DOI: 10.15514/ISPRAS-2026-38(5)-6

DIFFuzzer: One More Step Toward Specification-Based File System Quality Checking

A.S. Yanin, ORCID: 0009-0002-2627-8281 <slowmime@gmail.com>
V.M. Itsykson, ORCID: 0000-0003-0276-4517 <itsykson@yandex.ru>
V.V. Kechin, ORCID: 0009-0006-0845-2740 <valeraghst@gmail.com>
V.M. Kovalevsky, ORCID: 0009-0002-4501-9672 <slava0135@vk.com>

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

Abstract. We present a refinement of DIFFuzzer, a mutation-based differential file system fuzzer, by integrating formal specifications to improve its testing capabilities. Initially, DIFFuzzer relied on a reference-based approach and embedded POSIX behavior into the source code for workload generation, which limited its flexibility and scalability. Our improved approach decouples the fuzzer's logic from these rules by incorporating an external formal file system specification written in the LibSL language. This enables the creation of more precise test oracles and facilitates specification-driven workload generation. We detail the current state of the project, including the extension of LibSL and the implementation of a specification-based test oracle. We also discuss future directions and their associated challenges, such as the evolution of our POSIX formalization and the planned development of a specification-based workload generation module.

Keywords: model-based testing; fuzzing; formal specification; file systems; POSIX.

For citation: Yanin A.S., Itsykson V.M., Kechin V.V., Kovalevsky V.M. DIFFuzzer: one more step toward specification-based file system quality checking. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 5, 2026, pp. 91-102. DIFFuzzer: one more step toward specification-based file system quality checking. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 5, 2026, pp. 91-102. DOI: 10.15514/ISPRAS-2026-38(5)-6.

DIFFuzzer: шаг на пути к тестированию файловых систем на основе спецификаций

А.С. Янин, ORCID: 0009-0002-2627-8281 <slowmime@gmail.com>
В.В. Кечин, ORCID: 0009-0006-0845-2740 <valeraghst@gmail.com>
В.М. Ицыксон, ORCID: 0000-0003-0276-4517 <itsykson@yandex.ru>
В.М. Ковалевский, ORCID: 0009-0002-4501-9672 <slava0135@vk.com>

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

Аннотация. В статье описан следующий шаг развития мутационного дифференциального фаззера файловых систем DIFFuzzer, предполагающий интеграцию формальных спецификаций для улучшения его возможностей тестирования. Изначально DIFFuzzer разрабатывался на основе дифференциального подхода с интеграцией в исходный код модели POSIX, что ограничивало гибкость и масштабируемость инструмента. Предлагаемый улучшенный подход разделяет логику фаззера и модель при помощи выделения последней во внешнюю формальную спецификацию, написанную на языке LibSL. Это позволяет создавать более точные тестовые оракулы и облегчает генерацию рабочей нагрузки. В данной работе мы подробно описываем текущее состояние проекта, включая расширение LibSL и реализацию тестового оракула на основе спецификаций. Также описаны направления будущей работы и связанные с ними проблемы, такие как совершенствование формальной спецификации POSIX и развитие модуля генерации рабочей нагрузки на основе спецификаций.

Ключевые слова: модельно-ориентированное тестирование; фаззинг; формальные спецификации; файловые системы; POSIX.

Для цитирования: Янин А.С., Ицыксон В.М., Кечин В.В., Ковалевский В.М. DIFFuzzer: шаг на пути к тестированию файловых систем на основе спецификаций. Труды ИСП РАН, том 38, вып. 5, 2026 г., стр. 91–102 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(5)–6.

1. Introduction

File systems play an essential role in any modern operating system, server, or storage system. The high cost of file system failure imposes strict quality requirements on them. At the same time, file systems are extensively used. A typical user works with tens and hundreds of thousands of files, issuing a diverse range of operations, which brings the total number of them to hundreds of billions. Thus, even if a file system failure under a certain scenario is relatively unlikely, at least one user will most probably run into it at some point. To avoid this, great care must be taken to uphold high implementation quality and to discover errors before they manifest.

