ConfD: Analyzing Configuration Dependencies of File Systems for Fun and Profit

Tabassum Mahmud, Om Rameshwar Gatla, Duo Zhang, Carson Love, Ryan Bumann, and Mai Zheng, *Iowa State University*

https://www.usenix.org/conference/fast23/presentation/mahmud

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

CONFD: Analyzing Configuration Dependencies of File Systems for Fun and Profit

Tabassum Mahmud, Om Rameshwar Gatla, Duo Zhang, Carson Love, Ryan Bumann, Mai Zheng
Department of Electrical and Computer Engineering, Iowa State University, Ames, IA

Abstract

File systems play an essential role in modern society for managing precious data. To meet diverse needs, they often support many configuration parameters. Such flexibility comes at the price of additional complexity which can lead to subtle configuration-related issues. To address this challenge, we study the configuration-related issues of two major file systems (i.e., Ext4 and XFS) in depth, and identify a prevalent pattern called multilevel configuration dependencies. Based on the study, we build an extensible tool called CONFD to extract the dependencies automatically, and create six plugins to address different configuration-related issues. Our experiments on Ext4 and XFS show that CONFD can extract more than 150 configuration dependencies for the file systems with a low false positive rate. Moreover, the dependency-guided plugins can identify various configuration issues (e.g., mishandling of configurations, regression test failures induced by valid configurations).

1 Introduction

File systems (FS), such as Ext4 [54] and XFS [89] on Linux-based operating systems (OS) and NTFS [76] on Windows OS, play an essential role in modern society. They directly manage various files on desktops, laptops, and smartphones for numerous end users [12]. Moreover, they often serve as the local storage backend for distributed storage systems (e.g., Lustre [63], GFS [1], HopsFS [22], MySQL NDB Cluster [73]) to enable storage management at scale.

To meet diverse needs, many file systems are designed with a wide range of configuration parameters controllable via utilities [41, 45, 48, 52, 56, 58, 94, 100], which enables users to tune the systems with different tradeoffs. For example, Ext4 contains more than 85 configuration parameters which can be modified through a set of utilities called `e2fsprogs` [52]. The combination of the configuration parameters represents over 10^{37} configuration states [32].

While configuration parameters have improved the system flexibility, they introduce additional complexity for reliability.

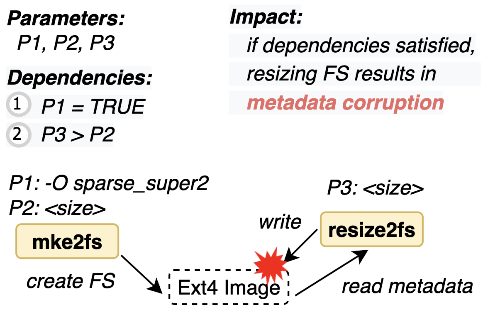

Figure 1: **A Configuration-Related Issue of Ext4.** When `sparse_super2` feature is enabled and the `size` parameter of `resize2fs` is larger than the Ext4 size, expanding the file system results in metadata corruption.

Subtle correctness issues often rely on specific parameters to trigger [6, 13]; consequently, they may elude intensive testing and affect end users negatively. For example, in December 2020, users of Windows OS observed that the checker utility of NTFS (i.e., `ChkDsk` [45]) may destroy NTFS on SSDs [60, 88]. The incident turned out to be configuration-related: two specific parameters must be satisfied to manifest the issue, including the '`/f`' parameter of `ChkDsk` and another (unnamed) parameter in Windows OS [87].

Similarly, Figure 1 shows another configuration-related issue involving Ext4 and its `mke2fs` and `resize2fs` utilities [52]. Two conditions must hold to trigger the bug: (1) the `sparse_super2` feature is enabled in Ext4 (via `mke2fs`); (2) the value of the `size` parameter of `resize2fs` must be larger than the size of Ext4 (i.e., expanding the file system). Once triggered, the bug will corrupt the Ext4 metadata with incorrect free blocks. The root cause behind the issue was logical: with the specific configuration, the free block count of the last block group of Ext4 was calculated before adding new blocks for expansion.

Due to the combinatorial explosion of configuration states and the substantial time needed to scrutinize a file system under each configuration state, it is practically impossible to exhaust all states for thorough testing today [9]. Moreover, with more and more heterogeneous devices and advanced features being introduced [65, 83, 86], the configuration states are expected to grow. Therefore, effective methods to help improve configuration-related testing and identify critical configuration issues efficiently are much needed.

1.1 Limitations of the State of the Art

There are practical test suites to ensure the correctness of file systems under different configurations (e.g., `xfstests` [95]). Unfortunately, their coverage in terms of configuration is limited: fewer than half of configuration parameters are used based on our study, which reflects the need for better tool support. Also, configuration-related issues have emerged in other software systems and have received much attention [4, 13, 24, 33, 35]. But unfortunately, existing efforts mainly focus on relatively simple configuration issues (e.g., typos [4]) within one single application, which is limited for addressing the file system configuration challenge involving multiple programs. Please refer to §2 for more details.

1.2 Our Efforts & Contributions

This paper presents one of the first steps to address the increasing configuration challenge of file systems. Inspired by a recent study [33] on configuration issues in Hadoop [40] and OpenStack [77], we focus on *configuration dependency*, which describes the dependent relations among configuration parameters [33]. Such dependency has been identified as a key source of complexity caused problems, and capturing the dependency is essential for improving configuration design and tooling [13, 19, 33].

While the basic concept of configuration dependency has been proposed in the literature (see §2), the understanding of specific dependency patterns and implications in the context of file systems is still limited. Therefore, we first conducted an empirical study on 78 configuration-related issues in two major file systems (i.e., Ext4 and XFS). By scrutinizing real-world bugs and the relevant source code, we answer one important question: What critical configuration dependencies exist in file systems?

Our study reveals a prevalent pattern called *multilevel configuration dependencies*. Besides the relatively simple configuration constraints (e.g., value range [13]), there are implicit dependencies among parameters from different utilities of a file system. The majority (96.2%) of issues in our dataset requires meeting such deep configuration dependencies to manifest. Interestingly, the workloads applied to the file system do not have to be configuration-specific: 71.8% issues only involve generic file system operations.

Based on the study, we built an extensible framework called CONFD to extract the multilevel configuration dependencies automatically and leverage dependency-guided configuration states for further analysis. One key challenge is how to establish the correlation between parameters specified through different utilities which have different ways of configuration handling. We address the challenge by metadata-assisted taint analysis, which leverages the fact that all utilities of a given file system share the same metadata structures. Moreover, based on the dependencies extracted, we created six plugins to help address configuration-related issues in file systems from different angles.

Our experiments show that CONFD can extract 154 different configuration dependencies with a low false positive rate (8.4%) for Ext4 and XFS. Moreover, with the dependency guidance, the CONFD plugins can identify various configuration-related issues, including inaccurate documentations, configuration handling issues, and regression test failures induced by valid configurations.

In summary, this paper makes the following contributions:

- Deriving a taxonomy of critical configuration dependencies of file systems based on real-world issues.
- Building the CONFD prototype [1] to extract configuration dependencies and expose relevant issues in file systems.
- Integrating with multiple practical tools (e.g.,fault injector [25], fuzzer [29], regression test suites [51, 95]) to improve their configuration coverage and effectiveness.
- Evaluating the methodology on two widely used file systems and demonstrating the effectiveness.

The rest of the paper is organized as follows: §2 introduces the background and related work; §3 presents the empirical study and findings; §4 describes the CONFD framework; §5 shows experimental results; §6 discusses limitations and potential extensions; §7 concludes the paper.

2 Background & Related Work

2.1 Background

File System Configurations. The configuration methods of file systems are different from that of many applications, which makes the problem arguably more challenging. As shown in Figure 2, a typical file system may be configured through a set of utilities at four different stages:

- **Create**. When creating file systems, the `mkfs` utility (e.g., `mke2fs` for Ext4) generates the initial configurations.
- **Mount**. When mounting file systems, certain configurations can be specified via `mount` (e.g., '`-o dax`' to enable the Direct Access or DAX feature [65]).

