольшей вероятностью приведет к выявлению и исправлению ошибки.

Помимо количественных оценок, инструмент помог выявить ряд особенностей, важных для дальнейшего анализа. В частности, был зафиксирован случай изменения текста ошибки при переходе между версиями. Одна и та же программа-репродьюсер на ветках 5.10 и 6.1 стабильно вызывала срабатывание с заголовком “KASAN: use-after-free Read in

jffs2_garbage_collect_pass", тогда как на ветке 6.12 сообщение об ошибке изменилось на "KASAN: slab-use-after-free Read in jffs2_garbage_collect_pass".

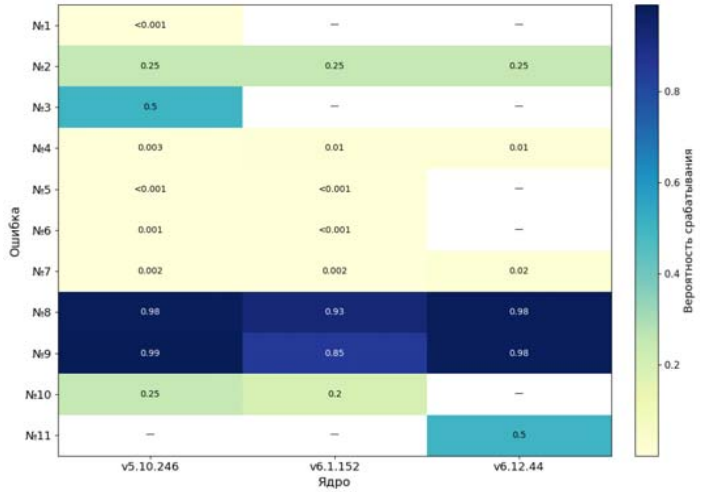


Рис. 5. Вероятность воспроизведения ошибок для различных веток ядра Linux.
Fig. 5. The probability of reproducing errors for different branches of the Linux kernel.

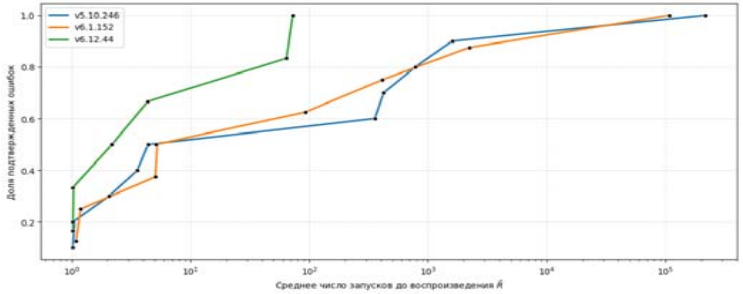


Рис. 6. Распределение трудности воспроизведения ошибок для различных веток ядра Linux.
Fig. 6. Distribution of reproduction difficulty across different Linux kernel branches.

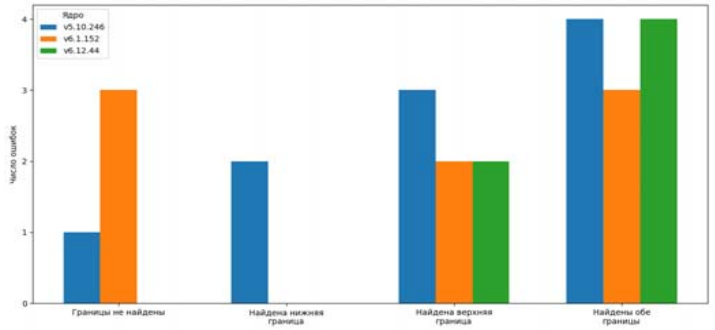


Рис. 7. Результаты бисекции.
Fig. 7. Bisection results.

Таблица 2. Общие результаты для веток 5.10, 6.1 и 6.12.
Table 2. General results for branches 5.10, 6.1 and 6.12.