We previously presented DIFFuzzer [1-2], a mutation-based differential file system fuzzer [3-4]. It proved capable of finding dozens of bugs in modern file systems, confirmed by their maintainers. The main distinguishing feature of the tool is its use of a reference-based approach: two implementations are run side-by-side with the same workload and compared for behavioral discrepancies. To improve the effectiveness of the tool, workload generation relies on several assumptions about file system behavior as prescribed by the POSIX specification [5]. This solution made it possible to prototype the tool rapidly and perform early testing. However, the built-in rules also became a limitation of DIFFuzzer. Some of the rules were not elaborated in sufficient detail as the assumptions were embedded in an ad hoc manner. Additionally, although POSIX is the dominant file system interface, it is not the only one in existence: for example, Linux introduces a plethora of extensions while Windows provides a different API altogether. Furthermore, some behavioral differences between implementations are expressly allowed under POSIX as implementation-defined behavior; accordingly, our reference-based approach cannot be used to test such features. These considerations motivated our decision to refine DIFFuzzer's approach by decoupling the fuzzing logic from the rules it verifies. To that end, we employ a file system specification (in our case, a subset of POSIX describing its file system interface), described formally in the LibSL

specification language [6-7]. The formal specification can be leveraged for creating precise and comprehensive test oracles, thereby allowing us to dispense with the need for a reference implementation. In addition, it facilitates workload generation by combining automaton-based and contract-based behavior specifications [8-9].

2. Motivation

The current implementation of DIFFuzzer uses a reference-based approach [10-11] for testing. A fundamental limitation of this approach is that it defines correctness in terms of behavioral equivalence with a reference implementation instead of relying on an explicit specification. As a result, anomaly detection is constrained by observable differences between executions, rather than by explicitly stated correctness properties. This limits the expressiveness of the testing oracle, as it becomes difficult to formulate and check fine-grained or semantic correctness conditions beyond simple behavioral differences.

Furthermore, reference-based testing introduces noticeable runtime overhead due to the need to execute the reference implementation in addition to the system under test and compare their behaviors. This overhead can negatively affect scalability, particularly for systems with complex state or high execution costs.

A natural way to address these limitations is to introduce an explicit model of the system, capturing its intended behavior in a structured and analyzable form. Formulating correctness in terms of the model makes it possible to define more precise and comprehensive criteria. This shifts the focus from detecting behavioral discrepancies to checking conformance against a formal specification, as done in model-based testing (MBT) [12]. It can be seen as a generalization of property-based testing, where properties correspond to partial or lightweight models. In this sense, MBT can be naturally adapted to operate at different levels of abstraction, ranging from simple invariants to fully formalized specifications.

While DIFFuzzer implements a reference-based approach, its workload generation mechanism makes several assumptions about file system behavior. For example, if it chooses to emit a *rename* operation, it also generates a call to *creat* to avoid attempting to rename a non-existent file. These embedded assumptions essentially form a fragmented, informal model. An explicit specification could replace this ad hoc model to facilitate test generation and serve as a test oracle, either to supplement or to replace the reference-based approach.

3. Related work

In this section, we review different approaches to model-based testing (MBT) and their implementations, focusing on how they can be used for workload generation or as a test oracle, as well as miscellaneous aspects such as maintainability. To provide a structured overview of existing research in MBT, we group prior works into three categories by model type: ad hoc, reference-based, and formal.

3.1 Ad hoc models

A defining feature of ad hoc (informal) models is the absence of rigorous mathematical semantics. As a result, their interpretation may be ambiguous in general and dependent on the implementation of the testing tool. The lack of formalization limits their ability to support systematic reasoning, such as proving correctness properties or performing exhaustive verification. Instead, ad hoc models are primarily used in a pragmatic manner, where the model is effectively encoded in the structure of the tests themselves rather than defined as a standalone artifact.

For example, the Krace tool [13] uses a model (based on a specification written in Python) to generate workloads and guide the generation and mutation of system call arguments. The authors achieved remarkable results and detected several bugs in widely used file systems. However, maintaining this tool or adapting it to new projects may prove challenging, as the authors do not

provide documentation or instructions for its use or for reproducing the results. Furthermore, the model lacks an explicitly declared format and documentation, and is designed exclusively for the tool. Modifying the specification requires a thorough understanding of the tool' underlying concepts, which takes a significant time investment.

