LLM-assisted Industrial-Scale Differential Testing of Package Incompatibilities in Linux Distributions

Yuhao Yang[1*], Chijin Zhou[2, 3*], Runzhe Wang[4], Weibo Zhang[1], Yuheng Shen[3], Xiaohai Shi[4]
Tao Ma[4], Chang Gao[4], Zhe Wang[4], Ying Fu[3], Heyuan Shi[1†]
[1]Central South University, Changsha, China
[2]East China Normal University, Shanghai, China
[3]Tsinghua University, Beijing, China
[4]Alibaba Co., Ltd, Beijing, China

***Abstract*—An open source Linux distribution often undergoes version upgrades and migrations, which is prone to incompatibility issues especially when it comes to large-scale software changes. Although differential testing has been widely used in software testing, it is still challenging to apply it for detecting such incompatibilities in the context of industrial settings. In this paper, we report our experience in leveraging LLMs to address the challenges faced by the Linux distribution community. Specifically, we develop an LLM-based differential testing method called Versify to assist maintainers of Linux distributions in locating incompatibilities during version upgrades and migrations. Its trial operation period within the Linux distribution community shows that it uncovered 8,489 instances of differing behavior, of which 644 were prioritized for attention by developers. After deduplication and filtering, 39 unique compatibility reports were identified. Feedback from Linux distributions developers indicates that our reports have provided valuable recommendations for package selection in future OS releases.**

***Index Terms*—Differential Testing, Software Test, Large Language Model, Linux Distribution**

I. Introduction

An open source Linux distribution often undergoes version upgrades and migrations, which is prone to incompatibility issues especially when it comes to large-scale software changes. For example, the transition from CentOS 7 to CentOS 8 involved significant changes in the underlying software stack, and thus the software packages installed in the system may not functionally work as expected after the upgrade. The maintainers of Linux distributions are expected to ensure the compatibility of thousands of software packages in operating systems, but this requires substantial labor due to the complexity of software dependencies and the diversity of software packages.

Although differential testing has been widely used in software testing [1], [2], it is still challenging to apply it for large-scale software packages across Linux distributions, especially in the context of industrial settings. The gap is mainly located in the following aspects:

- **Scalability.** Different software packages have different input formats and different testing requirements. The diversity makes it difficult to customize a unified test cases generation strategy for *all* software packages.
- **Effectiveness.** Maintainers of Linux distributions tend to care more about the software packages that are widely used or frequently updated. Therefore, test cases generation should be more effective for these packages in order to verify their compatibility.
- **Environment.** The testing of software packages involve multiple operating systems, different software versions, and complex dependencies. Once the testing environment is inconsistent, the same test case is likely to output different results. Therefore, it is necessary to ensure the consistency and correctness of the environment.
- **Misreports.** The inconsistencies of output are not necessarily compatibility issues. The ones that have no severe impact on the system should be classified as misreports, which need to be filtered to reduce the burden on maintainers.
- **Continuous Testing.** Given that the software packages are continuously updated, it requires a continuous process for analyzing and tracking test results over time. Therefore, it is non-trivial to manage the testing environment as well as configurations in a continuous manner.

In this paper, we conduct industry experience in leveraging large language models (LLMs) to address challenges in the Linux distributions community, a well-known open-source Linux distribution used by Alibaba Cloud. Specifically, we develop an LLM-based differential testing method called Versify[1] to assist maintainers of Linux distributions in locating incompatibilities during version upgrades and migrations. To demonstrate the effectiveness of Versify, this paper presents deployment results from its first month of use in AnolisOS 8 and AnolisOS 23. During this period, we tested all system software in AnolisOS, generating approximately 58,000 test cases. This process uncovered 8,489 instances of differing behavior, of which 644 were prioritized for attention by developers. After deduplication, 39 unique compatibility reports were identified. Feedback from AnolisOS developers indicates that our reports have provided valuable recommendations for package selection in future OS releases.

The main contributions are summarized as follows:

*Both authors contributed equally to this research.
†Heyuan Shi is the corresponding author.

[1]The LLM used by Versify is a general-purpose model. This paper focuses on how a general-purpose LLM can be integrated into differential testing to identify software packages' incompatibilities industrial settings.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00293

1) We identify the challenges in deploying automated differential testing on open source Linux distributions in the context of industrial settings.
2) We address these challenges with corresponding solutions, including generating test cases using LLM, Optimizing prompt and use text assistance, creating package-level testing environment using containers, constructing reports priority and filter misreports by rules, and developing a continuous differential testing framework.
3) In the real-world industry practice, Versify effectively test all system software in AnolisOS 8 and 23, finally identified 39 compatibility reports within a month.

II. Background and Related Work

A. Differential Testing

Differential testing is an automated testing process that focuses on identifying discrepancies and inconsistencies between different versions of a software system, which includes input generation, test cases execution, differential analysis, and continuous differential testing of the overall framework. For Linux distributions, differential testing is a collaborative effort between compatibility quality assurance, security management personnel, and internal development personnel. Development teams should focus on package compatibility between versions to help ensure safety during system upgrades or migrations.

The general differential testing process involves three main steps: 1): Input Generation: Test cases are created using automated generators or manual input strategies, recorded, and integrated into the framework. 2): Test Cases Execution: Test cases are run in isolated environments with results based on outputs, errors, and return codes. 3): Differential Analysis: Outputs are compared to identify differences, and high-risk reports are shared with developers for resolution.

Recent studies have explored integrating LLMs with differential testing. For example, Mokav [3] leverages LLMs to generate inputs that expose functional differences, LLMeDiff [4] applies multiple LLMs to detect inconsistencies in a medical rule system, and Liu et al. [5] proposed AID to identify tricky bugs through LLM-guided test generation. Braberman et al. [6] further investigated using LLMs to highlight discrepancies across implementations. In the software domain, differential testing has been widely employed to identify semantic bugs in various systems, including SSL/TLS implementations [7]–[10], C compilers [11], JVM implementations [12], web application firewalls [13], security policies for APIs [14], antivirus software [15], and file systems [16], among others. It has also been applied to the automated generation of inputs for different protocol implementations [17]. While differential testing does not guarantee equivalence of outputs, it significantly enhances confidence in the correctness of software, similar to conventional software testing practices.

