Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems

Qingrong Chen*, Teng Wang†, Owolabi Legunsen‡, Shanshan Li†, and Tianyin Xu*
*University of Illinois at Urbana-Champaign, Urbana, IL, USA
†National University of Defense Technology, Changsha, Hunan, China
‡Cornell University, Ithaca, NY, USA
{qc16,tyxu}@illinois.edu,{wangteng13,shanshanli}@nudt.edu.cn,legunsen@cornell.edu

ABSTRACT

A large percentage of real-world software configuration issues, such as misconfigurations, involve multiple interdependent configuration parameters. However, existing techniques and tools either do not consider dependencies among configuration parameters—termed *configuration dependencies*—or rely on one or two dependency types and code patterns as input. Without rigorous understanding of configuration dependencies, it is hard to deal with many resulting configuration issues.

This paper presents our study of software configuration dependencies in 16 widely-used cloud and datacenter systems, including dependencies within and across software components. To understand types of configuration dependencies, we conduct an exhaustive search of descriptions in structured configuration metadata and unstructured user manuals. We find and manually analyze 521 configuration dependencies. We define five types of configuration dependencies and identify their common code patterns. We report on consequences of not satisfying these dependencies and current software engineering practices for handling the consequences.

We mechanize the knowledge gained from our study in a tool, cDep, which detects configuration dependencies. cDep automatically discovers five types of configuration dependencies from bytecode using static program analysis. We apply cDep to the eight Java and Scala software systems in our study. cDep finds 87.9% (275/313) of the related subset of dependencies from our study. cDep also finds 448 previously undocumented dependencies, with a 6.0% average false positive rate. Overall, our results show that configuration dependencies are more prevalent and diverse than previously reported and should henceforth be considered a first-class issue in software configuration engineering.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; • **Computer systems organization → Maintainability and maintenance**.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409727

KEYWORDS

Configuration, dependency, cloud systems, datacenter systems

ACM Reference Format:
Qingrong Chen*, Teng Wang†, Owolabi Legunsen‡, Shanshan Li†, and Tianyin Xu*. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409727

1 INTRODUCTION

1.1 Motivation

Software misconfigurations are among the major causes of failures and performance issues in today's large-scale software systems that are deployed in cloud and data centers [33, 36, 47, 51, 54, 58, 61, 67, 89, 92]. For example, misconfigurations are reported as the second largest cause of service-level incidents in one of Google's main production services [36]; meanwhile, misconfigurations contribute to 16% of service-level incidents [79] at Facebook and are considered a key reliability challenge at Facebook scale [58].

Besides the prevalent and severe misconfigurations, users' configuration issues (e.g., difficulties in understanding configurations) also result in high support costs [34, 35, 87, 92, 96]. It has been reported that configuration issues are the dominant source of support costs incurred by cloud and datacenter software vendors [67, 92].

Among misconfigurations that cause real-world problems, 23.4%–61.2% involve more than one configuration parameter [92]. Further, in the cases with multiple parameters, the configuration parameters have *dependencies*—the correctness and effects of one parameter's value *depends on* other parameter values. In other words, the dependent configuration parameters should be considered together: setting one of them could affect the others.

Dependencies among multiple configuration parameters have been identified as a key source in complexity and error-proneness of software configurations [90]. System users face not only the enormous configuration space of very many parameters, but they also have to understand the dependencies. Note that exhaustively enumerating all possible dependencies leads to a combinatorial explosion. To make matters worse, configuration dependencies could also cross component boundaries—a parameter defined in one software component could depend on a parameter defined in a different component (or even in a different project). As we show (§4), inter-component configuration dependencies are not rare.

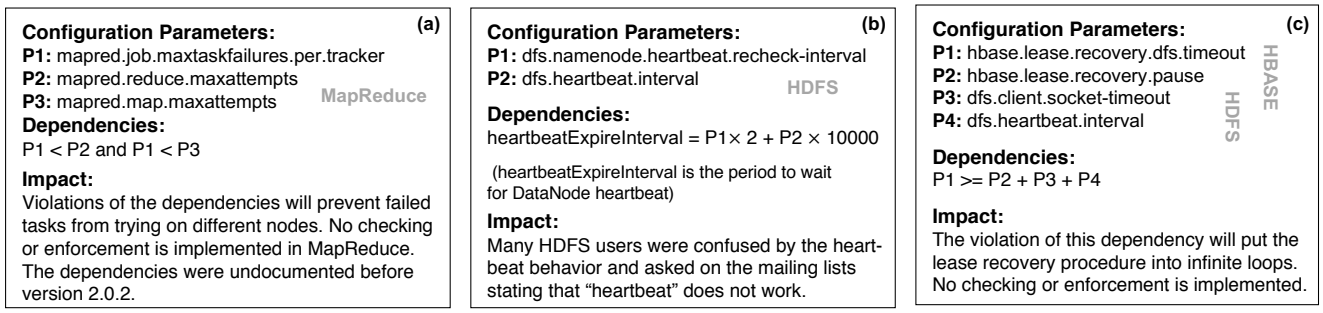

Figure 1: Examples of problematic configuration dependencies from cloud and datacenter software projects and their impact: (a) MapReduce; (b) HDFS, and (c) HBASE and HDFS. All these dependencies have caused real-world issues [1–4].