A relevant example of using an ad hoc model is demonstrated in the Commuter tool [14], which uses a model to infer commutativity rules of operations and generate test cases based on them. The authors provide a model of 18 POSIX file system and virtual memory system operations, written in Python. However, their primary intent is for end users to write their own API models for analysis with Commuter. While the authors provide documentation and example models, the latter are tightly coupled to the tool and tailored to its specific analysis workflow. In particular, the model is expressed as a general-purpose program (in Python), making it Turing-complete and shifting the responsibility for its correctness and completeness to the developer [15-17]. As a result, the model inherits the typical limitations of informal specifications: it lacks well-defined semantics, is difficult to analyze systematically, and does not provide guarantees about the properties it encodes.

Lastly, our tool DIFFuzzer [1] currently relies on an ad hoc model written in Rust for workload generation. In our view, this is a limitation of the tool: this approach is not widely used, and the model requires significant effort to develop and maintain.

Overall, ad hoc models offer a pragmatic and effective way to address specific testing tasks. Their tight integration with the implementation and reliance on general-purpose programming languages make them especially suitable for prototyping and handling complex system behaviors. However, this flexibility comes at the cost of limited abstraction, lack of formal semantics, and reduced opportunities for analysis and reuse.

3.2 Reference-based approaches

Reference-based approaches are built on the idea of using an implementation as a model of expected behavior. While we have already discussed it briefly in the previous section, we now provide a more detailed description of the approach, along with its advantages and limitations. One of the key advantages of this approach is its practicality: it eliminates the need for a specification altogether and significantly lowers the initial effort in constructing a test framework. Furthermore, reference-based testing can be highly effective at detecting inconsistencies between implementations, making it particularly useful in domains where multiple independent implementations exist, which is the case for file systems.

However, since correctness is defined in terms of equivalence with a reference implementation, the approach lacks an explicit definition of expected behavior. Consequently, the test oracle is restricted to observable differences, limiting its ability to capture deeper semantic properties or verify fine-grained correctness conditions. Additionally, the approach incurs runtime overhead due to the need to execute and compare multiple implementations, which can impact scalability for complex systems.

As mentioned earlier, DIFFuzzer's main mode of operation employs a differential approach. This allowed us to quickly develop the initial version of the tool and achieve notable results: we identified several bugs in multiple file systems, which were confirmed by the maintainers. However, during further development of the approach, we encountered limitations in its ability to define desired error conditions and generate controlled workloads.

Another illustrative example of a reference-based approach is Metis [18]. Its authors similarly demonstrated the effectiveness of their tool by detecting errors in several file systems. Nevertheless, its potential remains limited by the approach to observable differences.

Collectively, the reviewed implementations confirm the effectiveness of reference-based testing in practice while revealing limitations that motivate further development or integration of more expressive techniques.

3.3 Formal specifications

Approaches based on formal specifications rely on constructing an explicit, mathematically rigorous model of system behavior. Such models provide precise and unambiguous semantics, enabling the verification of correctness properties. In addition, they offer significant expressiveness in defining test oracles that capture not only observable behavior but also deeper semantic constraints and invariants. The formal specification is treated as a standalone artifact, which can be analyzed, reused, and extended upon independently.

Even so, these benefits come at a significant cost. Developing a formal specification is a complex and labor-intensive process that requires domain expertise. Accurately capturing the behavior of real-world systems formally often poses a significant challenge, which limits the practical applicability of this approach.

SibylFS [19] demonstrates the use of formal specifications as a mechanism for checking file system behavior. The authors provide a detailed, executable, non-deterministic model of POSIX-compliant file system behavior, which can serve as an oracle for testing and trace analysis. They also note that usability was one of their work's goals, claiming it has been achieved; however, we find this debatable. Their model is written in Lem, a mathematically rigorous language based on typed higher-order logic. While we consider it an effective solution, it is not user-friendly. Additionally, SibylFS does not support test generation.