[1] CONFD is on https://github.com/data-storage-lab/ConfD

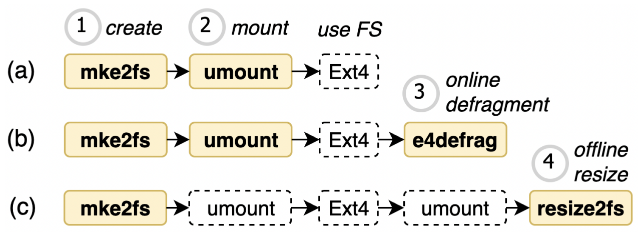

Figure 2: **Methods of Configuring File Systems**. This figure shows four typical stages to configure a file system: (a) at creation (e.g., `mke2fs`) or mount time (`mount`) before usage; (b) via online utilities (e.g., `e4defrag`); (c) via offline utilities.

- **Online**. Many utilities can change the configurations of a mounted file system directly by modifying the metadata online (e.g., Ext4 defragmenter `e4defrag` [53], Windows NTFS checker `ChkDsk` [45]).
- **Offline**. Offline utilities can also modify file system images and change the configurations (e.g., `resize2fs` [79], `e2fsck` [50])

Note that all the utilities have different configuration parameters to control their own behaviors, which will eventually affect the file system state. Moreover, the configuration parameters may affect the behavior of the file system long after the FS image is created, and some configurations cannot be changed later. Also, the validation of parameters may occur at both user level and kernel level. For example, the '`-O inline_data`' parameter of `mke2fs` and the '`-o dax`' of `mount` are further validated in the `ext4_fill_super` function of Ext4. Therefore, we believe it is necessary to consider the file system itself as well as all the associated utilities as an *FS ecosystem* to address the configuration challenge. For simplicity, we call the file system and utilities as *components* within the FS ecosystem.

The multi-stage configuration method is common among file systems. As listed in Table 1, many popular file systems follow similar modular designs and can be configured via different utilities at different stages. Therefore, we believe that the multi-component configuration challenge is general. We focus on Ext4 and XFS in this work because they are two major file systems on Linux and they support the latest DAX [65] configuration for non-volatile memories (NVM). We leave the others as future work (§6).

FS Test Suites. Practical test suites have been created to ensure the correctness of file systems under various configurations. Unfortunately, due to the complexity of configurations, their coverage in terms of configuration is limited. As shown in Table 2, fewer than half of configuration parameters are used in the standard test suites of Linux file systems (i.e., `xfstests` [95], `e2fsprogs/tests` [51]) based on our study. Since each parameter may have a wide range of values representing different states, the total number of missed

FS (OS)	Four Stages of Configuration			
	Create	Mount	Online	Offline
Ext4 (Linux)	[66]	[69]	[53], [79]	[50], [79]
XFS (Linux)	[68]	[69]	[91], [92]	[90], [93]
BtrFS (Linux)	[67]	[69]	[41], [43]	[42]
UFS (FreeBSD)	[75]	[70]	[59], [80]	[49], [58]
ZFS (FreeBSD)	[96]	[98]	[99], [100]	[97]
NTFS (Windows)	[55]	[72]	[45], [46]	[45], [81]
APFS (MacOS)	[48]	[71]	[48]	[48], [57]

Table 1: **Examples of configuration methods for different file systems**. The last four columns list example utilities that can affect the file system configuration states.

Test Suite	Target Software	# of Conf. Param.	
		Total	Used
`xfstests`	Ext4	>85	29 (< 34.1%)
`e2fsprogs`	`e2fsck`	>35	6 (< 17.1%)
`/tests`	`resize2fs`	>15	7 (< 46.7%)

Table 2: **Configuration Coverage of Test Suites.**

configuration states is much more than the number of unused parameters, which implies the need for better tool support.

Configuration Constraints & Dependencies. Configuration *constraints* specify the configuration requirements (e.g., data type, value range) of software [13]. Intuitively, such information can help identify important configuration states, and it has proved to be effective for addressing configuration-related issues in a wide range of applications [4, 8, 13, 14, 33]. Configuration *dependency* is one special type of constraint describing the dependent correlation among parameters [13, 33], which has shown recently to be critical for addressing complex configuration issues in cloud systems [33]. For simplicity, we use constraints and dependencies interchangeably in the rest of the paper. Note that although the basic concepts have been proposed, there is limited understanding of them in the context of file systems. This paper attempts to fill the gap.

2.2 Related Work

Analysis of Software Configurations. Configuration issues have been studied in many software applications [4, 6, 7, 13, 14, 24, 33, 35]. For example, ConfErr [4] manipulates parameters to emulate human errors; Ctests [35] detects failure-inducing configuration changes. In general, these works do not analyze deep dependencies within the software. The closest work is cDEP [33], which notably observes *inter-component dependencies* in Hadoop [40] and OpenStack [77]. Unfortunately, their solution is largely inapplicable for file systems. This is because their target components share configuration specifications (e.g., XML) and libraries [39], which makes them equivalent to one single program in terms of configuration. In contrast, the configuration dependencies in file systems may cross different programs and the user-kernel boundary, which requires non-trivial mechanisms to extract.

In addition, cDEP relies on a Java framework [82] which cannot handle C-based file systems.

Reliability of File Systems. Great efforts have been made to improve the reliability of file systems [2, 10, 17, 18, 29] and their utilities [3, 25, 26, 27, 78]. For example, Prabhakaran et al. [2] apply fault injection to analyze the failure policies of file systems and propose improved designs based on the IRON taxonomy; Xu et al. [30] and Kim et al. [29] use fuzzing to detect file system bugs; SQCK [3] and RFSCK [25] improve the checker utilities of file systems to avoid inaccurate fixes. While effective for their original goals, these works do not consider multi-component configuration issues. On the other hand, the configuration dependencies from this work may be integrated with these existing efforts to improve their coverage (see §4.2). Therefore, we view them as complementary.

Configuration Management Tools. Faced by the increasing challenge, practitioners have created dedicated frameworks for configuration management [31, 44]. For example, Facebook HYDRA [31] supports managing hierarchical configurations elegantly. While helpful for developing new applications, refactoring FS ecosystems to leverage such frameworks would require substantial efforts (if possible at all). Notably, the framework supports running a program with different compositions of configurations automatically. Nevertheless, since it does not understand configuration dependencies, it may generate many invalid configuration states (see §5.2.3). This work aims to address such limitations.

3 Configuration Dependencies in File Systems

In this section, we present a study on the Ext4 and XFS ecosystems to understand the potential patterns of configuration issues and guide the design of solutions. We discuss the methodology and findings in §3.1 and §3.2, respectively.

3.1 Methodology

Our dateset includes two parts: (1) the source code of Ext4 and XFS and seven important utilities including `mke2fs`, `mount`, `e4defrag`, `resize2fs`, `e2fsck`, `mkfs.xfs`, and `xfs_repair`, which are described in Table 3; (2) a set of 78 configuration-related bug patches for the two FS ecosystems, which are collected from the commit histories of their source code repositories via a combination of keyword search (e.g., 'configuration', 'parameter', 'option'), random sampling, and manual validation. Note that the patch collection method is inspired by previous studies of real-world bugs [5, 12, 36]. While time-consuming, it has proved to be valuable for driving system improvements [5, 12]. On the other hand, similar to previous studies, the findings of our study should be interpreted with the method in mind. For example, the 78 patches only represent a subset of issues that have been triggered and fixed; there are likely other configuration-related issues not yet discovered (see §6 for further discussion).

3.2 Findings

Based on the dataset, we analyzed each patch and the relevant source code in depth to understand the logic, which enables us to identify the configuration usage scenarios as well as configuration constraints that are critical. We summarize our findings in Table 3 and Table 4 and discuss them below.

Finding #1: *The majority of cases (96.2%) involve critical parameters from more than one component*. The first column of Table 3 shows six typical usage scenarios of file systems which cover all bug cases in our dataset (78 in total). 96.2% of the bug cases require specific parameters from at least two key utilities (i.e., the utilities in bold in each usage scenario) to manifest. This reflects the complexity of the issues and suggests that we cannot only consider one single component.

Finding #2: *There is a hierarchy of configuration dependencies*. We classify the configuration constraints derived from our dataset into three major categories as follows:

- **Self Dependency (SD)** means individual parameters must satisfy their own constraints (e.g., data type or value range). For example, the `blocksize` parameter of `mke2fs` has a value range of 1024 - 65536 and must be a power of 2.
- **Cross-Parameter Dependency (CPD)** means multiple parameters of the same component must satisfy relative relation constraints (e.g., two `mke2fs` parameters `meta_bg` and `resize_inode` cannot be used together).
- **Cross-Component Dependency (CCD)** means the parameters or behaviors of one component depend on the parameters of another component. Both dependencies in Figure 1 belong to this category becasue they involve parameters of `mke2fs` and the (buggy) behavior of `resize2fs` depend on them.