Taming software configuration dependency through configuration engineering and/or tooling is currently limited because the understanding of real-world dependencies is still preliminary. To make progress, a comprehensive study of configuration dependencies is needed. Better understanding would significantly benefit existing configuration tooling (e.g., for misconfiguration detection and diagnosis), reliability engineering (e.g., configuration correctness rule engineering [37, 49, 79]), configuration testing [78, 84], and customer support and documentation [86, 90, 91].

A few misconfiguration detection and diagnosis techniques consider configuration dependencies. However, all those techniques rely on *a priori* knowledge of only one or two dependency types and/or their code patterns as inputs. Tools that implement those techniques only cover a subset of dependencies, and also overlook several common and important dependency types and the corresponding code patterns. A detailed comparison is in §8.

1.2 Contributions

This paper makes two main contributions. First, we make the first attempt (to the best of our knowledge) to systematically study software configuration dependencies in modern cloud and datacenter software, for both intra- and inter-component dependencies. In particular, we comprehensively study configuration dependencies in 16 widely-used software projects across two different cloud and datacenter software stacks: the Hadoop-based data analytics stack and the OpenStack cloud computing infrastructure. We exhaustively search the configuration dependency information described in configuration metadata and manual pages of these projects, and identify the types of configuration dependencies that exist. In total, we discover 521 configuration dependencies, including 424 intra-component dependencies and 97 inter-component dependencies. We manually analyze each of the 521 configuration dependencies in depth, including their source code patterns, potential impact on the system when not satisfied, and existing engineering practices of dependency checking, violation handling and logging.

Based on our study, we define and formalize five types of configuration dependencies with the common code patterns that they manifest. These code patterns can be used to *automatically* discover configuration dependencies from code. Our study also reveals a number of missing opportunities in software configuration design and implementation for improving software reliability and usability.

Second, to discover configuration dependencies, we present a tool named cDep that mechanizes our understanding. cDep analyzes Java bytecode of the software programs of interest and automatically identifies specific types of dependencies and the interdependent parameters. cDep uses a novel and intuitive idea to discover configuration dependencies. cDep first "colors" program variables that store values of different configuration parameters based on static taint analysis—variables associated with different parameters have different colors; a variable derived from multiple parameters could have multiple colors. Then, cDep analyzes the dependencies between the colored variables based on the source code patterns from our study. cDep shows that it is feasible to effectively discover various types of configuration dependencies both within and across software components without the need to exhaustively evaluate all possible combinations.

We implement cDep on top of the Soot compiler framework [32]. We apply cDep to the eight Java and Scala projects in the Hadoop stack from our study. cDep finds 87.9% (275/313) of the configuration dependencies in our manually curated dataset from our study. cDep also finds 448 previously undocumented dependencies and incurs a false-positive rate of 6.0%. Running cDep on the Hadoop-based stack of eight large software systems takes no more than 160 minutes.

Overall, our results show that software configuration dependencies are more common and diverse than previously reported and should henceforth be considered a first-class issue in software configuration design and implementation, in tooling for misconfiguration detection and troubleshooting, as well as in configuration-aware testing and verification.

We have made the datasets and cDep publicly available at:

https://github.com/xlab-uiuc/cdep-fse

2 BACKGROUND

We show motivating examples of problems due to configuration dependencies, describe configurations and their usage model, and formally define configuration dependencies.

2.1 Motivating Examples

Figure 1 shows three real-world examples of configuration dependencies within and across the widely-used software projects that we studied. We can see that software configuration dependencies could be complex and even across software components. Figure 1 also

shows that failures to understand or satisfy configuration dependencies can have negative impacts. In fact, all three dependencies in Figure 1 have had significant implications on system reliability, and have caused real-world issues in the past. In particular, in Figures 1(a) and (c), if the dependencies are not satisfied, the failures occur during system recovery and caused catastrophic failures [94]. Additionally, the configuration dependencies were not always well documented—Figure 1(a)—and have repeatedly led to bad issues experienced by many different users—Figure 1(b). In Figure 1(c), the configuration dependency includes four configuration parameters across two different components—HDFS and HBase—from two separate software projects.

