DOI: 10.15514/ISPRAS-2025-37(4)-17

DIFFuzzer: Detecting File System Errors with Differential Grey-box Fuzzing

V.M. Kovalevsky, ORCID: 0009-0002-4501-9672 <slava0135@vk.com>
V.V. Kechin, ORCID: 0009-0006-0845-2740 <valeraghst@gmail.com>
V.M. Itsykson, ORCID: 0000-0003-0276-4517 <itsykson@yandex.ru>

ITMO University,
Kronverksky Pr. 49, bldg. A, St. Petersburg, 197101, Russia.

Abstract. File systems are a crucial component of any modern operating system, whether it is a general-purpose computing system or a specialized data storage system. The cost of a file system error is very high; consequently, there is a need for effective tools for analyzing quality and detecting errors in file systems. This paper presents the DIFFuzzer tool, which is based on fuzzing techniques using grey-box and black-box principles, and implements a differential dynamic analysis approach, where the behavior of the target file system is compared to that of another, known to be of higher quality file system, which serves as a generator of reference behavior. In comparing behaviors, both the response codes of system calls and the aggregated state of the file systems are analyzed. The toolkit also includes a reducer that minimizes the erroneous trace and generates a short fragment where the error still appears. The developed tool has been tested on several POSIX-compliant file systems and has discovered several errors even during a relatively short experiment.

Keywords: file systems; differential fuzzing; file system testing; grey-box fuzzing.

For citation: Kovalevsky V.M., Kechin V.V., Itsykson V.M. DIFFuzzer: Detecting File System Errors with Differential Grey-box Fuzzing. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 2, 2025, pp. 31-46. DOI: 10.15514/ISPRAS-2025-37(4)-17.

DIFFuzzer: обнаружение ошибок файловых систем с помощью дифференциального фаззинга серого ящика

В.М. Ковалевский, ORCID: 0009-0002-4501-9672 <slava0135@vk.com>
В.В. Кечин, ORCID: 0009-0006-0845-2740 <valeraghst@gmail.com>
В.М. Ицыксон, ORCID: 0000-0003-0276-4517 <itsykson@yandex.ru>

Университет ИТМО,
Россия, 197101, Санкт-Петербург, Кронверкский пр., д. 49, лит. А.

Аннотация. Файловые системы являются важнейшим компонентом любой современной операционной системы, будь то вычислительная система общего назначения или специализированная система хранения данных. Стоимость ошибки файловой системы очень высока; следовательно, существует потребность в эффективных инструментах для анализа качества и обнаружения ошибок в файловых системах. В данной статье представлен инструмент DIFFuzzer, который основан на методах фаззинга с использованием принципов серого и черного ящиков и реализует подход дифференциального динамического анализа, при котором поведение целевой файловой системы сравнивается с поведением другой, более качественной файловой системы, которая служит генератором эталонного поведения. При сравнении поведения анализируются как возвращаемые значения системных вызовов, так и агрегированное состояние файловых систем. Инструментарий также включает в себя редуктор, который минимизирует ошибочную трассу выполнения и генерирует короткий фрагмент, в котором ошибка все еще появляется. Разработанный инструмент был протестирован на нескольких файловых системах, совместимых с POSIX, и обнаружил несколько ошибок в ходе относительно короткого эксперимента.

Ключевые слова: файловые системы; дифференциальный фаззинг; тестирование файловых систем; фаззинг серого ящика.

Для цитирования: Ковалевский В.М., Кечин В.В., Ицыксон В.М. DIFFuzzer: обнаружение ошибок файловых систем с помощью дифференциального фаззинга серого ящика. Труды ИСП РАН, том 37, вып. 4, часть 2, 2025 г., стр. 31–46 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(4)–17.

1. Introduction

File systems embody one of the most complex components in modern computing systems, requiring extensive testing due to their critical role in data management and storage.

These considerations further emphasize the importance of rigorous testing. Most popular file systems like Ext4 have undergone extensive testing over many years of widespread use. However, traditional quality assurance methods may be insufficient for newer or specialized file systems. This is due to the vast number of possible system states and the complexity of operation sequences. Additionally, file systems are typically implemented in low-level programming languages, where memory-related errors are particularly common. For these reasons, there is a need to develop automated tools for ensuring correctness of file systems.

In this paper, we present a file system testing tool called DIFFuzzer based on differential and mutational fuzzing. The proposed approach considers the advantages and disadvantages of related works, while emphasizing the industrial relevance of the tool. Special attention is given to developing a tool that is both practical and applicable in industrial settings.

One of the main challenges was conducting an in-depth analysis of existing tools while balancing development speed with the exploration of innovative techniques. More details on related work are presented in section 3. While many tools adopt interesting and effective strategies, most are outdated or primarily academic in nature, limiting their real-world applicability. In our work, we aim to integrate these approaches into a user-friendly and efficient tool, while accounting for the constraints typical of industrial environments, such as time limitations and specific technology requirements.