As summarized in Table 4, each major category may contain a couple of sub-categories which describe more specific constraints. Together, these constraints form a hierarchy which we call *multilevel configuration dependencies*. Note that we only observed 7 out of 8 sub-categories in the dataset. We include the unseen "Value" sub-category in CPD based on the literature [13] for completeness.

Moreover, among all the dependencies, there is a subset which directly contribute to the manifestation of the bugs in our dataset: the relevant parameters are explicitly mentioned in the bug patches, and modifications to the corresponding functionalities are needed to fix the bugs (i.e., they are related to the root causes). We call this subset of dependencies as *critical dependencies*. The count of the critical dependencies for each sub-category is shown in the last column of Table 4. We are able to derive 168 critical dependencies manually in total, which is larger than the number of bug cases. This is because multiple critical dependencies may be needed to

FS Usage Scenarios (key configuration utilities are in bold)		Description	# of Bug	Multilevel Config. Dependencies		
				SD	CPD	CCD
1	**mke2fs** - **mount** - Ext4	create & mount an Ext4 to use	13	13 (100%)	1 (7.7%)	13 (100%)
2	**mke2fs** - **mount** - Ext4 - **e4defrag**	online defragmentation	1	1 (100%)	–	–
3	**mke2fs** - mount - Ext4 - umount - **resize2fs**	resize an umounted Ext4	17	17 (100%)	–	17 (100%)
4	**mke2fs** - mount - Ext4 - umount - **e2fsck**	check Ext4 & fix inconsistencies	36	36 (100%)	4 (11.1%)	34 (94.4%)
5	**mkfs.xfs** - **mount** - XFS	create & mount an XFS to use	5	5 (100%)	2 (40%)	5 (100%)
6	**mkfs.xfs** - mount - XFS - umount - **xfs_repair**	check XFS & fix inconsistencies	6	6 (100%)	1 (16.7%)	6 (100%)
		Total	78	78 (100%)	8 (10.3%)	75 (96.2%)

Table 3: **Distribution of Configuration Bugs in Six Scenarios.** This table shows the distribution of 78 configuration bugs in six typical usage scenarios of file system. The last three columns shows the percentages of bug cases that involve Self-Dependency (SD), Cross-Parameter Dependency (CPD), and Cross-Component Dependency (CCD), respectively.

Multilevel Config. Dependencies		Description	Observed?	Count
Self Dependency (SD)	Data Type	parameter P must be of a specific data type (e.g., integer)	Y	44
	Value Range	P must be within a specific value range (e.g., $P < 4096$)	Y	41
Cross-Parameter Dependency (CPD)	Control	$P1$ of $C1$ can be enabled iff $P2$ of $C1$ is enabled/disabled	Y	5
	Value	$P1$'s value depends on $P2$'s value (e.g., $P1 < P2$)	N	–
	Behavioral	component $C1$'s behavior depends on $P1$ and $P2$ of component $C1$	Y	1
Cross-Component Dependency (CCD)	Control	$P1$ of $C1$ can be enabled iff $P2$ of $C2$ is enabled/disabled	Y	1
	Value	$P1$'s value depends on $P2$ from another component	Y	1
	Behavioral	component $C1$'s behavior depends on $P2$ of $C2$	Y	75
		Total	7/8	168

Table 4: **Multilevel Configuration Dependencies.** This table describes the multilevel configuration dependencies observed. *Pn* means parameter, *Cn* means component. The last column shows the count of each sub-category of dependency observed.

trigger a bug. For example, both dependencies in Figure 1 are critical dependencies for this bug case.

As shown in the last three columns of Table 3, SD and CCD are almost always involved in all scenarios (100% and 96.2% respectively), while CPD is non-negligible (10.3%). This is because SD represents relatively simple constraints which always need to be satisfied first to make the target component work (e.g., correct spelling). SD is relatively easy to check and has been the focus of previous work [4]. However, this does not mean that 100% of the bugs could be avoided if SD is checked or satisfied. For example, a bug related to both the `bigalloc` and `extent` parameters (i.e., there is a CPD involved) may still occur even if the two parameters are spelled correctly. In other words, only considering simple constraints of individual parameters is not enough.

Interestingly, we observed both CPD and CCD between the DAX feature and other seemingly irrelevant configurations. In one case, a corruption was triggered when '`-O inline_data`' was used in `mke2fs` and the image was mounted with '`-o dax`' subsequently. In another case, the DAX feature conflicted with the '`has_journal`' configuration, which may lead to corruptions when changing the journaling mode online. Such unexpected dependencies implies the complexity of adding the DAX support to the Linux kernel.

Finding #3: *Configuration parameters are handled in heterogeneous ways in an FS ecosystem.* We identified four major sources of heterogeneity in FS configurations. First, different parameters may be mapped to different types of variables in the code. For example, the parameters of Ext4 may be stored in (at least) four different ways including (i) a local variable, (ii) a global variable, (iii) a bit in a bitmap accessed via bit operations, and (iv) directly in the superblock. Second, within the superblock, parameters may be kept either in one single field (e.g., `s_log_block_size`) or as one member of a compound field. Third, parameters can be loaded from the superblock either directly or through library calls. Lastly, different components may use different functions for handling configurations (e.g., `resize2fs` uses the "main" function, while `mke2fs` invokes a special function called "PRS"). Such heterogeneity makes previous solutions mostly inapplicable.

Finding #4: *The majority of cases (71.8%) do not require configuration-specific workloads to manifest.* Interestingly, despite the complexity, many bugs can be triggered without applying configuration-specific workloads. This suggests that we may re-use existing efforts on stressing file systems [51, 95] to analyze configuration-related issues effectively.

4 Extracting & Using Multilevel Configuration Dependencies

Based on the study, we built an extensible framework called ConfD to leverage the dependency information to address configuration-related issues. As shown in Figure 3, ConfD consists of two main parts: (1) *ConfD-core* (yellow box) for extracting multilevel configuration dependencies and generat-

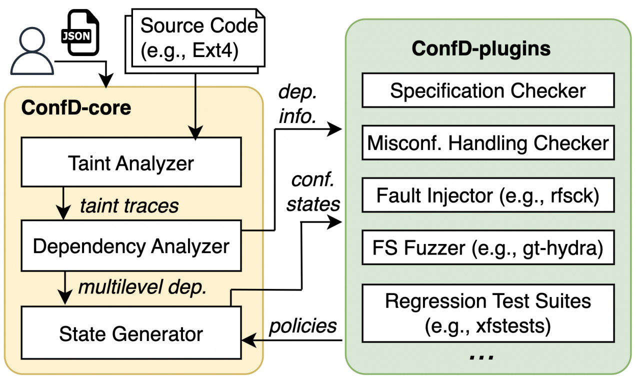

Figure 3: **Overview of ConfD**. There are two parts: (1) ConfD-core (yellow) for extracting configuration dependencies and generating critical states; (2) ConfD-plugins (green) for detecting various configuration-related issues.

ing critical configuration states, which further contains three sub-modules (i.e., *Taint Analyzer*, *Dependency Analyzer*, and *State Generator*); (2) *ConfD-plugins* (green box) for detecting various configuration-related issues based on the generated configuration states. We elaborate on the two parts in the following two subsections respectively.

4.1 Extracting Configuration Dependencies

4.1.1 Metadata-assisted Taint Analysis

As the first step, the *Taint Analyzer* of ConfD performs metadata-assisted taint analysis and generates taint traces to capture the propagation flow of configuration parameters in the target FS ecosystem.

It takes the source code of the target system as input, and uses the LLVM compiler infrastructure [85] to generate intermediate representation (IR) of the source code. It then tracks the propagation of each configuration parameter along the data-flow paths in IR based on the classic taint analysis algorithm [21]. We maintain a set to keep the initial configuration variables and any variables derived from the initial configuration variables while traversing the IR. When a new variable is added to the set, we add the corresponding IR instruction to the taint trace. We maintain a mapping between each configuration parameter and the variables derived from it to enable tracking if a variable may be derived from multiple parameters, which is essential for establishing the correlation across parameters. Our taint analysis is context-sensitive and can handle both intra-procedural and inter-procedural analysis. Context-sensitivity is important for inter-procedural analysis because one function can be called from different contexts, which is also crucial for deriving accurate dependency across different taint traces (§4.1.2).