B. LLM-based Test Cases Generation

LLM have emerged as a promising solution for automated test cases generation in software engineering, addressing the challenges of ensuring high coverage and efficiency in complex systems [18]. LLM-based test cases generation shares similarities with fuzz testing, as both approaches generate unpredictable outputs. However, while fuzz testing typically relies on random or mutated inputs, LLM-based test cases generation leverages natural language to guide the creation of structured and semantically meaningful test cases [19]–[26]. Once integrated with LLM, testers are no longer required to manually write test cases or develop and debug cumbersome fuzz testing tools. In addition, the automation of test cases generation accelerates the testing process, shortening the software development cycle and enabling more efficient allocation of testing resources by reducing manual intervention and allowing personnel to focus on test management and quality assurance.

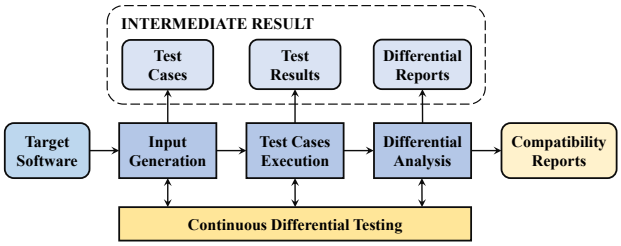


Fig. 1: Typically differential testing workflow.

After combining LLM, many interesting research of testing have been emerged. TESTPILOT [18] leverages LLM to automatically generate effective JavaScript unit tests, achieving high code coverage without additional training or manual effort. HITS [27] enhances LLM-based unit test generation for complex Java methods by decomposing them into slices. Mathews and Nagappan [28] demonstrate that providing LLMs with test cases alongside problem statements enhances the success rate of solving programming challenges, they investigate the incorporation of Test-Driven Development principles into LLM-based code generation. Pan et al. [29] present a generic pipeline that uses static analysis to guide Large Language Models in generating compilable and high-coverage unit tests for multiple programming languages. Similarly, LLM4Fin [30] is a fully automated approach that utilizes fine-tuned LLM to generate high-coverage test cases from natural language business rules for FinTech software acceptance testing. SymPrompt [31] introduces a code-aware prompting strategy for LLMs that decomposes test suite generation into multi-stage prompts based on execution paths, while ChatUniTest [32] presents an LLM-based framework for automated unit test generation that uses an adaptive focal context mechanism for precise prompting. Both approaches enhance coverage and improve test generation accuracy. Our work focuses on leveraging LLMs to generate high-quality test cases for packages in Linux, specifically for differential testing, aiming to alleviate the challenges developers face in creating robust and comprehensive test suites.

III. Procedures and Targets

Versify is an LLM-based differential testing method for Linux distributions that offers features such as batch gen-

eration of test cases, scripted testing processes, misreports filtering, and a continuous differential testing pipeline for submitting compatibility reports to the open-source Linux distribution community. In particular, Versify is optimized for AnolisOS, and is designed to assist maintainers in locating incompatibilities during version upgrades and migrations. In this section, we introduce the procedures of deploying Versify and the target Linux distributions.

A. Versify's Procedures

Versify integrates large language models (LLMs) into the differential testing process. As illustrated in Figure 1, its workflow consists of three key steps: input generation, test case execution, and differential analysis. This integration enables continuous differential testing for large-scale software packages.

Input Generation: Versify uses an automated software tracker to collect information about software packages from various operating system repositories, as well as other sources, including any relevant documents that provide supplementary details about the software packages. Then, the software list and the corresponding supplementary texts are combined to design the prompt, with the information chunked and inputted into the LLM in an organized manner. As a result, this step generates a set of test cases used in the next execution phase.

Test Cases Execution: In this step, each test case is executed in separated package-level test environment. Obviously, execution of test cases needs to maintain the independence of the testing environment, otherwise large-scale software would cause unpredictable dependency conflicts. Therefore, we usually need to manually construct a relatively separated testing environment. And then the log of execution is generated by recording the standard output, standard error output, and return code of each test case, which forms the difference report used as input for the subsequent differential analysis phase.

Differential Analysis: In this step, the logs collected from the two operating systems are compared. Any differing outputs are flagged as potential compatibility issues. However, there is a semantic gap between differences and actual compatibility problems, as output discrepancies do not necessarily indicate genuine vulnerabilities or functional incompatibilities. Therefore stronger causal validation is required to distinguish benign discrepancies from genuine compatibility issues. This step can result in a lot of low-grade differences that can be regarded as misreports which should be filtered, while using appropriate misreports filtering methods can promote the automation of differential testing. There are many methods for analyzing differences, for example, difference in return codes is assigned higher severity ratings and should receive more attention. This approach aims to minimize misreports while identifying critical differences, thereby reducing the resource burden on community contributors tasked with validating the submitted difference reports.

Continuous Differential Testing: In addition to the above procedures, Versify is integrated into the continuous development process, allowing users to conduct continuous testing. It provides a user-friendly report format to assist maintainers in reviewing and confirming reports.

TABLE I: Summary of target Linux distributions and software packages.

Distribution	#Packages Number	Shared Packages	Software Scope
AnolisOS 8	7792	3882 software packages with 228 popular ones	Basic system tools Develop tools Network service, etc.
AnolisOS 23	13604		Basic system tools Develop tools General computing Intelligent computing Cloud native, etc.

B. Linux Distribution and Software Packages

This paper introduces the deployment experience of Versify in two active AnolisOS versions, AnolisOS 8 and AnolisOS 23. The overview of these version is summarized in Table I. AnolisOS 8 supports software packages for x86_64, aarch64 and LoongArch architectures, with a total of 7792 software packages covering basic system tools, development tools, and network services, etc. AnolisOS 23 [33] is the next-generation operating system which is designed for hardware-software collaborative. Compared to AnolisOS 8, the software system has a more extensive application which includes 13604 software packages involving additional scope such as general computing, intelligent computing, cloud native, etc. Both versions of the Linux distribution are selected because not only they ensure broad compatibility with the mainstream software ecosystem and supporting diverse computer architectures, but also the active open-source community facilitates responsive feedback. By considering community's feedback, we can iteratively improve Versify.

We conducted static analysis of both software system to identify our target software that contains 3882 software packages shared between the two versions. Additionally, during the interaction with OpenAnolis community, community maintainers highlighted a list of software packages including 228 popular software that should receive more attention [34]. These target packages are selected based on their popularity, usage frequency, and the criticality of their functionalities in the system. As a result, these software packages are critical to the system, and vulnerability detection is crucial, therefore, more comprehensive testing methods are considered for them, particularly in the input generation phase, as they need more efficient test cases.

IV. Challenges and Solutions

As shown in Table II, we summarize the typical challenges and solutions in the typical steps of developing and applying LLM-based differential testing on software system of Linux Distributions. The details are introduced as follows.