The rest of the paper is organized as follows. In section 2 we state our objectives and talk about the prerequisites in detail. Research and analysis of existing works are given in section 3. Next section 4 describes problems in file system fuzzing. For example, we will elaborate on workload generation, bug detecting and reducing, execution of potential risky operations, etc. Section 5 focuses on describing our proposed approach to file system fuzzing. In section 6 we explain the details of architecture and implementation of the tool. Results of evaluation are presented in section 7. We make some conclusions, describe disadvantages and ways of development for the presented tool in section 8. Also, we share our thoughts on future bugs detection tools.

2. Motivation

The complexity of modern software development has grown considerably in recent years. At the same time, quality assurance practices have remained largely the same and often rely on unit tests and a limited number of integration or end-to-end test cases. While this may be sufficient in some cases, more sophisticated methods are necessary in highly complex projects such as file systems development, to effectively detect bugs. However, the application of scientific approaches in industrial development is not straightforward. These methods often require specialized knowledge and controlled environments, which complicates their integration into real-world systems.

Our work focuses specifically on detecting bugs in file system implementations. A file system is low-level software that interacts with both the operating system and the user. Any mistake in implementation may be critical, because file systems are used in nearly all modern computing devices.

3. Related work

In this section we review the most notable papers and tools in the domain of file system checking. Table 1 summarizes these works using criteria.

Table 1. Comparison of file system testing tools.

Name	Linux	File System Semantics	Grey-Box	FUSE	Bug Types
Syzkaller	6.14		+		Memory, Logic
kAFL	6.5		+	+	Memory, Logic
Metis	6.6	+		+	Memory, Logic, Spec
Dogfood	5.15	+		+	Memory, Logic, Crash Consistency
CrashMonkey	5.6	+			Crash Consistency
LFuzz	6.0	+	+		Memory, Logic
KRace	5.4	+	+		Memory, Logic, Data Races
Hydra	5.0	+	+	+	Memory, Logic, Spec, Crash Consistency
DIFFuzzer	6.14	+	+	+	Memory, Logic, Spec

3.1 Kernel fuzzers

Syzkaller [1] is a coverage-guided kernel fuzzer used extensively for Linux kernel testing. The fuzzer is structure-aware and uses language for modeling the kernel's programming interfaces. However, this language does not capture detailed file system semantics which makes it less effective for fuzzing file systems, as demonstrated in the Hydra study [2]. Another issue is that it uses KCov [3] for the feedback mechanism which is not suitable for FUSE [4] file systems. Also, Syzkaller focuses on finding non-semantic bugs such as memory errors that have a clear signal, and thus it fails to trigger deep semantic bugs in file systems.

kAFL [5] is another coverage-guided kernel fuzzer. As the name suggests, it is based on AFL [6], but it uses a special feedback mechanism based on Intel Processor Trace [7], which makes it suitable for fuzzing arbitrary binaries across different operating systems. Again, the main issue with general-purpose kernel fuzzers is that they lack a model of file system semantics and are thus significantly less effective. As with Syzkaller, it is only able to find memory and logic bugs.

3.2 Black-box file system fuzzers

Metis [8] is a tool for differential fuzzing. Authors introduce an approach based on the file system model created using the SPIN [9] model checker. Metis generates workloads via SPIN, executes them on two file systems, and compares the resulting abstract file system states.

We believe differential fuzzing is particularly well-suited to file system testing due to the stable and uniform nature of file system APIs. However, aside from our work it is the only tool known to apply this approach. Because of this black-box approach Metis is only able to find "shallow" bugs.

The authors of CrashMonkey [10] present a new approach to testing file system crash consistency by simulating power-loss crashes while the workload is being executed and checking if the file system recovers to a correct state after each crash. However, CrashMonkey uses Linux kernel version 5.6 (released in 2020), which is now outdated. This is a significant limitation, as the implementation depends on a loadable kernel module. Additionally, the workload length is limited to three operations (excluding dependencies), which prevents the generation of more complex inputs. CrashMonkey is black-box and tests are generated beforehand, but even for three operations it is not possible to generate or execute all possible workloads in a reasonable time.

Dogfood [11] is a related tool with a model-based approach for workload generation using layered file system models which fixes some of CrashMonkey limitations. However, like other tools, it is not capable of finding semantic bugs, except for crash consistency bugs, similar to CrashMonkey.

3.3 Grey-box file system fuzzers

Hydra [2] is a mutation fuzzing tool that provides a comprehensive approach for file system testing. It builds upon the fuzzing infrastructure provided by AFL. A distinguishing feature of Hydra is that input data exploration is done not just through mutation of system calls but also through direct mutation of metadata stored on disk. Hydra implements mechanisms for bug detection of various classes such as logic bugs, memory errors, crash inconsistency bugs, and specification violations.

Although Hydra is the most feature-complete file system fuzzer to date, it is difficult to use in practice due to maintainability issues. The tool relies on outdated infrastructure, including Linux kernel version 5.0 (released in 2019) and Ubuntu 16.04 (whose support ended in 2021). As a result, significant effort would be required to update Hydra to modern environments.