One unique challenge we encounter is how to establish the mapping between parameters of different components of the FS ecosystem. As mentioned in §3.2, the components in the FS ecosystem tend to load configurations in different ways and process equivalent FS information using different variables or functions. We address this challenge based on one key observation: all components need to access the same FS metadata structures. We can leverage shared metadata structures to connect relevant parameters of different components.

More specifically, the parameter values relevant to the FS configuration are (eventually) stored in the superblock structure of the file system. For example, the parameter `-I inode-size` from `mke2fs` is stored as the 27th member of the superblock (`s_inode_size`). When another component (e.g., `e2fsck`) loads the `s_inode_size` from the superblock to access it, it is essentially dependent on the `-I inode-size` parameter of `mke2fs`. We map the `mke2fs` parameter values to relevant superblock fields by tracking where the parameter value is being written in the superblock. Similarly, the accesses to the superblock in other components are also tracked. Based on the mapping to the same superblock fields (e.g., `s_inode_size`), we can establish the connection between taint traces from different components.

Note that since ConfD implements the taint analysis at the LLVM IR level, any file system that can be compiled to LLVM IR may benefit from it for configuration dependency analysis. The current prototype uses the Clang frontend of LLVM which supports C/C++/Objective-C languages [85].

4.1.2 Multilevel Dependency Analysis

Given the taint trace of every configuration parameter, the *Dependency Analyzer* further analyzes the potential correlations between parameters based on the multilevel dependencies derived from our study (§3).

Specifically, the self-dependency (SD) for each parameter is derived from their individual taint traces based on the data type and value range of the variables. We also examine the error statement immediately following a range check based on the observation that an error statement may indicate an invalid range. For CPD and CCD, we compare taint traces of multiple parameters. If there are common lines (which are context-sensitive), we consider them to be dependent. Moreover, after getting the dependent parameters, we also leverage the subsequent error statements to further analyze the specific types of dependency (e.g., should be enabled or disabled together). For example, the two parameters `resize_inode` and `meta_bg` from `mke2fs` cannot be enabled together, so there must be a common error statement immediately following the condition check shared by the two taint traces. All of the extracted dependencies are stored in the JSON format [61] to describe both the parameters and the corresponding dependent relations concisely.

4.1.3 Dependency-guided State Generation

With the dependency information, the *State Generator* generates concrete configuration states for further analysis. Instead of randomly generating combinations of configurations which may easily lead to useless states (§5.2.3), it leverages the extracted multilevel dependencies to generate states selectively.

Specifically, the *State Generator* uses a tree structure to maintain different configuration states. The root of the tree represents a default configuration state, and each child node on the tree represents a configuration state with exactly one modification made from its parent. The module operates similar to a Depth First Search (DFS) on a tree, except it leverages the dependency information to guide which children nodes are worth pursuing. For example, given the cross-parameter dependency (CPD) between the `bigalloc` and `blocksize` parameters of `mke2fs`, if the current node modifies `bigalloc`, then the child node to consider will be a state with a modification to `blocksize`.

Moreover, the module has a number of options that allow for tuning based on needs. The first option is 'depth', which dictates how deep the DFS is allowed to go. A larger value results in a greater number of states being generated. The default 'depth' is 3 which worked well in our experiments. Another option is the 'policy' under which the State Generator operates. There are two basic policies as follows:

Following Dependency. Under this policy, we always honor the extracted multilevel dependencies when creating a configuration state. For example, `sparse_super` should always be enabled if `resize_inode` is enabled for `mke2fs` according to the multilevel dependency, so the module may generate a state with both parameters (i.e., '`mk2fs -O resize_inode,sparse_super`'). Essentially, this policy only generates *valid* configuration states involving critical parameters for the target FS ecosystem, which is the basic requirement for running many FS applications or tools properly. Note that this policy is consistent with recent work on testing configuration changes which shows that *valid* configuration changes may induce production failures [35].

Violating Dependency. Under this policy, we intentionally violate the multilevel dependencies when creating a configuration state. For example, the `resize_inode` and `sparse_super` parameters of `mke2fs` have a cross-parameter dependency (CPD): `sparse_super` must be enabled if we want to enable `resize_inode`. To violate the CPD, the module may intentionally generate a state which disables the `sparse_super` parameter while enabling `resize_inode` (i.e., '`mke2fs -O resize_inode,^sparse_super`'). By generating *invalid* configuration states on purpose, we enable examining the (mis)configuration handling of the target system. Note that this policy is inspired by the previous work on simulating human errors in configuration [4]. However, different from the relatively shallow violations (e.g., typos) which have been largely handled in matured systems, we consider more subtle violations that involve non-trivial dependencies.

In addition, to provide more flexibility for different use cases, the *State Generator* supports customizing the two basic policies further with different tradeoffs (e.g., the number of parameters to consider, the type of dependency (i.e., SD/CPD/CCD) to use). As mentioned, a key challenge with analyzing configurations of file systems is that the space is too huge to exhaust. For example, `mke2fs` itself has more than 8 trillion possible parameter combinations. With the dependency guidance, CONFD can reduce the space to hundreds or tens of thousands depending on the use case (§4.2), which makes the configuration testing much more manageable in practice. And as will be shown in §5.2.1, the dependency-guided state generation will be more effective than dependency-agnostic alternatives for exposing configuration issues.

4.1.4 User Input

ConfD-core needs three types of input information from the user, which can be specified in one single JSON file. First, to start the taint analysis, the *Taint Analyzer* needs a function name as the entry point. In the case of a utility program, the function (which may invoke sub-functions) is expected to be the major function for processing configurations. In the case of the file system itself, the function can be either a function for processing configurations, or a function that is interesting (e.g., a newly added FS function). Second, the taint analysis also requires the names of the variables representing the configurations and the superblock in the source code, which are often different across programs based on our experience on Ext4 and XFS ecosystems. Third, to generate valid configuration states, the *State Generator* needs the command-line syntax of FS configurations. Note that all the input can be specified in the JSON format, and it is a one-time effort for each program to be analysed.

4.2 Leveraging Configuration Dependencies

The dependency information and the dependency-guided configuration states may be used in different ways to address different issues [4, 13, 33]. As mentioned in §2.2, there are existing efforts to improve FS ecosystems which cover a wide range of techniques including fault injection [2, 25], fuzzing [29, 30], regression test suites [51, 95], etc. While these tools are excellent for their original design goals, they are mostly agnostic to configuration dependencies and thus cannot address tricky configuration-related issues effectively. The CONFD plugin interface is designed to bridge the gap by introducing dependency awareness to the traditional methodologies and thus amplify the effectiveness.

The current prototype of CONFD includes six plugins. As summarized in Table 5, the first two plugins (#1 and #2) are built from scratch, the next two plugins (#3 and #4) are based on open-source research prototypes (R), and the last two (#5 and #6) are designed for enhancing standard test suites (S). We discuss them in more details below:

Plugin ID	Description	Base Tool (type)	ConfD Plugin
#1	Configuration specification checker for Linux file systems	N/A	`ConfD-specCk`
#2	Misconfiguration handling checker for Linux file systems	N/A	`ConfD-handlingCk`
#3	An open-source fault injector for file system utilities	`rfsck` [25] (R)	`ConfD-rfsck`
#4	An open-source fuzzer for file systems	`gt-hydra` [29] (R)	`ConfD-gt-hydra`
#5	Regression test suite for Linux file systems	`xfstests` [95] (S)	`ConfD-xfstests`
#6	Regression test suite for Ext4 utilities	`e2fsprogs/tests` [51] (S)	`ConfD-e2fsprogs`

Table 5: **Summary of ConfD Plugins.** 'Base Tool' means existing tools that have been integrated with ConfD through the corresponding plugins; 'R' means open-source Research prototype, 'S' means Standard test suites for file systems and utilities.