A. Input Generation

Challenge 1: Difficulty in generating valid test cases for various software packages. Generating test cases is often the

TABLE II: Challenge and Solution Overall.

Step	Challenge	Solution
Input generation	Difficulty in generating valid test cases for various software packages	Generate test cases using LLM
	Inefficient test cases generation for popular software	Optimize prompt and use text assistance
Test Cases Execution	Complex and interdependent testing environment	Create package-level testing environment using containers
Differential Analysis	Massive differential testing reports and misreports	Construct reports priority and filter misreports by rules
Continuous Differential Testing	Complexity of continuous differential testing	Develop a continuous differential testing framework

Resource-Unrelated

```
# ls
ls
ls -l
ls -a
ls -lh
ls -lt
ls -ltr
ls --color=auto
ls -R
ls -S
ls -i
ls -d */
ls -1
ls --time=atime
ls --group-directories-first
ls --hide=*.txt
```

Resource-Dependent

```
# ls
touch file1 file2; ls
mkdir test_dir; ls test_dir
mkdir -p dir1/dir2; ls -R dir1
touch hidden_file; mv hidden_file \
    .hidden_file; ls -a
touch fileA fileB fileC; ls -l
touch fileX; chmod 644 fileX; ls \
    -l fileX
touch fileY; ln -s fileY symlinkY; ls -l
touch fileZ; ls -i fileZ
touch file1 file2 file3; ls -t
touch fileA fileB fileC; ls -r
touch testfile; echo "Sample text" \
    > testfile; ls -lh
mkdir dir1; mkdir dir2; mv \
    testfile dir1; ls dir1 dir2
```

Fig. 2: Comparison of two sets of test cases for 'ls' software package. Right is resource-dependent which satisfy the environment dependency requirements for test cases execution while left is resource-unrelated, but the response of left occupies fewer contextual windows, allowing it to handle more software packages for test cases generation at once.

first step in the testing process, and its quality plays a crucial role in determining the efficiency of the overall testing. When the testing target involves *all* software packages within a Linux distribution, massive software requires a large-scale test cases generation to achieve comprehensive coverage. Furthermore, the dependencies of test cases, such as system configurations or external libraries, directly affect the accuracy of the testing process, thereby highlighting the importance of dependency in the design of effective test cases.

As shown in Figure 2, test cases must not only ensure the correctness of rules and the validity of semantics, but also guarantee the proper generation of its environmental dependencies. These factors collectively make it challenging for differential testing on massive software packages. Current methods rely on manually written test cases or fuzz testing tools to randomly generate test cases, despite techniques such as coverage-guided methods [35], [36], these approaches still demand significant resources and fail to address the challenges of large-scale test cases generation. As a result, large-scale test cases generation remains a challenge, which limits the performance of software differential testing.

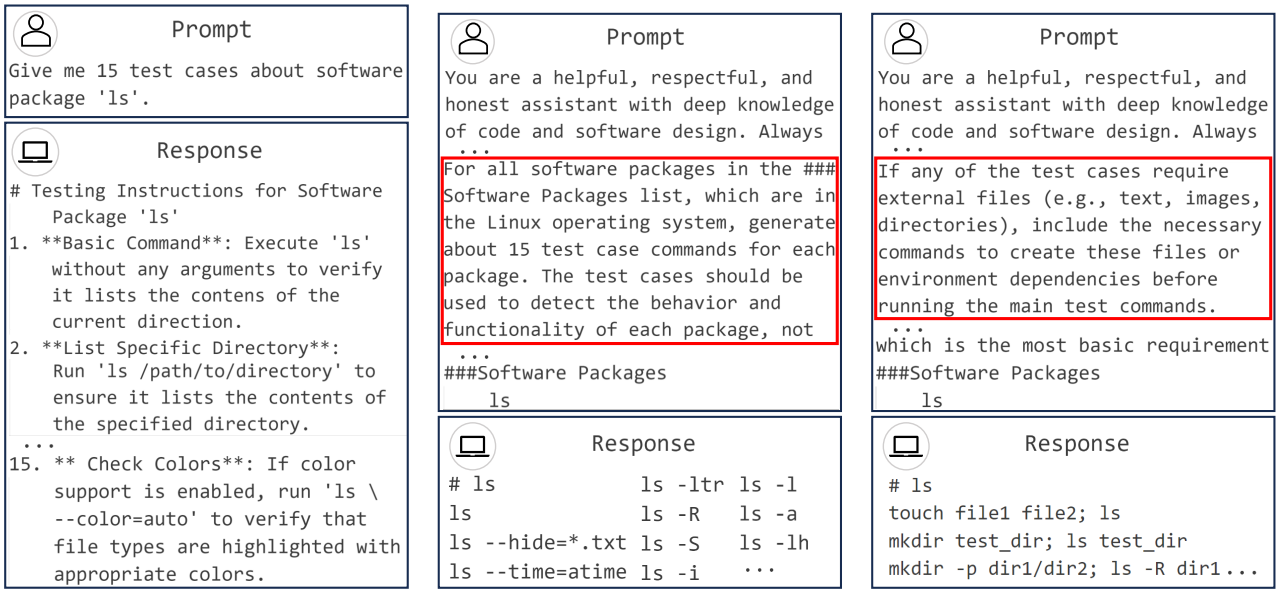
Solution 1: Generate test cases using LLM. To address this challenge, we leverage LLM to generate large-scale test cases for differential testing. First, we gather the default software packages lists from both systems and select the packages suitable for differential testing. Then, we design prompts through prompt engineering to optimize the output of the LLM. We analyzed specific tasks and summarized the rules that prompts should follow based on the Prompt Engineering [37], as shown in Table III. Specifically, we integrate the requirements with the software package names using markdown syntax, and iteratively refine the prompt with the LLM while monitoring the corresponding outputs. Our goal is to ensure that the final prompt enables the LLM to generate a set of test cases that are compliant with the input rules and effectively explore packages' functionalities.

In our experiments, we found that if a large number of software package names are input into the LLM at once, the attention mechanism causes a blurred understanding of the requirements and finally leads to meaningless outputs. Additionally, the length limitation of the context window also makes this approach invalid, because a large number of software packages must lead to excessively long prompt words. As shown in Figure 3, we found that the prompt words designed by us have better effects for LLM. Therefore, after determining the prompt template for test cases generation in differential testing of software, we divide the software packages into different groups and embed them into the template separately. This approach helps the LLM focus more on the key aspects of the prompt. Finally, we input the final prompt into the LLM to generate test cases, which are then collected for subsequent test cases execution phases. This approach also supports automation and aligns with industry standards.