Another representative example of using formal specifications is Ferrite [20]. The authors propose using models based on formal specifications written in Rosette [21] (a framework for building formal models and performing automated reasoning) to define the permitted states of a file system after a crash. They focus on crash behavior and do not attempt to model the full POSIX semantics (e.g., file permissions or symbolic links). Regarding the workload, referred to as litmus tests in the paper, the authors describe the structure of the tests but do not provide details on their generation; thus, we presume that the formal model does not participate in this process.

Taken together, these examples illustrate that formal specifications provide a precise and explicit way to define system behavior, which enables more structured reasoning about correctness. At the same time, they show that they can play an active role in testing: by serving as oracles or by constraining the space of admissible system states.

Importantly, formal specifications are maintained as independent artifacts, separated from the implementation and the testing logic. This separation enables the use of the same specification for different purposes, such as workload generation and behavior checking.

From this perspective, adopting an approach based on formal specifications appears as a natural extension of our work. It supplements differential testing with an explicit representation of expected behavior, thereby extending the capabilities of the tool for file system bug discovery.

4. Approach

Fuzzing effectively relies on two models: one to check execution results, and another to steer state space exploration by means of workload generation. Our approach aims to serve these two purposes with a unified model, described externally in a specification language. To achieve this, we combine automaton-based and contract-based behavior specifications of the file system's external API. In the following subsections, we shall focus on particular aspects of our approach.

4.1 Workload generation

We chose to specify the file system's behavior externally so that we treat it as a black box and interact with it only via its external API. The workload then becomes a sequence of function calls. The specification augments the workload with its semantics, allowing the pruning of nonsensical inputs — such as attempting to read from a file before opening it. Given the form of the workload, reporting

a potential bug found during fuzzing is exceedingly straightforward: one simply needs to produce a C program making the same function calls, which can be done formulaically.

4.2 External API: POSIX

While the approach outlined here is not tied to a particular choice of the external API, for our work we have selected to target the POSIX file system API. As the dominant file system interface, it encompasses a wide variety of implementations. A textual specification allows us to capture its requirements formally without relying on a particular implementation, thus ensuring universality. Additional file system features are often introduced as extensions of the POSIX API, which means that our formal specification can likewise be extended to support these features if necessary. For clarity, we will henceforth restrict our discussion to the POSIX file system interface.

4.3 System behavior specification: automata

Automata define a state space along with transitions between states. In POSIX, changes to the file system state can only be initiated with calls to file system functions, which lets us associate state transitions with function calls. Automata, in turn, correspond to file system objects, such as files and directories.

Automata states are explicitly named and enumerated. A potential transition from one state to another is fully determined by the choice of a function; however, for greater flexibility and precision, we extend automata with attributes. Contracts may access these to refine state transitions: while they cannot affect the source and target states of the potential transition, they can cancel it altogether, leaving the state unchanged.

The described semantics reflect the behavior of POSIX file operations, which generally do not modify the file system unless they succeed. If we then overlook cancellation, transitions between explicit states become effectively deterministic. This is the key idea that enables directed state space exploration. In a given source state, we choose an available target state to transition to. Each comes with a selection of functions, possibly many. However, once we commit to a function, if we succeed in changing the state at all, we are ensured to land in that target state and no other.

4.4 Function behavior specification: contracts

Whereas the automaton-based part of the behavior specification is primarily intended to facilitate workload generation, contracts allow fine-grained specification of a particular function's expected behavior in terms of its preconditions and postconditions. Preconditions describe requirements that have to be satisfied before the call. If they do not apply to the current state, no guarantees can be made about the state of the file system following the call. This renders further execution of such workload effectively useless. Therefore, we generate the workload one operation at a time; if the preconditions fail, the function call is not emitted in the first place. Postconditions, on the other hand, are checked after the call has been performed. Thus, their violation indicates an erroneous situation — in other words, a potential bug.

Preconditions and postconditions may refer to extended attributes of automata as well as call a number of intrinsic functions that provide access to the state of the file system under test. Preconditions are evaluated in the previous state and postconditions normally in the next state. There is, however, a way to mark a postcondition subexpression for evaluation in the state preceding the call. This accordingly splits postcondition checking into two phases (see Fig. 1). The first phase computes the values of the marked subexpressions, whereas the second phase uses them to evaluate the final truth value of the postcondition and derive the next state (including values of extended attributes).