Plugin #1: Configuration Specification Checker. The specifications for the configurations of Linux file systems are maintained through the Linux man-pages project [84]. Unfortunately, due to a variety of reasons (e.g., constant system upgrades, feature additions, bug fixes), the specifications may become inaccurate easily, which may confuse end users and/or lead to configuration-induced failures [35, 64]. The `ConfD-specCk` plugin is designed to mitigate the problem. It parses the Linux man-pages related to the file system configurations (e.g., `mke2fs`, `mkfs.xfs`) and checks a subset of multilevel dependencies (Table 4) based on keywords. For example, `resize_inode` and `meta_bg` cannot be enabled together for `mke2fs` (i.e., CPD), so `meta_bg` should appear in the description of `resize_inode` with 'disable' (or similar keywords) and vice versa. Similarly, value ranges (i.e., SD) and other value dependencies (e.g., `cluster_size` needs to be 'equal' or 'greater' than `block_size`) should also be specified in the descriptions accordingly. Such dependencies from man-pages are stored in the JSON format for further comparison with the dependencies extracted from the source code by *ConfD-core* (§4.1). A mismatch implies a potential specification issue.

Plugin #2: Misconfiguration Handling Checker. A well designed file system should be able to handle wrong configurations from end users (either by mistake or by intention) gracefully. Failing to handle misconfigurations elegantly implies *misconfiguration vulnerabilities* that could hurt system reliability and/or security [13]. The `ConfD-handlingCk` plugin is designed to expose the potential issues in misconfiguration handling. Thanks to the built-in 'Violating Dependency' policy (§4.1.3), the plugin can directly leverage the invalid configuration states generated by ConfD which violate inherent configuration dependencies. It applies such automatically generated misconfigurations to drive the target file systems and utilities, and records the symptoms accordingly for postmoterm analysis.

Plugin #3: Dependency-aware Fault Injector. Fault injection techniques have been applied to improve both file systems and utilities [2, 15, 25, 28, 51]. By systematically generating corrupted file system states, they enable analyzing the robustness of FS ecosystems thoroughly. However, given the complexity of file system metadata, one open challenge is how to generate vulnerable states efficiently. To mitigate the challenge, we integrate one open-source fault injector `rfsck` [25] with ConfD through the `ConfD-rfsck` plugin. Instead of relying on the default configuration, `ConfD-rfsck` leverages dependency-guided configurations to generate input images to initiate the fault injection campaign. Since the input images are configured with dependent parameters identified by ConfD, they represent more complicated states that are more difficult to remain consistent under fault. Note that the plugin only needs to provide an FS image with a different configuration as input to `rfsck`. No modification to the source code of `rfsck` is required. As will be shown in §5.2, this simple strategy can help trigger vulnerabilities effectively.

Plugin #4: Dependency-aware FS Fuzzer. Fuzzing techniques have also been applied to improve the reliability of file systems [11, 29]. Nevertheless, fuzzing file systems is still challenging due to the lengthy state exploration time needed to exercise a practical file system under each configuration (e.g., it may take multiple weeks to trigger one bug [29]). In other words, the time penalty for exploring a less interesting configuration state is high. To mitigate the challenge, we integrate one open-source fuzzer `gt-hydra` [2] with ConfD through the `ConfD-gt-hydra` plugin. Similar to plugin #3, `ConfD-gt-hydra` leverages dependency-guided configurations generated by ConfD to create FS images with more complicated dependencies and thus more chances of vulnerability for fuzzing. The plugin only changes the configurations of the input images for `gt-hydra`; no modification to the source code of the base tool is needed.

Plugin #5 & #6: Dependency-aware Regression Test Suites. Besides research prototypes, there are standard regression test suites developed for file systems (e.g., `xfstests` [95] and `e2fsprogs/tests` [51]), which include carefully designed workloads and test oracles to ensure the quality of the target. Nevertheless, existing test suites only use a subset of configuration parameters and they are mostly dependency-agnostic. To address the limitation, we create two plugins: `ConfD-xfstests` and `ConfD-e2fsprogs`, for `xfstests` and `e2fsprogs/tests` respectively. The plugins scan the test scripts and automatically replace the built-in FS configurations of the test cases with the configuration states generated

[2] To avoid confusion, we use `gt-hydra` to refer to the Hydra fuzzing framework created by GaTech researchers [29], and use FB-HYDRA to refer to the Hydra configuration management framework created by Facebook [31].

Target FS Ecosystem	Self Dependency (SD)		Cross-Parameter Dep. (CPD)		Cross-Component Dep. (CCD)		All Level Combined	
	Extracted	FP	Extracted	FP	Extracted	FP	Extracted	FP
Ext4	17	0	48	1 (2.1%)	46	3 (6.5%)	111	4 (3.6%)
XFS	18	2 (11.1%)	10	3 (30.0%)	15	4 (26.7%)	43	9 (20.9%)
Total	35	2 (5.7%)	58	4 (6.9%)	61	7 (11.5%)	154	13 (8.4%)

Table 6: **Multilevel Configuration Dependencies Extracted by ConfD.** This table shows the numbers of multilevel dependencies extracted from Ext4 and XFS ecosystems automatically. 'FP' means False Positive rate.

Target FS Ecosystem	# of Uncorrectable Images Reported			
	`rfsck` (1)	`ConfD-rfsck` (25)		
Ext4	11	< 11 (4)	$= 11$ (4)	> 11 (17)

Table 7: **Comparison of Two FS Fault Injectors.** `rfsck` explores 1 default configuration state and reports 11 uncorrectable images. `ConfD-rfsck` explores 25 configuration states; it reports $>$ 11 uncorrectable images (i.e., better than `rfsck`) in 17 out of 25 configuration states.

ID	Symptom of Uncorrectable Corruption	Triggered?	
		`rfsck`	`ConfD-rfsck`
1	Unable to mount the FS	N	Y (6)
2	Invalid file data	N	Y (24)
3	Truncated file data	Y (11)	Y (250)
	Total	11	280

Table 8: **Comparison of Corruption Symptoms Triggered.** `ConfD-rfsck` triggered ('Y') more types of corruptions. The counts are in parentheses.

by ConfD. The two plugins use the 'Follow Dependency' policy of ConfD to drive the test cases deeply into the target functionalities without early termination due to superficial configuration errors. In doing so, we reuse the well designed test logic and enhance the test suites with dependency awareness. If any test case fails with the *valid* configurations provided by ConfD, the result is saved for postmortem analysis.

Note that ConfD plugins are not limited to the six above. By modularizing the core module of ConfD (Figure 3), we expect that other software may benefit from ConfD conveniently via plugins (see §6 for more discussion).

5 Experimental Results

In this section, we describe the experimental results of applying ConfD to analyze Ext4 and XFS. First (§5.1), we show that ConfD can extract 154 multilevel configuration dependencies from the target systems effectively with a low false positive rate (8.4%). Second (§5.2), we demonstrate that ConfD can help address configuration-related issues more effectively compared to existing dependency-agnostic solutions. Through the experiments, we have identified various configuration-related issues including 17 specification issues, 18 configuration handling issues, and 10 regression test failures induced by valid configurations.

5.1 Can ConfD extract multilevel dependencies?

Table 6 summarizes the multilevel configuration dependencies extracted by ConfD from Ext4 and XFS automatically. As shown in the table, we were able to extract 154 unique dependencies in total, including 35 Self Dependency (SD), 58 Cross-Parameter Dependency (CPD), and 61 Cross-Component Dependency (CCD). The multilevel dependencies have been observed on both Ext4 and XFS, which is consistent with our manual study (§3).

We manually examined all the 154 dependencies extracted by ConfD automatically and found that the overall false positive rate is 8.4% (13/154), which is similar to that of the previous work on analyzing configuration constraints in other software systems [13, 33]. Note that ConfD is designed to handle the unique configuration methods of FS ecosystems (§2 and §3.2) which is arguably more challenging to analyze compared to the targets of existing work.

5.2 Can ConfD help address configuration issues?

5.2.1 Dependency-agnostic vs. Dependency-guided

In this section, we compare the effectiveness of two open-source research prototypes (i.e., `rfsck` [25] and `gt-hydra` [29]) with and without ConfD support. We focus on the two research prototypes and the corresponding plugins for comparison because they provide quantitative metrics to measure the effectiveness straightforwardly. We defer the results of other plugins to the next section.

In the first experiment, we applied fault injectors `rfsck` and `ConfD-rfsck` to analyze Ext4 and its checker utility `e2fsck`. The fault injectors interrupt the checker operation and examine if the interrupted checker could lead to uncorrectable corruptions on the file system (i.e., cannot be fixed by another run of checker). They report the number of repaired FS images containing uncorrectable corruptions (i.e., "uncorrectable image"). Each uncorrectable image implies a vulnerability in the FS ecosystem that could lead to data loss [25].