Challenge 2: Inefficient test cases generation for popular software. Maintainers of Linux distributions tend to care more about the packages that are widely used or frequently updated. We obtained a list of popular software for AnolisOS, which poses a challenge for community workers in terms of software selection requirements for the release of new versions of AnolisOS. The frequent updates, high attention, and high usage frequency of popular software require a more detailed and comprehensive set of test cases. Although existing fuzz testing tools, such as AFL [38] and syzkaller [39], can generate a large number of test cases, the effective parts are often too few to address key compatibility and behavioral issues. These tools struggle to meet the testing needs of popular software in terms of semantics, rules, and coverage. While large language models can address some of these limitations, the attention mechanism and context window constraints inherent in the transformer architecture still limit the ability of LLMs to generate test cases for popular software. As a result, there is a

TABLE III: Prompt designed for our task with assistance of Prompt Engineering.

Prompt Design Rules	Description
Clarity and Context Specification	Clearly define the assistant's role, task goals, and constraints to ensure precise outputs.
Step-by-Step Guidance (Chain of Thought)	Outline a logical process, ensuring each step is clear and sequential.
Few-Shot Learning with Examples	Demonstrates the expected output format and style by including specific examples.
Constraints and Limitations	Specific constraints (avoiding rpm, yum, dnf) and formatting requirements (use of ; , structured output).
Focus and Precision	Emphasize the importance of focusing solely on the task



(a) No design for prompt, just a simple request to LLM, receive a simple response

(b) Prompt designed for Challenge 1, receive batches of test cases that fit in input syntax.

(c) Prompt designed for Challenge 2, the response to which is more resource-dependent

Fig. 3: The behavior of LLMs varies significantly depending on the prompt used. Among these, (a) fails to meet the requirements of our task, as it does not fit in the syntactic rules for input, thereby adding the additional burden of modifying test cases. In contrast, options (b) and (c) correspond to the designs of Solution 1 and Solution 2 respectively.

need to improve the test cases generation methods specifically for popular software.

Solution 2: Optimize prompt and use text assistance. We continue to use the LLM to generate test cases, designing prompts suitable for popular software from the perspective of the attention mechanism and context window. First, we found that popular software tend to be limited in number, typically consisting of core system utilities or widely-used applications that are essential across many users or environments. For example, the list of popular software packages in AnolisOS contains only a little over 200 software packages. Therefore, each prompt can focus on one or two software packages, while requiring the LLM to generate more test cases. This approach allows for the LLM to concentrate its resources on generating comprehensive test cases for those packages within the limitations of the context window, which not only improves the quality but also increases the precision of the generated test cases. Then, due to the reduced number of software in the prompt, we can retrieve their corresponding documentation, such as help texts or official manuals, and integrate them into the prompt. This step can be done incorporating these help documents by manually. Additionally, as shown in Figure 3c, we have designed more detailed prompts for popular software, focusing on resource-dependent commands and syntactic rules, and require that the generated results be more accurate. From the perspective of the attention mechanism, the reduction in the number of software packages increases the proportion of documentation and requirements within the prompt, allowing the LLM to focus more on what needs to be generated, resulting in more effective test cases.

As shown in Figure 3, compared to Solution 1, in terms of LLM output, a significant characteristic of this step is the reduction in the number of target software packages that LLM must handle. This allows LLM to generate a greater number of better test cases per software package, resulting in higher test cases coverage and effectiveness, better meeting the needs of popular software packages.

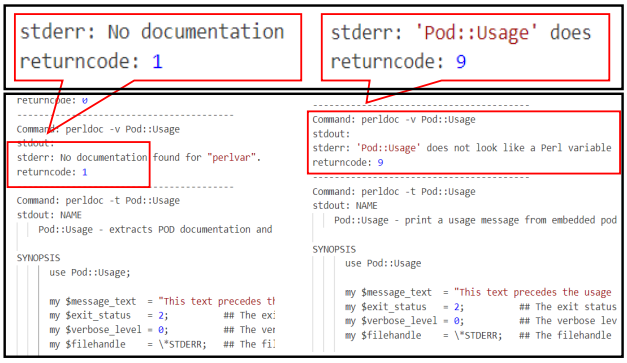
B. Test Cases Execution

Executing the programs is largely an engineering problem: each execution involves inputting test cases, which are then monitored to collect runtime information. However, the correct execution of the test cases is not straightforward, as the

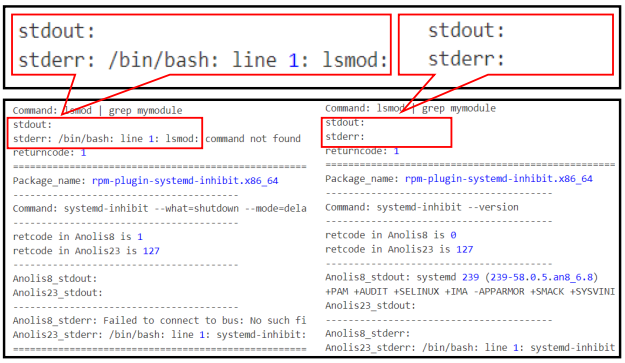
program's functionality may be influenced by environmental dependencies such as environment variables, configuration files, or command arguments. Therefore, we also need to address the following challenge.

Challenge 3: Complex and interdependent testing environment. During the execution of test cases, due to the involvement of multiple operating systems and different software versions, the testing environments are often inconsistent. In this event, the same test cases may output different results across different testing environments, which can easily lead to misreports behaviors. Furthermore, since our goal is to perform comprehensive testing on large-scale software within the system, issues such as package dependencies and conflicts between software packages may further increase the difficulty of test cases execution. As a result, to minimize misreports and maximize the correct execution of test cases during this stage as much as possible, addressing the challenge of complex and interdependent testing environments becomes a key task.