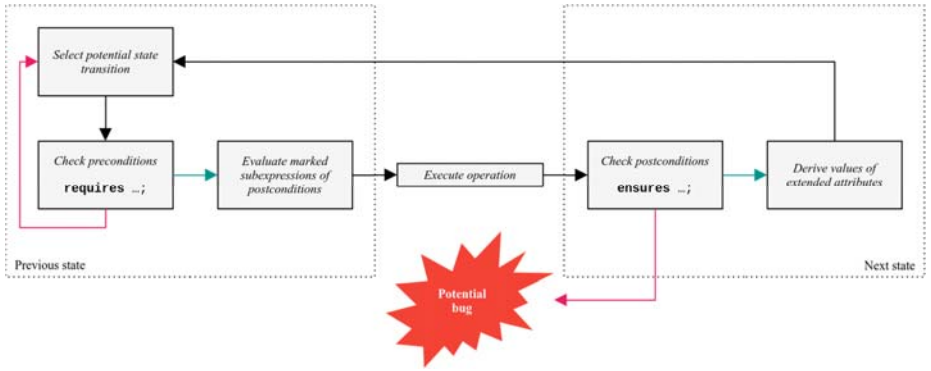

Fig. 1. File operation contracts.

4.5 Modeling nondeterminism

Implicit in the proposed treatment of postconditions is the need for online operation, in the sense of interleaving operation execution and postcondition verification one step at a time. This approach primarily aims to avoid a potential combinatorial explosion introduced by different kinds of nondeterminism inherent to file system operations while being easier to implement.

The first kind of nondeterminism arises from the fact that virtually no POSIX file operation can be assured to succeed. We account for this by canceling the potential state transition in a postcondition if we detect failure, describing, if possible, sufficient conditions to guarantee it. Furthermore, since an operation may fail due to several factors, the value of *errno* essentially depends on the order of checks in the implementation, left unspecified by POSIX. Therefore, we enumerate which values of *errno* may arise from an execution of the particular operation and attempt to state necessary conditions for each.

The second kind of nondeterminism relates to successful outcomes that allow a range of possible results. As a benign example, consider *read* or *write*, which may process any number of bytes up to the requested amount; however, modeling this in a formal specification poses little difficulty due to our online approach. More troublesome are *readdir* and *rewinddir*, which interact in a loosely-specified manner with interspersed modifications of the directory's contents. An approach employed by SibylFS models them by keeping track of open directory streams and maintaining the sets of what the authors refer to as "may" and "must" entries for each stream after any operation that affects them. We currently do not include these operations in our specification. A similar situation occurs with possibly-deferred actions, which in POSIX include updating access timestamps and reclaiming serial numbers of removed files. We likewise omit these features from our model.

The third kind of nondeterminism encompasses implementation-defined behavior. Our current specification handles it in the most general way by assuming indeterminate choice. However, the scope of implementation-defined behavior may be open-ended. Such is the case with the access control mechanism: POSIX allows the implementation to introduce additional and alternate mechanisms. This means that the most general treatment cannot faithfully model POSIX access control; however, it remains possible with a specialized specification.

4.6 Specification language: LibSL

One of our goals is to decouple the specification from the implementation of DIFFuzzer to allow changing the former independently. This way, extending the specification for specific needs would not require knowledge of the internal workings of our testing system. To represent the specification in text, we have settled on LibSL [6-7]. The language is designed for describing the behavior of

external functions and supports mixed automaton-contract specifications natively, which meets our needs particularly well.

5. Current state

As outlined earlier, our goal is to incorporate the use of a unified specification in DIFFuzzer for both workload generation as well as checking the results of test execution. However, since DIFFuzzer already includes a mutation-based workload generation module, we have decided to reuse it as is while concentrating our efforts on implementing a specification-based test oracle. The hybrid scheme allows us to validate our approach incrementally. Currently, we have adapted LibSL for our purposes and created a preliminary version of the POSIX file interface specification as well as the test oracle.