LFuzz [12] is another file system fuzzer that exploits file system localities, expanding on Hydra. LFuzz tracks recently accessed image locations and nearby locations to predict which locations will soon be referenced, making fuzzing more effective. Unlike Hydra, LFuzz can only find memory and logic bugs. Unfortunately, the source code is not available at the moment of writing.

Another tool, KRace [13], brings coverage-guided fuzzing to the concurrency dimension. This is the only tool that attempts to address the complex challenge of testing file systems in a parallel execution environment. However, the authors do not provide documentation or instructions for using the tool or reproducing their experiments. Several repository submodules are missing, making the tool impossible to build in its current state.

3.4 Summary

Overall, only two tools presented can find specification violation bugs, where the file system fails to adhere to agreed specifications (e.g., POSIX). One of them is black-box (Metis) and the other one has not been maintained for a long time (Hydra). While not as common as memory bugs, specification violation bugs can still cause serious system failures. Based on these observations, we decided to make our own tool called DIFFuzzer to fill the gap in finding deep semantic bugs using a differential grey-box fuzzing approach.

Another challenge is the frequent abandonment of research tools, even those that demonstrate promising results. Therefore, our work emphasizes practical applicability in industrial contexts, rather than producing one-time experimental results. Syzkaller has proven that it is possible to stay relevant in the field by providing continuous support and improvements, and we will try to do the same for DIFFuzzer.

4. Problem statement

The implementation of a practical file system fuzzing tool presents numerous conceptual and engineering challenges.

4.1 Tool usability and ease of use

Other tools created by the researchers provide novel and valuable approaches to file system testing. However, it is unrealistic to use their implementations during file system development due to the lack of documentation, complexity of configuration, and limited compatibility. In this work, we place particular emphasis on usability and maintainability to ensure the tool is suitable for practical use.

4.2 Workload generation

File system fuzzing can be done by performing some workload – a sequence of file operations like creating a new file, removing existing directory etc. However, the operations within a sequence must be carefully coordinated to maintain semantic validity. For instance, writing to a deleted file would result in an invalid sequence that produces errors, and should therefore be avoided in most test cases. Indeed, sometimes it is necessary to generate invalid operations for checking error handling, but generation of sequences still must be validated against specification. For instance, POSIX [14] describes approximately 40 operations for accessing a file system. Generating valid workloads that account for the interactions and dependencies among these operations presents a significant challenge.

4.3 Workload execution

Several key considerations must be addressed when implementing workload execution:

- The effectiveness of a fuzzer is closely related to the number of executions per second. However, this is constrained by the relatively slow performance of storage devices. One possible way to improve performance is to use temporary storage, e.g., RAM [15]. This also makes it easy to reset the file system state between executions.
- State reset between executions. It is necessary to isolate workload runs as much as possible, otherwise reproducing bugs can be impossible, due to incremental system state that is hard to restore. Reset can be done by loading a virtual machine snapshot after each run or just mounting and unmounting the file system.
- Workload execution can lead to kernel panic when fuzzing the kernel file systems. All results may be lost, and the fuzzing process will stop. Because of this, kernel fuzzers execute workloads inside virtual machines.

4.4. Unexpected behaviour detection

The simplest fuzzers detect bugs by observing whether a program crashes upon execution. More advanced fuzzers implement their own mechanisms for bug detection because program crashes are just one of the many common bug types. There is a broad spectrum of bug types [2] to consider when testing file systems:

- **Memory errors.** This includes *use-after-free*, *out-of-bounds*, *null-pointer-dereference* etc., errors commonly found in programs written in low-level programming languages with direct memory access. They can be easily detected at runtime with the help of compiler instrumentation (sanitizers).
- **Logic bugs.** Errors that are triggered by failed assertions about program invariants, for example, integer division by zero, and are easy to identify as well.
- **Specification violations.** Cases where the file system fails to adhere to agreed specifications (e.g., POSIX [14]). Violations of the specification can lead to incorrect operation of application programs and are more difficult to identify.
- **Crash inconsistencies.** Cases when the system does not recover to the correct state after a failure. A file system is crash consistent if it is always restored to the correct state after a system crash due to power loss or an operating system kernel panic. Requires special facilities to detect, thus very hard to find.
- **Race conditions.** Cases where two or more threads gain unmanaged access to a shared memory without proper write synchronization or ordering. Extremely hard to detect due to the non-deterministic nature of thread interleaving.

4.5 Execution feedback

In order for a grey-box fuzzer to determine which workloads from corpus are interesting and are worth mutating later, it needs some kind of guidance or feedback, usually, code coverage.

4.6 Workload mutation

In grey-box fuzzing mutations allow to guide fuzzer towards uncovered code. Workloads that discover new coverage should be saved to the corpus for further mutations. The main challenge is making sure that mutations do not produce invalid workloads. However, in some cases, generating invalid operations is useful for testing error-handling behavior. Nevertheless, failed operations typically do not alter the file system state. Therefore, to reduce the search space and minimize workload length, it is generally preferable to avoid generating invalid operations.