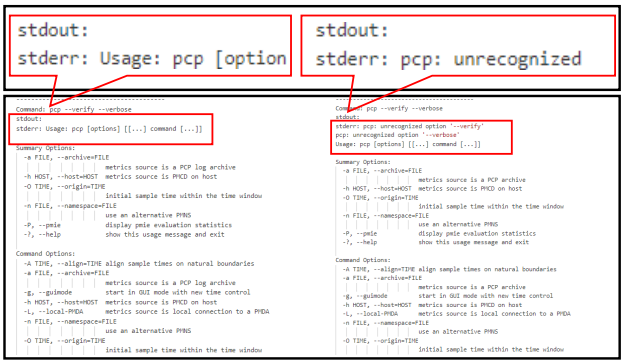
Solution 3: Create package-level testing environment using containers. We utilize Docker, which is a environment isolation technology, to address this challenge. First, we construct an initial image to manage the complexity of dependencies. Specifically, we begin by manually handling some of the environment dependencies, such as creating files that are likely to be used by the software packages in their test cases and configuring basic environment variables. The LLM is tasked with generating resource-dependent commands along with test cases to ensure that they can be executed correctly. This process significantly increases the success rate of test cases execution, which is particularly crucial for popular software. In addition, the lifecycle of each container begins with the installation of a software package and ends once all test cases for that software package have been executed. By constructing package-level separated testing environments, Versify minimizes issues such as dependency conflicts between software packages. This not only reduces misreports but also simplifies the testing process and facilitate subsequent continuous differential testing. Additionally, although our current deployment focused on AnolisOS, the package-level containerized testing approach is also applicable to other Linux distributions such as Debian, Ubuntu, and Fedora, which feature different ecosystems and dependency structures. With the above techniques, we addressed this challenge effectively.



(a) Retcodes are not equal.



(b) Stderrs/stdouts are not coexisting.



(c) Stderrs/stdouts are not equal.

Fig. 4: Examples of reports with different priorities.

C. Differential Analysis

Challenge 4: Massive differential testing reports and misreports. Since we are testing large-scale software, even if only a few test cases are generated for each software package, the resulting test logs are still massive. Moreover, differential testing must distinguish between 'results' (*e.g.*, output values) and 'behavior' (*e.g.*, program flow or interactions with other components) difference. After collecting the test cases execution information from the test cases of both two systems, a difference in 'results' simply indicates that there is a discrepancy between the two sets of information, but these discrepancies in results may not always represent significant behavioral changes, leading to potential misreports. However, there are currently no suitable evaluation criteria and misreports filtering rules for our task. Furthermore, community developers tend to prioritize handling bugs that are more severe and easier to fix, which makes it essential to perform an initial filtering of reports to focus resources on the most critical issues and reduce the workload of developers. As a result, the challenge of filtering valid differential testing reports and minimizing misreports remains a significant issue in large-scale differential testing environments.

Solution 4: Construct reports priority and filter misre-

TABLE IV: The priority table of the differential behaviors, the smaller the number, the higher the severity, with level 5 indicating passing the differential testing.

Priority	Description
0	retcode1 != retcode2
1	bool(stderr1) != bool(stderr2)
2	bool(stdout1) != bool(stdout2)
3	stderr1 != stderr2
4	stdout1 != stdout2
5	/

ports by rules. To improve the proportion of valid differential reports, we establish priority for differential behaviors, then analyze misreports and extract rules to filter them. By the way, LLM can also assist in determining differential reports. Firstly, we arrange differential behaviors and differential reports by prioritizing them, as shown in Table IV, the differential testing logs are mainly composed of three parts: return code, standard error output, and standard output. Considering that the impact of different behavioral discrepancies of software packages on system compatibility may vary, we design six priority levels. Level 0 represents the most critical behavior discrepancy, which is also the most concerning for community developers and should be prioritized for processing. Specific examples are shown in Figure 4. Generally, discrepancies in return codes represent the most critical reports and should be prioritized, as differences in return codes indicate fundamentally different software behaviors, as illustrated in Figure 4a. Next, as shown in Figure 4b, cases where standard error output or standard output differ are considered moderate reports. These compatibility issues are less severe and are assigned a medium priority. Finally, as shown in Figure 4c, discrepancies in either standard error output or standard output are more common due to software or system updates. Such compatibility differences are often tolerable and therefore assigned the lowest priority. Through this prioritization approach, we can identify and address the most critical differential reports effectively.

Secondly, after identifying high-priority reports, we need to filter out misreports. Specifically, we analyze test logs to identify common misreport patterns. For example, in AnolisOS 23, the same execution error may produce additional auxiliary output. To address this, we establish rules to match these misreports, excluding all reports that conform to these patterns. While this method requires real-time updates to the rules when new types of misreports emerge, it is both highly efficient and accurate, significantly reducing the occurrence of misreports and saving human effort in verifying compatibility reports. It is worth mentioning that due to the large number and variety of software packages, manually reviewing all differential testing reports requires specialized knowledge of each software and would consume substantial resources, which is practically impossible. Therefore, when we obtain the last batch of differential test logs that require human judgment after filtering, we can sent these reports to the LLM for useful suggestions, and thereby we can evaluate these reports more accurately to reveal compatibility issues.

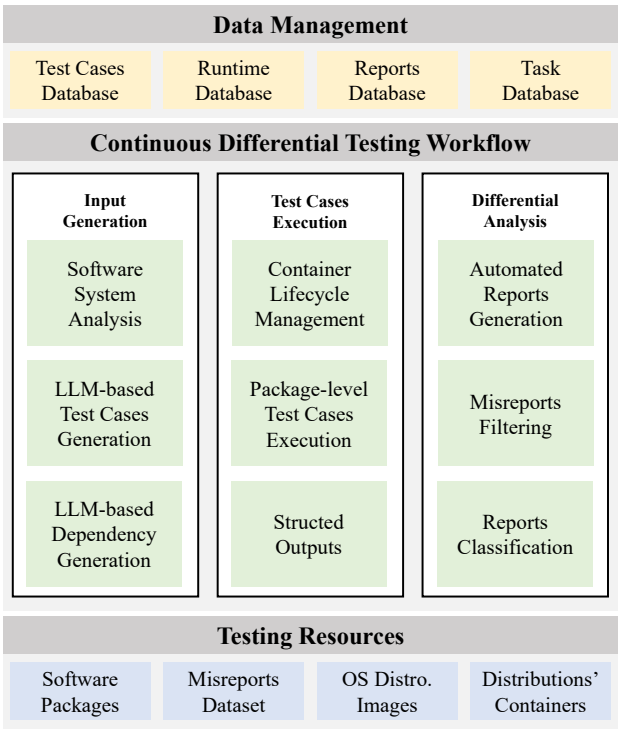


Fig. 5: Continuous Differential Testing Framework.

Finally, we manually verify the filtered differential behavior reports and provide a script to reproduce the issues, allowing community maintainers to perform the final confirmation. As a result, we can focus on reports that are more likely to indicate significant compatibility differences and effectively filter out misreports, reducing the burden of community maintainers.