Originally, the LibSL ecosystem was only available for JVM-based languages and could not be leveraged for DIFFuzzer easily given that the latter was developed in Rust. Accordingly, we have created a Rust implementation of LibSL language support libraries while introducing several changes to the language. Our guiding concerns were reducing inconsistencies and improving user experience while allowing for sufficient expressiveness. The most prominent changes we made were a revision of the contract syntax and the addition of the set data structure. These are shown in a function's contract specification given in Fig. 2.

```
fun read(fdNum: Int, buf: *unsigned8, len: size_t): ssize_t {
   requires fs.getFd(fdNum) !is null;
   ensures result >= -1;
   ensures {
       val fd = fs.getFd(fdNum) as FileDescription;
       if (len != 0) {
           if (fd.mode' !in {OpenMode.R, OpenMode.RW}) {
               result == -1;
           }
       }
       if (result == -1) {
           errno in {
               Errno.EBADF, Errno.EINTR, Errno.EIO,
               Errno.ENOBUFS, Errno.ENOMEM
           };
           EBADF: if (errno == Errno.EBADF) {
               fd.mode' !in {OpenMode.R, OpenMode.RW};
           }
       } else {
           result <= len;
           if (fd.file.*.size' - fd.offset' > 0) {
               result <= fd.file.*.size' - fd.offset';
           } else {
               result == 0;
           }
           fd.offset == fd.offset' + result;
       }
   }
}
```

Fig. 2. The behavior specification of read.

The figure demonstrates an abridged version of the behavior specification of *read*. We have compiled a contract-based specification of a subset of the POSIX file interface comprising 20 commonly used functions: *read*, *pread*, *write*, *pwrite*, *close*, *lseek*, *mkdir*, *open*, *link*, *unlink*, *chown*, *lchown*, *chmod*, *symlink*, *rename*, *truncate*, *ftruncate*, *fsync*, *fdatasync*, *sync*. Since we are repurposing the workload generation module, the specification presently omits automaton definitions and cancellation.

We are currently working on the execution checker. Fig. 3 depicts its initialization. The specification is initially uploaded to a test runner VM, which loads the file into memory while parsing it and performing semantic analysis. The checker also supplies definitions of built-in types and intrinsic operations that can be imported into the user-provided specification. The parsed specification is then searched for file operation definitions. Since DIFFuzzer works in terms of its own file operation model, we check that these definitions conform to function signatures expected by DIFFuzzer. This completes the initialization.

The execution checker is essentially a LibSL interpreter. It works in tandem with the differential approach and implements the algorithm outlined in Fig. 1 with a few differences owing to our reusing the mutation-based workload generator. Since the entire test case is generated at once as opposed to step-by-step, potential state transition selection is predetermined. We still verify preconditions, however: if they fail, the checker assumes it must be an internal error, since it betrays a mismatch in expectations between the workload generator and the specification. The model-based checker then skips the rest of the workload, which will be handled solely by the differential checker.

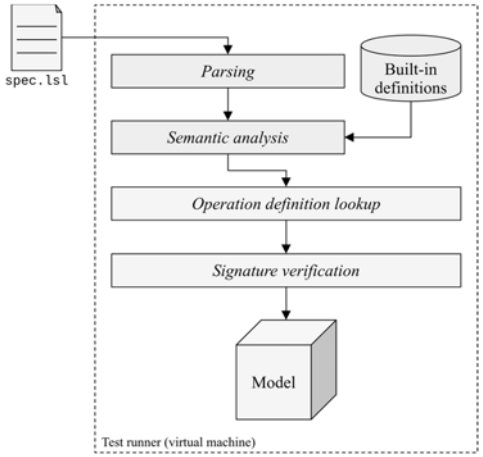

Fig. 3. Execution checker initialization.

We have conducted preliminary experiments by testing the newly extended DIFFuzzer on Ext4, Btrfs, and LittleFS. By running traces corresponding to bugs previously discovered by DIFFuzzer in LittleFS, we were able to confirm the specification-based checker's ability to discover bugs in real file systems. We also ran the fuzzer for 24 hours continuously on these file systems but were unable to discover new bugs. The overhead introduced by the specification-based execution checker varies greatly depending on the actual contents of the workload, but generally the fuzzer exhibits a 2× slowdown compared to the previous version: in particular, fuzzing Ext4 with Btrfs processes 1.3 traces per second with the specification and 2.5 without.