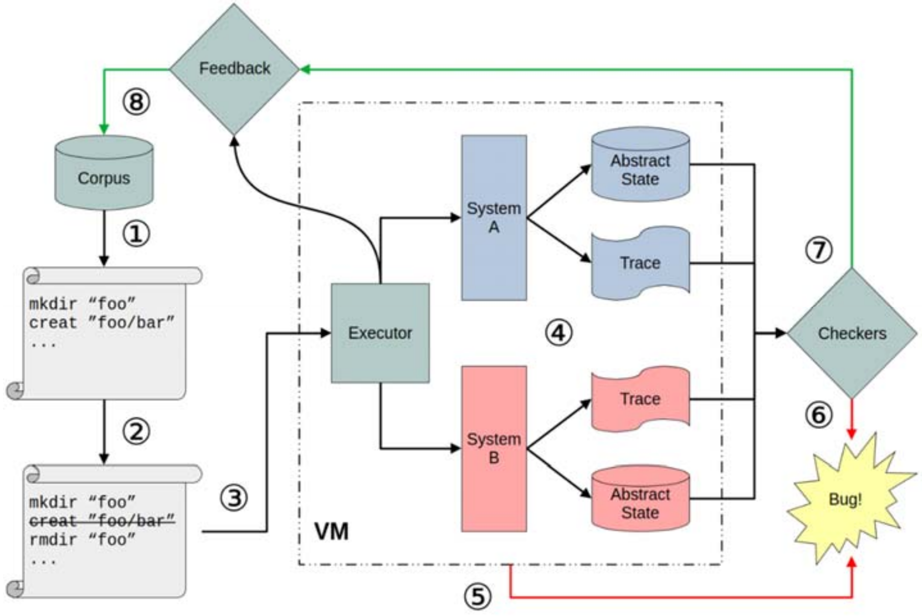
4.7 Workload reduction

Workloads that trigger bugs may include numerous unrelated operations. In practical scenarios, these workloads should be minimized to help developers localize the root cause of the bug more efficiently. However, removing or simplifying operations may alter the bug's manifestation, potentially complicating its diagnosis.

5. Approach

5.1 Overview

Fig. 1 illustrates the overall workflow of DIFFuzzer. In each fuzzing iteration DIFFuzzer loads a workload from corpus (1). The workload is then mutated (2) and encoded into a C program. The program is sent to the VM and is compiled together with Executor library (3). Compiled test cases are executed on both target and reference file systems (4). If VM panics or reboots, then the potential bug is reported (5). Otherwise, execution traces and abstract states are sent to checkers. If inconsistencies between the file systems are detected, a bug is reported (6). Alternatively, feedback from the Executor is analyzed (7). If new code coverage is discovered, the workload is saved to corpus (8) to guide further exploration. Otherwise, workload is discarded. Reduction and replication are not performed during fuzzing but can be done afterwards.



```
mkdir “foo”
creat ”foo/bar”
...
```

```
mkdir “foo”
~~creat ”foo/bar”~~
rmdir “foo”
...
```

Fig. 1. Overview of DIFFuzzer workflow.

5.2 Workload generation

As was already discussed, a sequence of file operations must respect both implicit and explicit dependencies between operations. To address this, we built an abstract file system model that approximates real file system behavior. The model is capable of generating valid file operation sequences of arbitrary length. It also records all generated operations, which can later be replayed or used to generate test cases for execution on actual file systems.

The generated workloads can be exported in JSON format for internal serialization, or as C programs, which are compiled with the Executor to produce executable test binaries.

5.3 Workload execution

Most file system operations can only be invoked through the system’s C API. For this reason, we developed a C++ Executor library that provides the file system API with some additional features and is used by generated C workloads.

Workloads are executed inside a virtual machine (VM), as test execution may trigger system panics or reboots – events that would be disruptive in non-isolated environments. Another reason is that using a VM allows us to set up an instrumented kernel with code coverage and sanitizers without modifying the host system. This also makes it easy to test file systems on different versions of the kernel.

5.4 Unexpected behavior detection

As was established earlier, file systems are susceptible to a wide range of bug types. Detecting memory and logic bugs is relatively straightforward, because runtime errors cause system panic (in kernel space) or program crash (in user space).

In contrast, identifying semantic bugs is significantly more challenging. Detecting specification violations requires evaluating the system’s behavior against the expected specification. However, no formal specification model currently exists that can be directly utilized by automated tools. SibylFS [16] attempted to create a formal specification for file systems, but it is no longer maintained and is incompatible with modern systems.

To avoid developing a complete formal file system API specification, one can use another implementation of the same specification to detect violations using differential testing. The core idea is conceptually simple: execute identical inputs on two different file system implementations and compare their outputs. However, specifications often permit multiple valid outputs for a given input. Therefore, strict output equivalence may lead to false positives during comparison

Output comparison involves analyzing system call exit codes, error codes, and the resulting abstract file system state. The abstract state may include attributes such as file paths, sizes, link counts, permissions, and user/group identifiers – combined to form a unified representation of the file system.

Detecting crash consistency bugs and data races is significantly more difficult and remains outside the scope of this paper.