D. Continuous Differential Testing

Challenge 5: Complexity of continuous differential testing. Differential testing is often perceived as a single-stage process, which concludes once the software is tested and the results are reported. However, this approach overlooks the iterative nature of software updates, where frequent changes can introduce new discrepancies and behaviors that need to be continuously tested. Moreover, the current methods for differential testing are often inadequate in handling large-scale software systems, as they lack efficient testing frameworks and fail to ensure the accuracy of results due to the high volume of updates. Continuous differential testing should be seen as a sustained process, integrating with development pipelines and leveraging automated testing tools for frequent updates. This requires not only robust testing frameworks but also tools for analyzing and tracking test results over time. However, the complexity of managing continuous differential testing, especially with large-scale software environments and diverse configurations, presents significant challenges.

Solution 5: Develop a continuous differential testing framework. To address the challenges of continuous differential testing, a comprehensive solution should be implemented that ensures that testing is a continuous process. The continuous differential testing framework is illustrated in Figure 5. The database mainly consists of four components: test cases, runtime logs, reports, and tasks. Specifically, test cases in the database are executed to compare behavioral differences and quickly identify those triggering vulnerabilities. The runtime database stores execution logs and provides a basis for reproducing issues by avoiding the need to re-execute test cases during vulnerability analysis. The reports database records detailed differential reports information and tracks the number of test cases, aiding in the interpretation and evaluation of test results while offering a shortcut for reproducing vulnerabilities. Finally, the tasks database manages and schedules all activities within the testing process, optimizing the execution order of tasks and resource allocation. By decoupling various processes through the collection of outputs at each step, Versify effectively shortens the testing cycle.

Subsequently, the continuous differential testing workflow includes three steps. In the input generation step, after obtaining the target software, LLM makes the test case generation efficient and automated, which is a prerequisite for continuous differential testing. Additionally, testing resources ensure that test cases can be executed in separated environments at the package-level. Specifically, by using Docker to save the initial test environment as a snapshot, dependency conflicts between software are avoided, making continuous differential testing feasible and reducing misreports. After the test cases execution, compatibility reports generated by differential analysis are submitted to the corresponding community maintainers. The recycling of these three steps forms the whole framework of continuous differential testing, improving the efficiency of large-scale software testing.

V. Results

After overcoming these challenges, we successfully implemented Versify and conducted continuous differential testing for the target AnolisOS versions. The default LLM used in our experiments is ChatGPT-4o. In Section VI, we will discuss the impact of different LLMs on the results.

During the first month of use in AnolisOSs, we have effectively generated about 58000 test cases for all 3882 system software in both versions of AnolisOSs. We identified a total of 8489 behavioral differences, as shown in Table V. After filtering misreports through differential analysis, we obtained 1989 differential reports of software packages in AnolisOS, out of which 644 reports belong to the priority range between 0-2 among these differential reports. Given that different versions of Linux distributions are not fully compatible, and differences in the design of the operating systems may also bring out these compatibility reports, we further conducted a manual review of the compatibility reports and finally a total of 39 reports were identified as unique compatibility reports. Note that the "-" symbol in Table V indicates that the corresponding reports are classified as lower-priority and are currently under review; this does not imply that they are not compatibility issues.

TABLE V: Statistical results of differential testing on 58,000 generated test cases.

Priority	#Diff. Behaviors	#Diff. Reports	#Compatibility Issues
0	1472	563	39
1-2	85	81	-
3-4	6932	1345	-

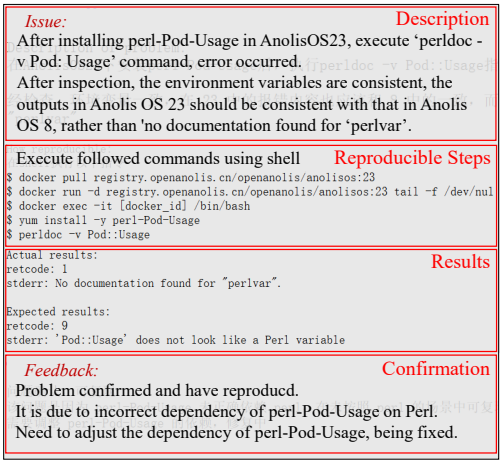


Fig. 6: Confirmed compatibility issues about software package *perl-Pod-Usage*.

A. Differential Reports Analysis

In this section, we discuss the differential reports obtained during the experiment. After filtering all observed behavioral differences, we identified 1,989 differential reports, most of which reflected semantic discrepancies in output but they are still intermediate results. Among these, 644 were prioritized in our severity rating as requiring more attention.

Since certain compatibility issues may be tolerable, we further evaluated 39 compatibility reports from the perspective of system interoperability. These reports highlight areas where the system may need improvement. Currently, the compatibility issues confirmed by community maintainers among the submitted reports mainly include problems such as *issues with software package source code*[2], *incorrect configuration of dependencies*[3], and *conflicting requests*, etc. As shown in Figure 6, command *perldoc -v Pod:Usage* triggers the unexpected result, mainly due to the incorrect dependency of *perl-Pod-Usage* on *Perl*, actually the result in AnolisOS 8 is the expected one, therefore we found the compatibility issue successfully and community maintainers have confirmed it and conclude that they need to adjust the dependency of

[2]https://bugzilla.openanolis.cn/show_bug.cgi?id=12247
[3]https://bugzilla.openanolis.cn/show_bug.cgi?id=12344

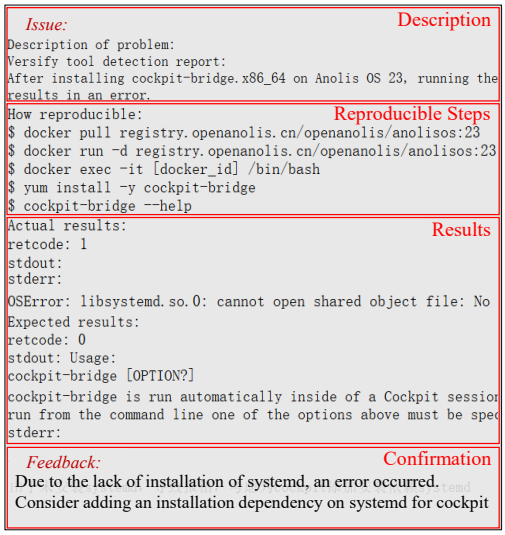


Fig. 7: Confirmed compatibility issues about software package *cockpit-bridge*.