The fact that the specification-based checker can indeed find bugs, even if they could already be discovered by comparing execution traces, opens the potential to rely entirely on the specification-based oracle for fuzzing. Since that does away with the need to run each workload on two file systems, the resulting performance gain can make up for the overhead. On the other hand, combining both types of oracles greatly reduces the barrier to adoption of formal specifications, since it allows using partial specifications that only focus on the essential functionality (as determined by the user).

6. Future work

We plan to continue work on the specification-based execution checker to improve its performance and user experience. Following that, next steps in our research involve improving the precision and

completeness of our POSIX specification, and implementing a specification-based workload generation module for DIFFuzzer.

Although our POSIX specification covers functions that we would expect to find in a typical program, it is still incomplete. For example, as alluded to previously, we do not support directory stream operations: *opendir*, *readdir*, *rewinddir*. The *at*-family of functions (*openat*, *mkdirat*, *renameat*, etc.) is also excluded from the specification: LibSL does not currently provide a way to parametrize predicates so they could be reused across several functions, which means that adding support for an *at*-counterpart of an operation would essentially duplicate its specification. We aim to describe these and other functions formally in the future. Similarly, we intend to improve the precision of our specification by including more file system features, most notably modeling file contents.

While we are currently pursuing a hybrid approach combining the test oracle with specification-independent, mutation-based workload generation, we eventually aim to substitute the latter with a specification-based counterpart. Achieving this poses several challenges, some of which we list below:

1) Dynamic spawning of automata may eventually exhaust machine resources. The implementation needs to manage automaton lifetimes to prevent this.
2) Some operations change the state of multiple automata at once (such as renaming a file, which affects the file itself as well as the two involved directories). Our approach so far only allows for a single automaton to change its state as a result of an operation.
3) Furthermore, the complexity of POSIX means that extracting explicit states may prove difficult. For example, one may want to declare separate file descriptor states for each mode of access (read-only, write-only, read-write, etc.) it was opened with so that *write* calls are not attempted for files opened for reading only. However, this requires that the target state is determined by not only the source state and the called function (here, *open*) but the call's arguments as well, which demands a more sophisticated treatment of state transitions.

Developing specification-based workload generation requires solving all these challenges.

7. Conclusion

We have established a new direction for DIFFuzzer's evolution, incorporating a specification-based approach to file system fuzzing. Our proposed method endows the specification with two levels of defining the expected file system behavior: automata for coarse-grained, whole-system behavior and contracts for fine-grained, function-specific behavior. This enables the application of a unified specification in workload generation as well as for checking test execution results. So far we have developed a LibSL language support library in Rust for its use in DIFFuzzer and implemented an initial version of the specification-based test oracle module. Furthermore, we have developed a preliminary contract-based formal specification of POSIX's file system interface. Although the addition of the specification-based execution checker did not yield new bugs in Ext4, Btrfs, or LittleFS, it was able to confirm the results of the differential oracle. Future work includes the enhancement of the specification-based oracle's performance, the improvement of the precision and completeness of our POSIX formalization, and the development of a specification-based workload generation module.

References

[1]. Ковалевский В.М., Кечин В.В., Ицыксон В.М. DIFFuzzer: обнаружение ошибок файловых систем с помощью дифференциального фаззинга серого ящика. Труды ИСП РАН, 2025, том 37, вып. 4, часть 2, стр. 31-46 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(4)–17. / 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, 2025, vol. 37, issue 4 (part 2), pp. 31-46. DOI: 10.15514/ISPRAS-2025-37(4)-17.

[2]. Ковалевский В.М., Кечин В.В., Янин А.С., Ицыксон В.М. Легковесная проверка согласованности файловой системы после сбоев с помощью дифференциального фаззинга. Труды ИСП РАН, 2026, том 38, вып. 1, стр. 79-92 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(1)–7. / Kovalevsky V. M., Kechin V. V., Yanin A. S., Itsykson V. M. Lightweight file system crash-consistency checking with differential fuzzing. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 1, pp. 79-92. DOI: 10.15514/ISPRAS-2026-38(1)-7.