The result of the experiment is summarized in Table 7. `rfsck` reports 11 uncorrectable images with the default configuration. `ConfD-rfsck` can explore different configuration states and we analyze the reports generated under 25 configuration states for comparison. In 4 out of the 25 states,

Target FS	# of Issues Reported (in two weeks)	
	`gt-hydra`	`ConfD-gt-hydra`
Ext4	1	17

Table 9: **Comparison of Two FS Fuzzers.** `ConfD-gt-hydra` reports more hangs given the same fuzzing time.

`ConfD-rfsck` generates less than 11 uncorrectable images; in 4 states, `ConfD-rfsck` generates the same amount of uncorrectable images (i.e., '= 11'); in the majority states (17), `ConfD-rfsck` generates more uncorrectable images (i.e., '> 11'), which suggests it is more effective in exposing potential vulnerabilities in the FS ecosystem.

Table 8 further compares the symptoms of uncorrectable corruptions triggered by `rfsck` and `ConfD-rfsck`. Overall, `ConfD-rfsck` triggers three different types of symptoms, while `rfsck` only triggers one symptom in our experiment. Since different symptoms typically imply different vulnerabilities in metadata protection and/or recovery in the FS ecosystem, the result also suggests that the dependency-guided configuration states used by `ConfD-rfsck` can help improve the effectiveness of `rfsck`.

In the second experiment, we applied `gt-hydra` and `ConfD-gt-hydra` to fuzz the Ext4 file system. The fuzzers systematically generate various inputs (i.e., FS metadata corruptions and system calls) to explore different code paths in the file system for triggering latent bugs [29]. We run each fuzzer continuously for two weeks. The fuzzers report the number of reliability issues detected on the target file system within the running period. The issues may include different types depending on the bug checkers used. We use the default SymC3 checker which can detect crash inconsistency bugs. Meanwhile, since the fuzzers are based on the AFL fuzzer [38], they also report crash and hang issues (detected by AFL) by default. Note that the only difference `ConfD-gt-hydra` introduces is the dependency-guided configurations, i.e., it does not change the test logic or criteria for reporting issues. Therefore, both the types of issues (e.g., 'crash', 'hang', 'crash inconsistency') and the number of issues reported can be used as the metric to evaluate effectiveness.

The result of the fuzzing experiment is summarized in Table 9. To make the comparison fair, we limit the two fuzzers to the same total execution time (i.e., two weeks each). We set the `ConfD-gt-hydra` to switch to a new dependency-guided configuration state every 12 hours, which leads to 28 critical configuration states being explored within two weeks. While each configuration in `ConfD-gt-hydra` is explored with only 1/28 of the time used by `gt-hydra` for its configuration, the overall result of `ConfD-gt-hydra` is better: `gt-hydra` only detects 1 issue on Ext4 by the end of the two week period, while `ConfD-gt-hydra` detects 17 issues in total. Interestingly, all issues reported in the experiment are 'hang'. This is expected because triggering more complicated semantic bugs may require multiple weeks.

ConfD Plugin (Type of Issue Reported)	# of Issue Reported		
	Ext4	XFS	Total
`ConfD-specCk` (undoc./wrong dep.)	13	4	17
`ConfD-handlingCk` (bad reaction)	13	5	18
`ConfD-xfstests` (test case failure)	5	4	9
`ConfD-e2fsprogs` (test case failure)	1	N/A	1
`ConfD-rfsck` (uncorrectable image)	280	–	280
`ConfD-gt-hydra` (hang)	17	–	17

Table 10: **Summary of Issues.** This table summarizes configuration-related issues observed via ConfD plugins.

Target FS Ecosystem	# of Undocumented/Wrong Dep.			Total
	SD	CPD	CCD	
Ext4	7	4	2	13
XFS	2	2	0	4
Total	9	6	2	17

Table 11: **Specification Issues.** This table summarizes the undocumented or wrong dependencies observed. 'SD', 'CPD', and 'CCD' are defined in Table 4.

In summary, the two sets of comparison experiments above show that ConfD can amplify the effectiveness of existing FS tools for identifying vulnerabilities quickly, which is particularly valuable for time-consuming methodologies like fault injection or fuzzing. Note that in all experiments, we do not randomly generate combinations of configurations. This is because a naive algorithm without any knowledge of inherent dependencies can easily lead to time-wasting configurations, as will be demonstrated further in §5.2.3.

5.2.2 Summary of Configuration Issues

Table 10 summarizes the configuration-related issues triggered by ConfD plugins in our experiments. Overall, we observed more than 300 issues of various types. The issues are diverse because the plugins are created for different purposes or based on different base tools (Table 5). Note that all the issues require dependency-guided configuration states generated by ConfD to manifest. In other words, continuously running the original research prototypes or standard test suites cannot expose the issues. Also, since we do not change the test logic of the base tools, the enhancement is purely contributed by the dependency information from ConfD. Since `ConfD-rfsck` and `ConfD-gt-hydra` have been discussed in §5.2.1, we focus on others below.

Table 11 summarizes the specification issues detected by `ConfD-specCk`. We have identified 17 inaccurate specification issues in total. The issues mainly manifest as undocumented critical dependencies or wrong dependencies, which may occur to both Ext4 and XFS and involve SD, CPD, and CCD. For example, there is a CPD extracted by ConfD which specifies that two parameters of `mke2fs` (i.e., `meta_bg` and `resize_inode`) cannot be used together, but this CPD is missing from the Linux man-pages. As another example, there

ID	Reaction	Description	Observed?
1	Early Termination	the utility program exits w/o pinpointing the configuration error	Y
2	Functional Failure	the utility fails functional testing w/o pinpointing the configuration error	Y
3	Silent Violation	the system changes input configurations to different values w/o notifying users	Y
4	Silent Ignorance	the system ignores input configurations	N
5	Crash/Hang	the system crashes or hangs	N
6	Partial Report	the utility partially identify the violated configuration dependencies	Y

Table 12: **Suboptimal Reaction of Configuration Dependency Violation.** This table summarizes the bad handling behaviors observed when the configuration dependencies are violated. The first five are based on the definitions from [13].

is a CCD which implies that `resize2fs` may not be used for Ext4 when the `bigalloc` feature is enabled through `mke2fs`. Violating the CCD may corrupt the file system, which is unfortunately not mentioned in the specification.

Table 12 summarizes the suboptimal handling of misconfigurations identified through `ConfD-handlingCk`. We follow the criteria in the literature [13]: when a misconfiguration occurs (i.e., a dependency is violated), the system should pinpoint either the offending parameter's name/value or its location information; failing to do so implies misconfiguration vulnerabilities. Specifically, there are six types of misconfiguration vulnerabilities based on different reactions, including 'Early Termination', 'Functional Failure', 'Silent Violation', 'Silent Ignorance', 'Crash/Hang', and 'Partial Report'. The first five types are based on the definitions from [13], while the last one is unique in our study because we consider more complicated multilevel dependencies.

As an example, the `mke2fs` parameter `-E encoding` enables the `casefold` feature and set the encoding in Ext4. But if the user tries to disable the `casefold` feature when using the `-E encoding`, instead of showing an error or warning, the utility enables the `casefold` feature silently without informing the user. We consider this as 'Silent Violation'.

When more than one dependency is violated, utilities often only show a partial message (i.e., 'Partial Report'). For example, the `mkfs.xfs` parameter `sunit` involves two dependencies: (1) it does not allow unit suffixes, and (2) it cannot be specified together with `su`. But when both dependencies are violated, the utility may only show one of the violations.

In total, we have observed 4 out of the 6 types of suboptimal reactions, which suggests that FS ecosystems are not immune from misconfiguration vulnerabilities reported in other practical systems. Note that `ConfD-handlingCk` leverages the static analysis of CONFD to violate specific dependencies carefully, which avoids many duplicate and valid configuration states for testing. This reduces the manual effort needed for the post-mortem analysis.