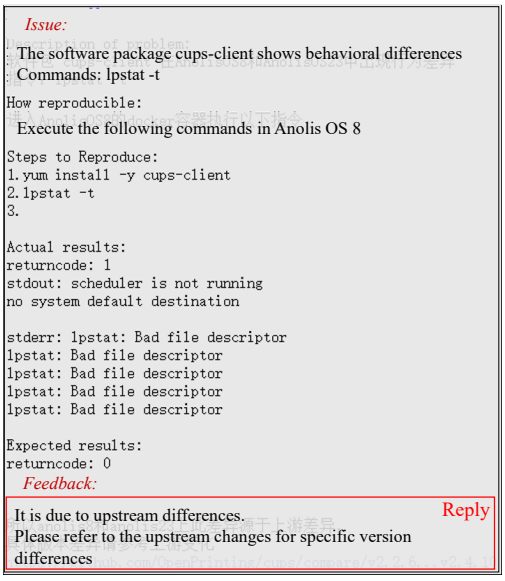


Fig. 8: Misreport of package *cups-client* due to the discrepancies of default versions across operating systems.

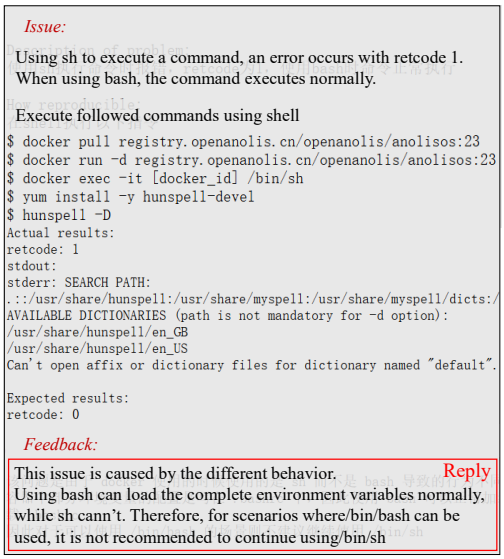


Fig. 9: Misreport of package *hunspell* due to no longer recommended *sh* to execute commands.

perl-Pod-Usage. As shown in Figure 7, AnolisOS 8 fails to fulfill the dependency requirements of *cockpit-bridge*, leading to an unforeseen compatibility issue. This issue was confirmed by the community maintainers during the process of reproducing it using the command *cockpit-bridge –help* in both operating systems. Additionally, there are some compatibility issues with unknown causes that require further investigation. These findings reveal that the identified problems are not only related to the design of software systems but also to flaws within the software itself. This demonstrates that Versify is capable of detecting software compatibility across different operating systems as well as identifying vulnerabilities within the software itself, which is a point well worth the attention of software developers.

B. Misreports Analysis

During our manual review of all the differential reports, we identified 22 misreports that were not compatibility issues but cause different outputs between two systems. We investigated the reasons behind these misreports and found that they were mainly due to the following factors.

First, LLM-generated test cases are not always robust. Some of them are semantically or syntactically incorrect, leading to undefined behaviors when software packages are executed. Therefore, no matter the execution outputs are different or not, both software packages and operating systems have correctly handled these test cases, and thus the differences are not actual compatibility issues. To avoid such misreports, we need to improve the robustness of the test cases generated by LLMs.

Second, the versions of software packages are not aligned across different systems, leading to discrepancies in execution outputs[4]. Variations in software versions across operating systems can naturally result in execution differences for same test cases. Figure 8 shows an example of such misreport, where OS developers consider the discrepancy is caused by the upstream version differences. Such misreports are unavoidable and thus need manual investigation for confirmation.

Third, a part of features may be deprecated or removed in the new version of operating systems, which leads to different outputs between two systems[5]. Figure 9 is an example of such

[4]https://bugzilla.openanolis.cn/show_bug.cgi?id=12251

[5]https://bugzilla.openanolis.cn/show_bug.cgi?id=12346

TABLE VI: The statistics of 1,000 test cases generated by four LLMs for popular software.

Selected LLM	#Diff. Reports	Time Consumed (s)
Qwen-Plus	10	385.59
ChatGPT-4o	10	147.93
DeepSeek-V3	9	405.8
DeepSeek-R1	20	1070.39

misreport. This is because the *sh* is no longer recommended to be used in the new version of AnolisOS, while it is the default shell in the old version. With assistance from community maintainers, we discovered that such differences are mainly due to variations in how environment dependencies are loaded. Such a misreport is difficult to filter out, as it is caused by the design of the operating system itself. Therefore, the feedback from community developers is essential to identify them.

C. Evaluation of Different LLMs

In this section, we evaluate how different models perform in generating test cases for software differential testing. We consider three general-purpose LLMs, Qwen-Plus, ChatGPT-4o, and DeepSeek-V3, together with a reasoning-enabled LLM, DeepSeek-R1. The results are shown in Table VI. We instruct each model to generate a total of 1,000 test cases for a certain set of popular software packages, and subsequently execute the generated cases within our workflow. The Diff. Reports in Table VI present the differential reports with priority 0, which are the most critical and indicative of an LLM's ability to identify compatibility issues in software packages.

Our results indicate that the three general-purpose LLMs deliver comparable effectiveness. ChatGPT-4o exhibits faster response times, presumably due to its smaller parameter count. The reasoning-enabled LLM DeepSeek-R1 shows superior capability, uncovering twice as many differential reports as the general-purpose models. However, its time cost is considerably higher, exceeding theirs by a factor of more than 2.5x. This experiment also confirms that our workflow can flexibly incorporate different LLMs to meet diverse task requirements.

VI. Lesson learned

In this section, we share our insights and lessons learned from the deployment of Versify in AnolisOS.

Exploring the Potential of LLMs in Differential Testing. Versify has been in use for over a year in the AnolisOS testing process. Before its deployment, the annual number of reported issues in the community was 523. After its deployment, the number has dropped to 95, indicating a significant reduction in the number of issues reported by AnolisOS users. Although LLMs are effective, their performance is limited by factors such as attention mechanisms and context window size, particularly when handling industrial-scale scenarios. Our experiments demonstrate that incorporating more powerful LLMs can decrease prompt dependency, enabling greater flexibility in generating accurate results. Future research should investigate the balance of LLM strength and computational efficiency.

Enhancing Collaboration and Reducing Human Labor. Differential testing often prioritizes discrepancy analysis and environmental setup, contrasting with industrial focus on high-severity vulnerabilities. To bridge this gap, fostering collaboration through shared frameworks and automated discrepancy filtering could minimize manual workload and enhance practical utility. For example, leveraging LLMs to pre-categorize discrepancy reports based on severity could reduce human intervention by approximately 90%, as seen in our testing pipeline. Promoting collaboration with community maintainers is crucial to align testing practices with real-world needs.