5.5 Execution feedback

Similar to many other grey-box mutational fuzzers, we use code coverage as a feedback mechanism for selecting interesting inputs. There are many ways to collect code coverage:

- **Instrumented binary.** Compiler inserts tracking code at the end of every basic block of program – linear sequence of program statements. However, kernel instrumentation cannot be performed in the same manner as for user space applications. Instead, Linux provides its own code coverage mechanism (KCov [3]), which is used in some other kernel fuzzers like Syzkaller [1]. However, this approach has notable drawbacks: it increases binary size and RAM consumption and can reduce performance by over 100%.
- **Intel Processor Trace.** Technology available in modern Intel CPUs [7] that allows efficient tracing of all the instructions executed by a process. Its main advantages include low performance overhead (less than 5%) and applicability to closed-source targets, including operating systems. However, as demonstrated in kAFL [5], this method requires extending both QEMU [17] and KVM [18].
- **QEMU CPU emulation.** Used in TriforceAFL [19] and able to capture coverage for any binary, but with one major drawback: it can be up to 50 times slower, compared to native performance.

We chose to use instrumented binaries, as extending QEMU and KVM to support Intel Processor Trace would introduce significant engineering overhead. Additionally, QEMU-based emulation is prohibitively slow for file system fuzzing, operating at an order of magnitude below user space fuzzers. KCov is limited to kernel-space file systems. However, we also aim to support FUSE-based [4] file systems, which can be implemented in various programming languages and may require different coverage mechanisms. Currently, we support LCov [20] coverage for C++ and C, but we plan to add support for other coverage formats as well.

5.6 Workload mutation

Workloads are sequences of file operations. Selected workloads are mutated by inserting or removing operations at specific positions in the sequence. Mutations are implemented by replaying sequences in the abstract file system model before the point, applying the mutation at the index and replaying the second half of the workload after the point. If the replay results in an error, the mutation is discarded and a new mutation is attempted. This ensures that mutated workloads are valid, unless there is a bug in the model itself. Bugs in the model can be also found by fuzzing. If both file system traces contain errors, then it can be considered a bug in the model.

5.7 Workload reduction

Bug reduction consists of trying to remove a single operation from the end at a time. Removing operations starting from the end in succession is enough because each operation can only depend on previous operations. If the reduction can remove an operation without affecting the result, it is considered valid. This, in turn, may enable the removal of earlier operations. Sometimes, removing an operation may reveal a different bug than the original one. We suggest keeping all the bug variants. They are saved and can be checked manually later, but only the original bug is reduced further. It should be noted that if a bug is found by difference between trace exit codes, then we can trim workload up to that operation and go to the described algorithm.

6. Implementation

Implementation consists of three major parts:

- **Executor.** C++ runtime, used to execute tests.
- **Dash.** Differential abstract state hasher, used to compare file system states, written in Rust.
- **DIFFuzzer.** Fuzzer itself, also written in Rust.

We picked Rust for our tools, because fuzzing benefits from high execution speed, and Rust performance is on par with C++, which is commonly used for implementing fuzzers. However, C++ is not very ergonomic and developer experience with Rust is better, especially when using developer tools.

Another reason is that in earlier stages of development we used LibAFL [21], fuzzing framework made in Rust, in order to reuse some existing fuzzing components. Unfortunately, we quickly discovered it wasn't very suitable for our specific case.

Executor and Dash are reusable components that can operate independently and potentially can be integrated into other file system testing frameworks.

6.1 Executor

Generated workloads produce two types of files. JSON files are used for serialization and deserialization in order to save and load tests from disk. C files are used exclusively for execution and are linked against runtime to produce a test binary. These operations do not directly invoke system calls. Instead, they call a runtime library that wraps system functionality, tracks operation results (e.g., exit and error codes), and collects kernel coverage via KCov [3].

There are minor differences between real operations and those provided by the runtime. For instance, according to POSIX [14], the *creat* function should leave the file descriptor open, similar to *open* function. However, our runtime closes it immediately, as writing to the file is not always necessary. Often, file system bugs just require the file entry to exist in the directory.

The runtime is implemented in C++, unlike other components, because Linux interfaces, especially KCov functions, are not trivial to use in any other language.

At the moment Executor supports the following operations: *mkdir*, *creat*, *rmdir*, *unlink*, *symlink*, *hardlink*, *rename*, *open*, *close*, *read*, *write*, *fsync*, *truncate*, *ftruncate*, *lseek*.

6.2 Dash

Dash is a differential abstract state hasher, implemented in Rust as a standalone binary. It is used to compare states between file systems, similar to how it's done in Metis [8]. There are some subtleties, though. For example, different file systems may use different block sizes, resulting in variations in file sizes. Another issue is that directory hard links are not supported by any major file system, and instead the hard link count can represent the number of files in directory – as is the case with Ext4, but not Btrfs, for instance. Because of this, some properties can be optionally ignored by changing configuration when required.

Dash represents the abstract state by calculating a hash for the entire file system, recursively processing each directory. All files are sorted by their relative paths. If files have different properties or if a file is missing in one file system, the hash will differ, and the fuzzer will detect this unexpected behavior. Additionally, the abstract state is saved in JSON format, allowing the root cause of the issue (e.g., a missing file) to be identified in the bug report.