[3]. Miller B.P., Fredriksen L., So B. An empirical study of the reliability of UNIX utilities. Commun. ACM, 1990, vol. 33, no. 12, pp. 32–44. DOI: 10.1145/96267.96279.

[4]. Manes V.J. M., Han H., Han C., Cha S.K., Egele M., Schwartz E.J., Woo M. The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering, 2021, vol. 47, no. 11, pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[5]. 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), 2024, 4107 p.

[6]. Itsykson V.M. Partial Specifications of Libraries: Applications in Software Engineering. Communications in Computer and Information Science, 2021, vol. 1288, pp. 3-25. DOI: 10.1007/978-3-030-71472-7_1.

[7]. Itsykson V.M. LibSL: Language for Specification of Software Libraries. PROGRAMMNAYA INGENERIA, 2018, vol. 9, pp. 209-220. DOI: 10.17587/prin.9.209-220.

[8]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTesK Test Suite Architecture, 2002, pp. 77-88. DOI: 10.1007/3-540-45614-7_5.

[9]. Гриневич А.И., Кулямин В.В., Марковцев Д.А., Петренко А.К., Рубанов В.В., Хорошилов А.В. Использование формальных методов для обеспечения соблюдения программных стандартов. Труды ИСП РАН, 2006, том 10. / Grinevich A.I., Kulyamin V.V., Markovtsev D.A., Petrenko A.K., Rubanov V.V., Khoroshilov A.V. Using formal methods to ensure compliance with software standards. Trudy ISP RAN/Proc. ISP RAS, 2006, vol. 27 (in Russian).

[10]. Groce A., Holzmann G., Joshi R. Randomized Differential Testing as a Prelude to Formal Verification, 2007, pp. 621-631. DOI: 10.1109/ICSE.2007.68.

[11]. McKeeman W.M. Differential Testing for Software. Digit. Tech. J, 1998, vol. 10, pp. 100-107.

[12]. Petrenko A. Model-based testing. In Proceedings of the 27th international conference on Software engineering (ICSE '05). Association for Computing Machinery, 2005, pp. 722-723. DOI: 10.1145/1062455.1062636.

[13]. Xu M. et al. Krace: Data race fuzzing for kernel file systems. IEEE Symposium on Security and Privacy (SP). IEEE, 2020, pp. 1643-1660. DOI: 10.1109/SP40000.2020.00078.

[14]. Clements A.T. et al. The scalable commutativity rule: Designing scalable software for multicore processors. ACM Transactions on Computer Systems (TOCS). Association for Computing Machinery, 2015, vol. 32. issue 4, pp. 1-47. DOI: 10.1145/2699681.

[15]. Fuchs N. Specifications are (preferably) executable. Software Engineering Journal, 1992, pp. 323-334.

[16]. Lamport L. Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Boston, MA, United States, 2002, 364 p.

[17]. Jackson D. Software Abstractions - Logic, Language, and Analysis. The MIT Press, Cambridge, MA, United States, 2006.

[18]. Liu Y. et al. Metis: File system model checking via versatile input and state exploration. 22nd USENIX Conference on File and Storage Technologies (FAST 24). USENIX Association, 2024, pp. 123-140.

[19]. Ridge T. et al. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. Proceedings of the 25th Symposium on Operating Systems Principles. Association for Computing Machinery, 2015, pp. 38-53. DOI: 10.1145/2815400.2815411.

[20]. Bornholt J. et al. Specifying and checking file system crash-consistency models. Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 2016, pp. 83-98. DOI: 10.1145/2872362.2872406.

[21]. Torlak E., Bodik R. Growing solver-aided languages with rosette. Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software. Association for Computing Machinery, 2013, pp. 135-152. DOI: 10.1145/2509578.2509586.

Информация об авторах / Information about authors

Александр Сергеевич ЯНИН – студент магистратуры и инженер Института прикладных компьютерных наук университета ИТМО. Сфера научных интересов: динамический и статический анализ программ, фаззинг.

Alexander Sergeevich YANIN – graduate master student and engineer at 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.

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

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

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

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