In terms of `ConfD-xfstests` and `ConfD-e2fsprogs`, we have observed 10 new test case failures which can be induced by *valid* configuration states generated by CONFD. For example, `ConfD-xfstests` triggers an Ext4 corruption when applying the online defragmentation tool `e4defrag` to the file system with the `bigalloc` feature enabled. Note that a FS test case may involve multiple utilities. Due to the complexity of the test case and the FS ecosystems, a test case may fail for various subtle reasons (e.g., timing at `mount`) in practice, which is time-consuming to diagnose even for developers [47]. In our experiments, we observed more than 10 newly failed test cases after changing with valid configurations. We only count the cases that we have manually verified and reproduced at the time of this writing. Also, since CONFD limits the change to the configuration states without modifying the test logic, it may help narrow down the root cause of a test case failure to the configuration-related code paths.

Framework	# of States	# of Duplicate	# of Invalid
FB-HYDRA	56,592	42,745 (75.5%)	15,146 (26.8%)
CONFD	30	0	0

Table 13: **Comparison of State Generation.**

5.2.3 State Generation: FB-HYDRA vs. CONFD

One unique feature of CONFD is it generates configuration states based on multilevel dependencies, which is critical for analyzing configuration issues given the huge configuration space. To the best of our knowledge, the FB-HYDRA configuration management framework [31] provides the most similar functionality. It includes a "multirun" feature to support running an application with different configurations in different runs automatically. We compare the configuration states generated by FB-HYDRA and CONFD in this section to demonstrate the difference.

Table 13 shows the states generated by FB-HYDRA and CONFD for the same program (i.e., `mke2fs`) given the same set of configuration parameters. For simplicity, we only use 10 parameters with limited ranges in this experiment. As shown in the table, even with this simplified scenario, FB-HYDRA may generate many duplicated or invalid states. This is because FB-HYDRA is agnostic to the configuration constraints of `mke2fs`. Specifically, FB-HYDRA maintains a list for each parameter and its possible values. It passes all lists to the `itertools.product()` function which returns the cartesian product of the values in the lists. Such a simple algorithm is incompatible with FS ecosystems. For example, '`mke2fs -b 1024 -C 2048`' and '`mke2fs -C 2048 -b 1024`' are equivalent in practice but are considered as different in FB-HYDRA. Moreover, invalid states can easily be created by FB-HYDRA due to violation of dependencies, which suggests the importance of dependency analysis.

Note that FB-HYDRA has other features that CONFD does not have (e.g., Python library support). Also, FB-HYDRA supports plugins which makes it possible to benefit from the state generation of CONFD (see §6 for more discussion). Therefore, we view FB-HYDRA and CONFD as complementary.

6 Limitations & Future Work

No study or tool is perfect, and our work is no exception. We discuss the limitations of our work as well as a few promising extensions in this section.

Limitations of the multilevel taxonomy. As briefly mentioned in §3.1, the multilevel configuration dependencies should be interpreted with the study methodology in mind, because they are derived from an incomplete set of configuration-related issues from two FS ecosystems. It is likely that there are more complex dependencies in FS ecosystems, which deserves further investigation.

Limitations of the CONFD framework. The current prototype requires a few user inputs (§4.1.4) to guide the automated dependency analysis, which we hope to reduce through more sophisticated state analysis. Also, CONFD can only handle a subset of LLVM IR for taint analysis and it only considers two parameters at a time for CPD and CCD, which may lead to incomplete dependency or false positives. We hope to improve these through more advanced software engineering efforts in the future, which will likely improve the effectiveness further. Similarly, there are limitations in plugins. For example, `ConfD-handlingCk` only induces at most two violations for one configuration state for simplicity; there may be more issues if we consider more than two. `ConfD-xfstests` only transforms a subset of the test suite due to the irregular configuration handling. Despite the limitations, CONFD has been effective in analyzing dependencies and exposing configuration-related issues in our experiments, so we believe that it will be valuable to the community.

Integration with other file systems and tools. As mentioned in Table 1, many file systems can be configured through different utilities, which could potentially benefit from the multilevel dependency analysis of CONFD after minor customization (e.g., providing FS-specific inputs in JSON format 4.1.4). Also, CONFD is complementary to other modern tools besides the base tools used in current plugins. For example, FB-HYDRA [31] uses YAML files to store configurations which is compatible with the JSON files used by CONFD. Moreover, it supports a set of plugins called "Sweepers" to manipulate the selection of parameters. The dependency-based state generation in CONFD could be implemented as one special "Sweeper" for FB-HYDRA [31]. Similarly, the configurations generated by CONFD could potentially be integrated into CI/CD frameworks [62] to enable pipelined configuration-oriented testing and deployment. We leave the integration with other file systems (e.g., ZFS) and tools as future work.

Support for other software. Configuration dependency is not limited to file systems. For example, NDCTL [74] is a utility to configure the `libnvdimm` subsystem in Linux. We expect that adding NDCTL to the dependency analysis will likely help address NVM-specific configuration issues more effectively. Also, researchers have observed functionality or correctness dependencies between local file systems and other software (e.g., databases [16], distributed storage systems [20, 23, 34, 37]), many of which are also related to configurations. The dependencies studied in this work may serve as a foundation for investigating such configuration-related issues beyond file systems. Also, since LLVM supports compiling a wide set of languages (e.g., C++, Rust, Swift) to IR through various frontends [85], the core analysis of CONFD is expected to be applicable to software written in other languages as well.

Better configuration design. An alternate perspective of the configuration challenge studied in this work is that we may have too many parameters today. One might argue that it is perhaps better to reduce the parameters to avoid vulnerabilities or confusions, instead of adding new configurations for more features. Also, one might suggest that (in theory) we can implement every utility functionality in the file system itself to avoid tricky cross-component configuration dependencies. Essentially, these are trade-offs of the file system and configuration design that deserve more investigation from the community. We hope that by studying real-world configuration issues and releasing the CONFD prototype, our work can help identify problematic configuration parameters and further help with the reduction of such parameters to improve the configuration design in general.

7 Conclusion

We have presented a study on 78 real-world configuration issues and built an extensible framework called CONFD for addressing various configuration issues. Our experiments on Ext4 and XFS demonstrate that CONFD can help address configuration issues effectively by leveraging configuration dependencies. In the future, we would like to improve CONFD further and investigate other systems as discussed in §6. We hope that CONFD can facilitate follow-up research on addressing the increasing challenge of configurations in general.

Acknowledgments

We thank Ethan Miller, our shepherd, and the anonymous reviewers for their tremendous feedback. We also thank Runzhou Han and Wei Xu for their help on validating and reproducing a few bug cases. This work was supported in part by National Science Foundation (NSF) under grants CNS-1855565, CCF-1853714, CCF-1910747 and CNS-1943204. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Sanjay Ghemawat et al. “The Google file system”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 2003.

[2] Vijayan Prabhakaran et al. “IRON File Systems”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*. 2005.

[3] Haryadi S. Gunawi et al. “SQCK: A Declarative File System Checker”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2008.

[4] Lorenzo Keller et al. “ConfErr: A tool for assessing resilience to human configuration errors”. In: *Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks (DSN)*. 2008.

[5] Shan Lu et al. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008.

[6] Huning Dai et al. “CONFU: Configuration Fuzzing Testing Framework for Software Vulnerability Detection”. In: *Int. J. Secur. Softw. Eng.)* 1.3 (2010).

[7] Ariel Rabkin et al. “Static extraction of program configuration options”. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 2011.

[8] Zuoning Yin et al. “An Empirical Study on Configuration Errors in Commercial and Open Source Systems”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[9] Edmund Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification*. Jan. 2012.

[10] Daniel Fryer et al. “Recon: Verifying File System Consistency at Runtime”. In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. 2012.

[11] Christoph Albrecht et al. “Janus: Optimal Flash Provisioning for Cloud Storage Workloads”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*. 2013.

[12] Lanyue Lu et al. “A Study of Linux File System Evolution”. In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. 2013.

[13] Tianyin Xu et al. “Do Not Blame Users for Misconfigurations”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[14] Dongpu Jin et al. “Configurations Everywhere: Implications for Testing and Debugging in Practice”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 2014.

[15] Thanumalayan Sankaranarayana Pillai et al. “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014.

[16] Mai Zheng et al. “Torturing Databases for Fun and Profit”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014.

[17] Changwoo Min et al. “Cross-Checking Semantic Correctness: The Case of Finding File System Bugs”. In: *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 2015.

[18] James Bornholt et al. “Specifying and checking file system crash-consistency models”. In: *SIGPLAN Not.* 51.4 (2016).

[19] Scott Klemmer Tianyin Xu Vineet Pandey. “An HCI View of Configuration Problems”. In: *arXiv*. 2016.