6.3 DIFFuzzer

DIFFuzzer is the core component of the system. Its functionality is best explained by outlining the steps it performs from start to finish.

At the startup DIFFuzzer launches a QEMU [17] instance, using a preconfigured Linux OS image. The image must have a kernel built with custom configuration that enables KCov coverage collection and kernel memory bug detection. The image should also have necessary libraries and packages installed, such as *make* and a C++ compiler. For convenience, we provide detailed documentation on setting up QEMU and Linux images.

Communication with VM is handled via the QEMU Monitor Protocol (QMP) [22] and SSH [23]. QMP is used to receive events about the VM state and jobs in progress, as well as to save and load snapshots. SSH is used to execute remote commands and copy files between host and VM.

DIFFuzzer waits a fixed amount of time for the VM to boot. It then copies the testing runtime and compiles it. After this a VM snapshot is saved using QMP. The snapshot serves as a clean state that is used to restore the VM whenever it panics or reboots, avoiding the need to set up everything from scratch.

At this point fuzzer is ready to begin generating and executing workloads. The fuzzer starts with an empty corpus, containing an empty workload. The corpus is a collection of seeds – workloads that are considered interesting, for example, if their execution covered new code paths. Optionally, the corpus can be loaded from a directory with tests generated from previous runs to speed up state exploration.

At the start of each fuzzing iteration the fuzzer picks a random seed from the corpus. There are many ways to choose seeds. Most primitive way is to use a queue and pick seeds in a circle. However, this is not very effective, and a different strategy is used in our implementation. As explained in the AFLFast paper [24] an efficient coverage-based grey-box fuzzer prioritizes inputs that have not been fuzzed very often and inputs that exercise low-frequency paths. This approach allows for faster exploration of newly discovered code regions.

After workload is picked, random mutations are applied to it. Mutated workload is translated to a C program and copied to VM. The program is then compiled and run by a test harness.

The test harness is the component of the system responsible for running tests. First it sets up a file system by mounting it, then the test is executed and at the end the file system is unmounted. Additionally, other components can observe or change the state before and after each execution using callbacks. If the VM panics or reboots during execution, the state is restored from the previously saved snapshot. To speed up execution, file systems are created on a special storage device – block RAM device (BRD) [15], provided by a standard loadable Linux kernel module.

Each test is executed for both file systems. Each run can result in one of three outcomes: successful execution, a timeout after a fixed period, or a VM panic or reboot. If both runs complete successfully then fuzzer compares the traces and abstract states of two file systems, reporting a bug if they are different. Otherwise, no additional bug detection is performed, and the timeout or panic is reported.

If no unexpected behavior is observed, the workload is evaluated by a feedback mechanism. If executing the workload covers new code paths, it is added to corpus for future use.

The cycle repeats: a new workload is selected, mutated, executed, and analyzed.

Reduction is not performed during fuzzing, as it is time-consuming and fuzzing already suffers from inherently long execution times. Instead, test cases can be reduced in a separate mode by reading them from saved JSON files.

6.4 Usage and extensibility

DIFFuzzer is not only a grey-box fuzzer, but it can also perform black-box fuzzing, launch single tests for one or two file systems, and reduce found bug cases – all within a single tool. Most internal components are shared across these modes, which simplifies the architecture and improves robustness.

DIFFuzzer can be easily configured with a single configuration file. For example, the user can configure the number of mutations, mutation and operation weights, and workload length, among other options. DIFFuzzer is also easy to extend: new file systems can be added by just implementing a single Rust trait (interface). Support for seven kernel file systems (Ext4, Btrfs, XFS, F2FS, BcacheFS, JFS, NILFS2) and one FUSE-based [4] file system (LittleFS [25]) was added with minimal effort.

For DIFFuzzer using we've prepared detailed documentation, manuals and script examples.

7. Evaluation

We evaluated DIFFuzzer on a host machine running Ubuntu 22.04 and Ubuntu 24.04 server cloud images inside a VM with an 8-core Intel Core i7-10700 2.90GHz CPU and 32 GB RAM. A single VM uses 2 virtual cores and 4 GB RAM.

Most of the processor time is spent executing tests. Within a single VM, it is possible to execute approximately 2.5 tests per second using non-instrumented kernel and about 1.5 tests per second with instrumented kernel (with KCov and sanitizers enabled) on both file systems, tested on Ext4 and Btrfs. These numbers vary across file systems; test execution for a single file system is roughly twice as fast.

```
#include "executor.h"

int fd_0;

void test_workload()
{
    do_mkdir("1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
    fd_0 = do_create("1/2", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
    do_remove("1/2");
    do_remove("1"); // Directory not empty(39)
    do_close(fd);
}
```

Listing 1. Test where LittleFS is unable to remove directory with open file.