Ядро	5.10.246			6.1.152			6.12.44		
№ ошибки	$\bar{R}$	Приоритет	Ранг	$\bar{R}$	Приоритет	Ранг	$\bar{R}$	Приоритет	Ранг
1	1598	1,00063	6	-	-	-	-	-	-
2	4	1,25	2	4	1,25	1	4	1,25	2
3	2	1,5	1	-	-	-	-	-	-
4	356	1,0028	3	92	1,011	2	73	1,014	4
5	217482	1,0000046	7	107320	0,0000093	9	-	-	-
6	783	1,0013	5	2220	0,00045	7	-	-	-
7	424	1,0024	4	412	1,0024	3	64	1,016	3
8	1,02	0,98	9	1,08	0,93	4	1,02	0,98	5
9	1,01	0,99	8	1,18	0,85	5	1,02	0,98	5
10	4	0,25	10	5	0,2	6	-	-	-
11	-	-	-	-	-	-	2	1,50	1

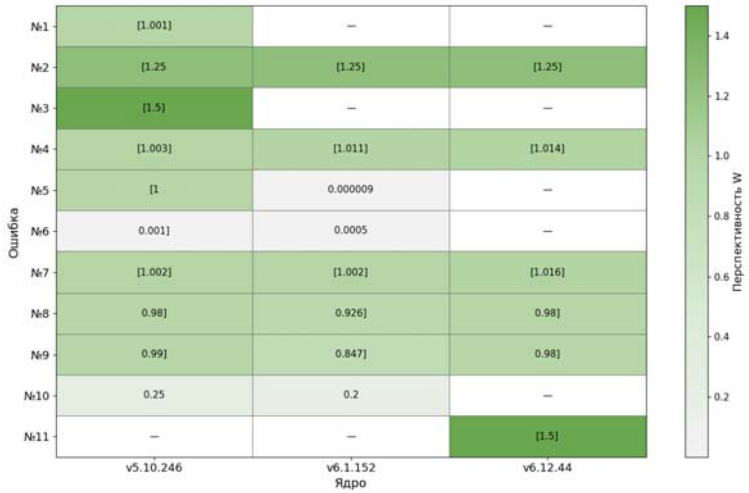


Рис. 8. Приоритизация ошибок.
Fig. 8. Error prioritization criteria.

Другой эффект, обнаруженный для той же самой ошибки: на всех исследуемых ветках ядра репродьюсер вызвал до 6 различных по тексту сообщений об ошибке. Дополнительные проявления были найдены также еще в 3 ошибках. При использовании стандартных инструментов Syzkaller эти ошибки были бы классифицированы как разные. Разработанный инструмент позволил выяснить, что эти сообщения являются разным проявлением одной и той же ошибки. Отметим, что подобное поведение может быть вызвано и наличием

нескольких разных ошибок, которые проявляются при выполнении одного репродьюсера, но данный фактор пока остается вне рамок данной работы, и данные о дополнительных проявлениях предоставляется эксперту для верификации.

До использования разработанного инструмента весь набор из 37 ошибок для эксперта представлял собой идентичные, с точки зрения потенциала к решению, ошибки, после проведённого анализа эксперт получил множество дополнительных данных об ошибке и теперь может приоритизировать с каких из изученных ошибок стоит начать дальнейший анализ и исправление.

6. Заключение

В работе предложен и программно-реализован метод автоматизированного анализа трудновоспроизводимых ошибок ядра Linux, основанный на последовательном выполнении детерминации и бисекции для набора целевых ошибок. Разработанный инструмент позволяет в автоматическом режиме оценивать воспроизводимость ошибок, определять границы их проявления и ранжировать их по перспективности дальнейшего исследования, а также выявлять различные текстовые проявления одной и той же ошибки, которые штатными средствами Syzkaller были бы классифицированы как независимые. В рамках апробации инструмента была проанализирована выборка из 37 ошибок. Для 11 из них (29,7%) воспроизводимость была подтверждена хотя бы на одной из трех исследованных веток ядра, что позволило сократить множество кандидатов для углубленного анализа более чем на 70%. Для подтвержденных ошибок была выполнена бисекция. На основе полученных данных рассчитана количественная оценка приоритетности, позволяющая эксперту обоснованно выбирать очередность дальнейшей работы. Дополнительно инструмент выявил несколько случаев, в которых один и тот же репродьюсер вызывал до шести различных сообщений об ошибках. Это позволяет рассматривать их как разные проявления одной ошибки, а не как независимые ошибки.

Полученные результаты демонстрируют, что предложенный метод применим для практического анализа трудновоспроизводимых ошибок ядра Linux и может способствовать повышению эффективности работы экспертов за счет автоматизации рутинных операций и предоставления структурированной информации о перспективности дальнейшего анализа каждой ошибки.