Improving Misreport Reduction. Misreport reduction remains a key challenge. Differential testing aims to distinguish between compatibility differences and genuine vulnerabilities, but defining this boundary is inherently context-sensitive. We leave this challenge to future work, as it requires a deeper understanding of the specific software and its intended behavior. To address this, we recommend developing adaptive models that learn from context to improve classification accuracy.

Leveraging Self-Trained and Task-Oriented LLMs. In our preliminary experiments, we used general-purpose LLMs, which, while effective, often struggled with domain-specific tasks. Self-trained LLMs, on the other hand, can be fine-tuned with task-specific data, enhancing their understanding of the software and its environment. Therefore, we recommend investigating self-trained LLMs that are specifically designed for this task.

VII. Conclusion

In this paper, we present industry practices for differential testing on two Linux distributions, addressing key challenges in implementing differential testing for large-scale software packages in Linux distributions. We developed Versify, which integrates LLMs into differential testing for input generation and uses containers to execute test cases, building rules to analyze reports and enabling continuous testing at scale. After month-long deployment, Versify has uncovered 8,489 instances of differing behavior, of which 644 were prioritized for attention by developers. After deduplication, 39 unique compatibility reports were identified. Future work includes improving misreport filtering through context-aware models, developing task-specific LLMs for more effective input generation, and designing better metrics to distinguish benign differences from true compatibility issues.

ACKNOWLEDGMENTS

This research is supported in part by NSFC Program (No.62472448, 62202500), Alibaba Group through Alibaba Innovative Research Program, National Key R&D Program of China (No.2022YFB3104003), Hunan Provincial Natural Science Foundation (No.2023JJ40772), Changsha Science and Technology Key Project (No. kh2401027), Hunan Provincial 14th Five-Year Plan Educational Science Research Project (No.XJK23AJD022), Ministry of Education Industry-University Cooperation Collaborative Education Project (No.220500643274437), and High Performance Computing Center of Central South University.

REFERENCES

[1] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *ASE*. IEEE Computer Society, 2017, pp. 308–318.

[2] Z. Deng, G. Meng, K. Chen, T. Liu, L. Xiang, and C. Chen, "Differential testing of cross deep learning framework apis: Revealing inconsistencies and vulnerabilities," in *USENIX Security Symposium*. USENIX Association, 2023, pp. 7393–7410.

[3] K. Etemadi, B. Mohammadi, Z. Su, and M. Monperrus, "Mokav: Execution-driven differential testing with llms," *arXiv preprint arXiv:2406.10375*, 2024.

[4] E. Isaku, C. Laaber, H. Sartaj, S. Ali, T. Schwitalla, and J. F. Nygård, "Llms in the heart of differential testing: A case study on a medical rule engine," *arXiv preprint arXiv:2404.03664*, 2024.

[5] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, "Llm-powered test case generation for detecting tricky bugs," *arXiv preprint arXiv:2404.10304*, 2024.

[6] V. A. Braberman, F. Bonomo-Braberman, Y. Charalambous, J. G. Colonna, L. C. Cordeiro, and R. de Freitas, "Tasks people prompt: A taxonomy of llm downstream tasks in software verification and falsification approaches," *arXiv preprint arXiv:2404.09384*, 2024.

[7] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

[8] Y. Chen and Z. Su, "Guided differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 793–804.

[9] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 2017, pp. 615–632.

[10] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 521–538.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[12] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[13] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1690–1701.

[14] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, "A security policy oracle: Detecting security holes using multiple api implementations," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 343–354, 2011.

[15] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 80–94.

[16] 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)*, 2024, pp. 123–140.

[17] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation." in *USENIX Security Symposium*, vol. 15, 2007.

[18] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.

[19] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols." in *NDSS*, 2023.

[20] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2517–2532.

[21] F. Ma, Y. Chen, Y. Zhou, J. Sun, Z. Su, Y. Jiang, J. Sun, and H. Li, "Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos," ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1182–1196. [Online]. Available: https://doi.org/10.1145/3576915.3623071

[22] F. Ma, Y. Chen, Y. Zhou, Z. Yan, H. Sun, and Y. Jiang, "Finding metadata inconsistencies in distributed file systems via cross-node operation modeling," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[23] Y. Chen, F. Ma, Y. Zhou, Z. Yan, Q. Liao, and Y. Jiang, "Themis: Finding imbalance failures in distributed file systems via a load variance model," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 329–344. [Online]. Available: https://doi.org/10.1145/3689031.3696082

[24] Y. Chen, F. Ma, Y. Zhou, Z. Yan, and Y. Jiang, "Cafault: enhance fault injection technique in practical distributed systems via abundant fault-dependent configurations," in *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '25. USA: USENIX Association, 2025.

[25] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi, "Evm: From offline detection to online reinforcement for ethereum virtual machine," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 554–558.

[26] F. Ma, M. Ren, Y. Fu, M. Wang, H. Li, H. Song, and Y. Jiang, "Security reinforcement for ethereum virtual machine," *Inf. Process. Manage.*, vol. 58, no. 4, Jul. 2021. [Online]. Available: https://doi.org/10.1016/j.ipm.2021.102565

[27] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1258–1268.

[28] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1583–1594.

[29] R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, "Multi-language unit test generation using llms," *arXiv preprint arXiv:2409.03093*, 2024.

[30] Z. Xue, L. Li, S. Tian, X. Chen, P. Li, L. Chen, T. Jiang, and M. Zhang, "Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1643–1655.

[31] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[32] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[33] Anolis OS, "Anolis os 23," 2024, accessed: 2024-05-06. [Online]. Available: https://package.openanolis.cn/release/Anolis%20OS%2023

[34] Anolis. (2024) Anolis next os software. [Online]. Available: https://gitee.com/anolis/community/blob/371adde472c7159394953074665fe2aa6577cd35/sig/sig-distro/content/articles/next_os_software.md

[35] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2312–2331, 2021.

[36] M. R. Golla and S. Godboley, "Automated SC-MCC test case generation using coverage-guided fuzzing," *Softw. Qual. J.*, vol. 32, no. 3, pp. 849–880, 2024.

[37] OpenAI, "API docs - guide to prompt engineering," https://platform.openai.com/docs/guides/prompt-engineering, accessed: 2025-01-15.

[38] M. Zalewski, "American Fuzzy Lop (AFL) - Security-Oriented Fuzzer," https://lcamtuf.coredump.cx/afl/, 2025, accessed: 2025-01-22.

[39] Google Inc., "syzkaller - Unsupervised Coverage-Guided Kernel Fuzzer," https://github.com/google/syzkaller, 2023, accessed: 2025-01-22.