We discovered three specification violation bugs in the FUSE [4] file system LittleFS [25] in fewer than 1000 executions. The first bug (Listing 1) occurs when LittleFS fails to remove a directory if a file within it is open, even if *unlink* function was called successfully before.

```
#include "executor.h"

int fd_0, fd_1;

void test_workload()
{
    do_create("/1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
    fd_0 = do_open("/1");
    do_rename("/1", "/2");
    // "do_write" writes 10 bytes from a buffer filled
    // with random data starting from index 0
    do_write(fd_0, 0, 10);
    do_close(fd_0);
    fd_1 = do_open("/2");
    do_read(fd_1, 10); // 0 bytes read, expected 10
}
```

Listing 2. Test where file content is lost when renaming open file before writing.

```
Index,Command,ReturnCode,Errno,Extra
    0, CREATE,   4,Success(0),
    1,   OPEN,   4,Success(0),
    2, RENAME,   0,Success(0),
    3,  WRITE,  10,Success(0),
    4,  CLOSE,   0,Success(0),
    5,   OPEN,   4,Success(0),
    6,   READ,  10,Success(0),hash=2fd…
```

```
Index,Command,ReturnCode,Errno,Extra
    0, CREATE,   4,Success(0),
    1,   OPEN,   4,Success(0),
    2, RENAME,   0,Success(0),
    3,  WRITE,  10,Success(0),
    4,  CLOSE,   0,Success(0),
    5,   OPEN,   4,Success(0),
    6,   READ,   0,Success(0),hash=1
```

Listing 3. Execution traces generated for Ext4 (left) and LittleFS (right).

The second bug (Listing 2) appears when data is written to a file using an open file descriptor; if the file was renamed prior to the write, the data is silently lost. This can be seen from comparing traces (Listing 3).

```
#include "executor.h"

int fd_0;

void test_workload()
{
    do_create("/1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
    fd_0 = do_open("/1", O_RDWR);
    // fills file with 64 zero bytes
    do_ftruncate(fd_0, 64);
    // `SEEK_END` sets descriptor offset to LENGTH + N
    do_lseek(fd_0, 0, SEEK_END); // offset is 0, expected 64
}
```

Listing 4. Test where descriptor file length is not updated after truncation.

The third bug (Listing 4) is triggered by truncating a file (extending file with zeros) with an open file descriptor and causes the file descriptor to use old file length instead of new. All bugs were detected using differential fuzzing by comparing LittleFS behavior against Ext4, which allows DIFFuzzer to identify inconsistencies that most other tools may miss. The bugs are reported on the project GitHub repository [26-28].

Several bugs were also discovered in a proprietary file system, which was one of the main goals of this research. These issues, primarily related to specification violations, were identified through differential fuzzing. All bugs were confirmed by the developers and subsequently fixed. Unfortunately, we don't have permission to share more details about the results.

After two months of intermittent testing – totaling about 10 million executions – DIFFuzzer did not discover any bugs when comparing Ext4 against Btrfs (Linux 5.15) and Ext4 against BcacheFS (Linux 6.14). One possible explanation is insufficient time; since fuzzing is inherently time-consuming, it may require tens or hundreds of millions of executions to uncover a single hidden bug. Another possible reason is that we used the latest available kernel versions, in which many previously reported bugs may have already been fixed. This could also explain why recent work such as Metis [8] failed to identify bugs in Ext4 and Btrfs, even after billions of executions. These results suggest a need to improve workload mutation strategies and enhance error detection mechanisms.

Ext4 was selected as the default reference file system due to its extensive testing history and relatively high performance. However, other file systems can also be used as reference targets.

In summary, the primary goal of this work has been achieved. However, further development is necessary. While the proposed approach successfully identifies bugs, additional efforts are required to improve the tool's effectiveness – whether by refining the current approach, optimizing performance, or incorporating techniques from related work.

8. Conclusion

In this paper, we present DIFFuzzer, a differential grey-box fuzzer designed to find bugs in file system implementations. It detects memory errors, logic errors, and specification violations, all of which may compromise the correct functioning of operating systems and applications. Unlike existing tools for file system testing, DIFFuzzer combines differential and grey-box fuzzing. This allows it to detect behavioral inconsistencies deep within the execution paths of two file systems, without requiring a formal specification model. In our experiments, DIFFuzzer discovered bugs in the evaluated file systems. Some of these bugs are difficult or impossible to detect using most existing tools. Therefore, we believe that DIFFuzzer represents a valuable contribution to file system development and testing. It is worth noting that we did not find bugs in widely used file systems like Ext4 and Btrfs. However, these results also highlight the tool's limitations and indicate the need for further improvements.

References

[1]. (2015) Syzkaller: coverage-guided kernel fuzzer. Google. Accessed: 2025-03-29. [Online]. Available: https://github.com/google/syzkaller.

[2]. S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in Proceedings of the 27th ACM Symposium on Operating Systems Principles, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 147–161. [Online]. Available: https://doi.org/10.1145/3341301.3359662.

[3]. KCov: code coverage for fuzzing. The Linux Foundation. Accessed: 2025-03-29. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kcov.html.