6.1 Направления развития

Для предотвращения чрезмерной нагрузки на систему, вызванной параллельным выполнением множества виртуальных машин, планируется разработка планировщика ресурсов, который будет балансировать нагрузку и гарантировать стабильность работы всей системы.

Планируется сосредоточиться на локализации недетерминированных ошибок. Основная идея заключается в применении и адаптации методов спектральной локализации к данным о покрытии ядра, собранным в ходе множества запусков репродьюсеров. Целью является создание методики, которая, несмотря на недетерминизм проявления ошибки, сможет статистически достоверно указывать на потенциально проблемные участки кода ядра.

Список литературы / References

[1]. Larabel M. The Linux Kernel Hit A Decade Low In 2024 For The Number Of New Commits Per Year. Доступно по ссылке: https://www.phoronix.com/news/2024-Linux-Git-Stats, дата обращения 20.05.2025.

[2]. Google. Syzkaller unsupervised coverage-guided kernel fuzzer. Доступно по ссылке: https://github.com/google/syzkaller, дата обращения: 20.05.2025.

[3]. Google. Syz-verifier Documentation. Доступно по ссылке: https://github.com/google/syzkaller/blob/master/docs/syz_verifier.md, дата обращения: 20.05.2025.

[4]. Google. Syscall descriptions syntax. Доступно по ссылке: https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md, дата обращения: 20.05.2025.

[5]. Pedro Jos´e Sousa da Fonseca. Effective Testing for Concurrency Bugs. Thesis for obtaining the title of Doctor of Engineering Science of the Faculties of Natural Science and Technology I of Saarland University, 2015. DOI: 10.22028/D291-26605.

[6]. Mara Mihali. Detecting semantic bugs using differential fuzzing. Linux Plumbers 2021. Доступно по ссылке: https://lpc.events/event/11/contributions/1033/attachments/742/1621/syz-verifier%20-%20Linux%20Plumbers%202021.pdf, дата обращения: 22.01.2026.

[7]. Li, Xie & Yuan, Zhaoyue & Zhang, Zhenduo & Sun, Youcheng & Zhang, Lijun. (2025). Towards Large Language Model Guided Kernel Direct Fuzzing. DOI: 10.48550/arXiv.2503.02301.

[8]. Git Bisect Documentation. Доступно по ссылке: https://git-scm.com/docs/git-bisect, дата обращения: 25.02.2026.

[9]. Go Documentation. Доступно по ссылке: https://go.dev/doc/, дата обращения: 15.03.2026.

[10]. Sqlite Documentation. Доступно по ссылке: https://sqlite.org/docs.html/, дата обращения: 15.03.2026.

Информация об авторах / Information about authors

Павел Андреевич НИКУЛЬШИН – лаборант ИСП РАН, выпускник бакалавриата ФКН НИУ ВШЭ. Сфера научных интересов: операционные системы, безопасность операционных систем, фаззинг.

Pavel Andreevich NIKULSHIN – research assistant at ISP RAS; graduate of the Faculty of Computer Science, HSE University. Research interests: operating systems, operating system security, and fuzzing.

Евгений Васильевич ДУБИЧ – лаборант ИСП РАН, выпускник бакалавриата ФКН НИУ ВШЭ. Сфера научных интересов: операционные системы, безопасность операционных систем, фаззинг.

Evgeniy Vasilievich DUBICH – research assistant at ISP RAS; graduate of the Faculty of Computer Science, HSE University. Research interests: operating systems, operating system security, and fuzzing.

Алексей Владимирович ХОРОШИЛОВ – кандидат физико-математических наук, ведущий научный сотрудник, руководитель Центра исследований безопасности системного программного обеспечения ИСП РАН, доцент кафедр системного программирования МГУ, ВШЭ и МФТИ. Основные научные интересы: методы проектирования и разработки ответственных систем, формальные методы программной инженерии, методы верификации и валидации, тестирование на основе моделей, методы анализа требований, операционная система Linux.

Alexey Vladimirovich KHOROSHILOV – Cand. Sci. (Phys.-Math.), Leading Researcher, Director of the Linux Verification Center at ISP RAS, Associate Professor of System Programming Departments at MSU, NRU HSE, and MIPT. Main research interests: design and development methods for critical systems, formal methods of software engineering, verification and validation methods, model-based testing, requirements analysis methods, Linux operating system.