[20] Aishwarya Ganesan et al. “Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions”. In: *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)*. 2017.

[21] Aravind Machiry et al. “DR. Checker: A Soundy Analysis for Linux Kernel Drivers”. In: *Proceedings of the 26th USENIX Conference on Security Symposium (SEC)*. 2017.

[22] Salman Niazi et al. “HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases”. In: *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 2017.

[23] Jinrui Cao et al. “PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems”. In: *Proceedings of the 2018 International Conference on Supercomputing (ICS)*. 2018.

[24] Mikaela Cashman et al. “Navigating the Maze: The Impact of Configurability in Bioinformatics Software”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 2018.

[25] Om Rameshwar Gatla et al. “Towards Robust File System Checkers”. In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 2018.

[26] Om Rameshwar Gatla et al. “Towards Robust File System Checkers”. In: *ACM Transactions on Storage (TOS)* 14.4 (2018).

[27] Kuei Sun et al. “Spiffy: Enabling File-System Aware Storage Applications”. In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 2018.

[28] Shehbaz Jaffer et al. “Evaluating File System Reliability on Solid State Drives”. In: *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 2019.

[29] Seulbae Kim et al. “Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019.

[30] Wen Xu et al. “Fuzzing file systems via two-dimensional input space exploration”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[31] Omry Yadan. *Hydra - A framework for elegantly configuring complex applications*. Github. 2019. URL: https://github.com/facebookresearch/hydra.

[32] Zhen Cao et al. “Carver: Finding Important Parameters for Storage System Tuning”. In: *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 2020.

[33] Qingrong Chen et al. “Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020.

[34] Runzhou Han et al. “Fingerprinting the Checker Policies of Parallel File Systems”. In: *IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*. 2020.

[35] Xudong Sun et al. “Testing Configuration Changes in Context to Prevent Production Failures”. In: *PProceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[36] Duo Zhang et al. “A Study of Persistent Memory Bugs in the Linux Kernel”. In: *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR)*. 2021.

[37] Runzhou Han et al. “A Study of Failure Recovery and Logging of High-Performance Parallel File Systems”. In: *ACM Transactions on Storage (TOS)* 18.2 (2022).

[38] *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/.

[39] *Apache Common Configuraitons*. https://commons.apache.org/proper/commons-configuration/userguide/upgradeto2_0.html.

[40] *Apache Hadoop*. https://hadoop.apache.org/.

[41] *btrfs-balance*. https://man7.org/linux/man-pages/man8/btrfs-balance.8.html.

[42] *btrfs-check*. https://man7.org/linux/man-pages/man8/btrfs-check.8.html.

[43] *btrfs-scrub*. https://man7.org/linux/man-pages/man8/btrfs-scrub.8.html.

[44] *CFEngine*. https://github.com/cfengine/core.

[45] *chkdsk*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/chkdsk.

[46] *defrag*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/defrag.

[47] *Discussion between Ext4 developers and newbie on finding bugs on Ext4*. https://lore.kernel.org/linux-ext4/Yx9fUHiiZaKXeLUw@mit.edu/.

[48] *disk utility*. https://www.dssw.co.uk/reference/diskutil.html.

[49] *dump*. https://www.freebsd.org/cgi/man.cgi?query=dump&apropos=0&sektion=8&manpath=FreeBSD+13.1-RELEASE+and+Ports&arch=default&format=html.

[50] *e2fsck*. https://linux.die.net/man/8/e2fsck.

[51] *e2fsprogs-test*. https://sourceforge.net/projects/e2fsprogs/files/e2fsprogs-TEST/.

[52] *E2fsprogs: Ext2/3/4 Filesystem Utilities*. https://e2fsprogs.sourceforge.net/.

[53] *e4defrag*. https://man7.org/linux/man-pages/man8/e4defrag.8.html.

[54] *Ext4*. https://ext4.wiki.kernel.org/index.php/Main_Page.

[55] *format*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/format.

[56] *fsck*. https://man.minix3.org/cgi-bin/man.cgi?query=fsck.

[57] *fsck_apfs*. https://www.manpagez.com/man/8/fsck_apfs/.

[58] *fsck_ufs*. https://www.freebsd.org/cgi/man.cgi?query=fsck_ufs.

[59] *growfs*. https://www.freebsd.org/cgi/man.cgi?growfs(8).

[60] *HotHardware: Windows 10 20H2 Update Reportedly Damages SSD File Systems If You Run ChkDsk*. https://hothardware.com/news/windows-10-20h2-update-damages-ssd-file-systems-chkdsk.

[61] *JavaScript Object Notation*. https://www.json.org/json-en.html.

[62] *Jenkins*. https://www.jenkins.io/.

[63] *Lustre*. https://www.lustre.org/.

[64] *Maintaining Linux man-pages*. https://www.kernel.org/doc/man-pages/maintaining.html.

[65] Wilcox Matthew. *DAX: Page cache bypass for filesystems on memory storage*. https://lwn.net/Articles/618064/.

[66] *mke2fs*. https://linux.die.net/man/8/mke2fs.

[67] *mkfs.btrfs*. https://man7.org/linux/man-pages/man8/mkfs.btrfs.8.html.

[68] *mkfs.xfs*. https://man7.org/linux/man-pages/man8/mkfs.xfs.8.html.

[69] *mount*. https://man7.org/linux/man-pages/man8/mount.8.html.

[70] *mount*. https://www.freebsd.org/cgi/man.cgi?query=mount.

[71] *mount_apfs*. https://www.manpagez.com/man/8/mount_apfs/.

[72] *mountvol*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/mountvol.

[73] *MySQL NDB Cluster*. https://en.wikipedia.org/wiki/NDB_Cluster.

[74] *NDCTL*. https://github.com/pmem/ndctl.

[75] *newfs*. https://www.freebsd.org/cgi/man.cgi?newfs(8).

[76] *NTFS*. https://www.ntfs.com/index.html.

[77] *OpenStack*. https://www.openstack.org/.

[78] *OpenStack Swift*. https://docs.openstack.org/swift/latest/.

[79] *resize2fs*. https://linux.die.net/man/8/resize2fs.

[80] *restore*. https://www.freebsd.org/cgi/man.cgi?query=restore.

[81] *shrink*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shrink.

[82] *Soot - A framework for analyzing and transforming Java and Android*. http://soot-oss.github.io/soot/.

[83] *The First and Only Adaptive Computational Storage Platform*. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[84] *The Linux man-pages Project*. https://www.kernel.org/doc/man-pages/maintaining.html.

[85] *The LLVM Compiler Infrastructure*. https://llvm.org/.

[86] *Trim*. https://en.wikipedia.org/wiki/Trim_(computing).

[87] *Windows 10 2004/20H2: Microsoft fixes chkdsk issue in update KB4592438*. https://borncity.com/win/2020/12/21/windows-10-2004-20h2-microsoft-fixes-chkdsk-issue-in-update-kb4592438/.

[88] *Windows 10 20H2: ChkDsk damages file system on SSDs with Update KB4592438 installed*. https://borncity.com/win/2020/12/18/windows-10-20h2-chkdsk-damages-file-system-on-ssds-with-update-kb4592438-installed/.

[89] *XFS*. https://xfs.wiki.kernel.org/.

[90] *xfs_admin*. https://man7.org/linux/man-pages/man8/xfs_admin.8.html.

[91] *xfs_fsr*. https://man7.org/linux/man-pages/man8/xfs_fsr.8.html.

[92] *xfs_growfs*. https://man7.org/linux/man-pages/man8/xfs_growfs.8.html.

[93] *xfs_repair*. https://man7.org/linux/man-pages/man8/xfs_repair.8.html.

[94] *xfsprogs*. https://www.linuxfromscratch.org/blfs/view/svn/postlfs/xfsprogs.html.

[95] *xfstests*. https://github.com/kdave/xfstests.

[96] *zfs-create*. https://www.freebsd.org/cgi/man.cgi?query=zfs-create.

[97] *zfs-destroy*. https://www.freebsd.org/cgi/man.cgi?query=zfs-destroy.

[98] *zfs-mount*. https://www.freebsd.org/cgi/man.cgi?query=zfs-mount.

[99] *zfs-rollback*. https://www.freebsd.org/cgi/man.cgi?query=zfs-rollback.

[100] *zfs-set*. https://www.freebsd.org/cgi/man.cgi?query=zfs-set.