[4]. FUSE. The Linux Foundation. Accessed: 2025-03-29. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[5]. S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted feedback fuzzing for OS kernels," in 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[6]. Afl. Google. Accessed: 2025-03-29. [Online]. Available: https://github.com/google/AFL.

[7]. (2013) Intel processor trace. Intel Corporation. Accessed: 2025-03-29. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[8]. Y. Liu, M. Adkar, G. Holzmann, G. Kuenning, P. Liu, S. A. Smolka, W. Su, and E. Zadok, "Metis: File system model checking via versatile input and state exploration," in 22nd USENIX Conference on File and Storage Technologies (FAST 24). Santa Clara, CA: USENIX Association, Feb. 2024, pp. 123–140. [Online]. Available: https://www.usenix.org/conference/fast24/presentation/liu-yifei.

[9]. Spin. Accessed: 2025-03-29. [Online]. Available: https://spinroot.com/spin/whatispin.html.

[10]. J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "CrashMonkey and ACE: Systematically testing file-system crash consistency," ACM Trans. Storage, vol. 15, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3320275.

[11]. D. Chen, Y. Jiang, C. Xu, X. Ma, and J. Lu, "Testing file system implementations on layered models," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1483–1495. [Online]. Available: https://doi.org/10.1145/3377811. 3380350.

[12]. W. Liu and A.-I. A. Wang, "LFuzz: Exploiting locality-enabled techniques for file-system fuzzing," in Computer Security – ESORICS 2023, G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis, Eds. Cham: Springer Nature Switzerland, 2024, pp. 507–525.

[13]. M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRace: Data race fuzzing for kernel file systems," in 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1643–1660.

[14]. "IEEE/Open Group standard for information technology–portable operating system interface (POSIX™) base specifications, issue 8," IEEE/Open Group Std 1003.1-2024 (Revision of IEEE Std 1003.1-2017), pp. 1–4107, 2024.

[15]. Using the RAM disk block device with linux. The Linux Foundation. Accessed: 2025-03-29. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/blockdev/ramdisk.html.

[16]. T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell, "SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–53. [Online]. Available: https://doi.org/10.1145/2815400.2815411

[17]. QEMU: a generic and open source machine emulator and virtualizer. Accessed: 2025-03-29. [Online]. Available: https://www.qemu.org.

[18]. Kernel virtual machine. Accessed: 2025-03-29. [Online]. Available: https://linux-kvm.org/page/MainPage.

[19]. (2016) TriforceAFL. NCC Group. Accessed: 2025-03-29. [Online]. Available: https://github.com/nccgroup/TriforceAFL.

[20]. LCov. Accessed: 2025-03-29. [Online]. Available: https://github.com/linux-test-project/lcov.

[21]. A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602.

[22]. QEMU machine protocol. Accessed: 2025-03-29. [Online]. Available: https://wiki.qemu.org/Documentation/QMP.

[23]. OpenSSH. OpenBSD Foundation. Accessed: 2025-03-29. [Online]. Available: https://www.openssh.com.

[24]. M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428.

[25]. (2017) LittleFS-FUSE. Accessed: 2025-03-29. [Online]. Available: https://github.com/littlefs-project/littlefs-fuse.

[26]. LittleFS-FUSE issue tracker. Renaming open file causes written data to be lost. Accessed: 2025-04-11. [Online]. Available: https://github.com/littlefs-project/littlefs-fuse/issues/78.

[27]. LittleFS-FUSE issue tracker. Removing directory with unlinked open file fails. Accessed: 2025-04-11. [Online]. Available: https://github.com/littlefs-project/littlefs-fuse/issues/79.

[28]. LittleFS-FUSE issue tracker. Descriptor file length is not updated after truncation. Accessed: 2025-05-02. [Online]. Available: https://github.com/littlefs-project/littlefs-fuse/issues/81.

Информация об авторах / Information about authors

Вячеслав Максимович КОВАЛЕВСКИЙ – инженер Института прикладных компьютерных наук университета ИТМО. Сфера научных интересов: динамический и статический анализ программ, фаззинг.

Vyacheslav Maksimovich KOVALEVSKY – engineer of the Institute of Applied Computer Science of the ITMO University. Research interests: dynamic and static software analysis, fuzzing.

Валерий Владимирович КЕЧИН – инженер Института прикладных компьютерных наук университета ИТМО. Сфера научных интересов: динамический и статический анализ программ, фаззинг.

Valeriy Vladimirovich KECHIN – engineer of the Institute of Applied Computer Science of the ITMO University. Research interests: dynamic and static software analysis, fuzzing.

Владимир Михайлович ИЦЫКСОН – кандидат технических наук, доцент Института прикладных компьютерных наук университета ИТМО. Сфера научных интересов: статический и динамический анализ программ, верификация программного обеспечения, методы обнаружения дефектов в исходном коде, методы автоматизации тестирования программ.

Vladimir Mikhailovich ITSYKSON – Cand. Sci. (Tech.), associate professor of the Institute of Applied Computer Science of the ITMO University. Research interests: static and dynamic software analysis, software verification, methods for detecting defects in source code, methods for automating software testing.