2.2 Configurations and Their Usage

A configuration is a mapping from a parameter to its value. Configurations allow customization of system behavior without making changes to the code. We assume the following model of how configurations are used in programs.

Loading. Configurations are loaded by reading from an external file or database and storing parameter values in program variables. Mature projects have well-defined application programming interfaces (APIs) for loading configurations [38, 44, 56, 57, 65, 66, 87, 89]. All projects evaluated in this paper have such APIs. Hadoop projects load configurations using *getter methods* that take a parameter and return a value (e.g., `getInt`, `getString`), declared in wrapper classes for `java.util.Properties` or `apache.commons.Configuration`. OpenStack projects use the `configparser` API, a part of the standard Python library which provides getter methods (e.g., `getint` and `getboolean`). We found that tracking usages of getter methods is effective for finding where parameter values are loaded.

Propagation and transformation. Once loaded, parameter values may be propagated along a program's data-flow paths using assignment statements and may be transformed using arithmetic or string operations. Propagation is commonly *inter-procedural* through arguments and return values or through message passing with sockets or Remote Procedure Calls (RPCs).

Usage. Eventually, parameter values are used in statements that change program behavior, e.g., branch conditions or system calls.

This model of configuration usage is the basis of our static analysis for discovering configuration dependencies: it reasons about interactions of program variables that store parameter values (§6).

2.3 Configuration Dependencies

Conceptually, a configuration dependency is either (1) *functional* if a parameter value is influenced by other parameter values, or (2) *behavioral* if a set of parameter values combine to influence a particular system behavior.

We define a *functional configuration dependency* as a pair, (M, f). Let P be the set of all parameters. M maps a parameter $p \in P$ to a non-empty set of parameters $Q \subseteq P$ if the value or scope of p depends on a function f of the value of parameters in Q. f is the function that computes the value of p from the values of parameters in Q. For example, let $Q = \{q_1, ..., q_n\}$. Then, $(p \mapsto \{q_1, ..., q_n\}, f)$ is a configuration dependency if the value or scope of $c(p)$ is determined by $f(c(q_1), ..., c(q_n))$, where c is a getter method. We put the functional configuration dependencies in our study into four categories based on what f computes (§4.1).

A *behavioral configuration dependency* is a function, $g : R \rightarrow \{\texttt{true}, \texttt{false}\}$ which returns *true* if there is a method in the program that takes the set of values of parameters in $R \subseteq P$, i.e., $\{c(r_1), ..., c(r_n)\}$ as arguments and can return a non-zero exit code, and `false` otherwise. For example, let $R = \{\texttt{ip.address}, \texttt{port.number}\}$, and let `connect(a,b)` be a method in the program that creates a network connection at an IP address `a` on port `b`. Then $g(R) = \texttt{true}$ is a behavioral configuration dependency, because the system can fail if elements in R are misconfigured, e.g., if the IP address does not allow connections on the specified port number.

We also categorize configuration dependencies based on where parameters are defined, essential for analyzing software stacks with multiple components. A functional configuration dependency is *intra-component* if all parameters in $\{p\} \cup Q$ are defined in the same component and *inter-component* if $x, y \in (\{p\} \cup Q)$ such that x and y are not defined in the same component. Similarly, a behavioral dependency is intra-component if all parameters in R are defined in the same component, and inter-component otherwise.

3 STUDY METHODOLOGY

The key challenge in studying software configuration dependencies lies in the fact that dependency information is neither usually explicitly specified in code nor documented elsewhere. As the first step towards a comprehensive understanding, we manually collect a large dataset of configuration dependencies both within the same component (i.e., intra-component dependencies) and across inter-related components (i.e., inter-component dependencies).

In this section, we describe our methodology for collecting configuration dependency information and for validating and analyzing the collected data.

3.1 Software Systems Studied

To collect both intra- and inter-component configuration dependencies, we studied software systems in two widely-used cloud and datacenter stacks: the Hadoop-based data analytics stack and OpenStack for cloud computing. Both stacks contain a number of independent but inter-related open-source software systems. The Hadoop stack includes 17 components for data processing and analytics, as well as underlying services for cluster management, scheduling, storage, coordination, etc. Similarly, OpenStack consists of 33 components for computing, storage, networking, imaging, etc., which can be used for building cloud computing platforms.

Table 1 gives a short description of the 16 components that we studied: eight components from Hadoop (Hadoop Common [14], HDFS [16], YARN [28], HBASE [13], Alluxio [9], ZooKeeper [29], MapReduce [20], and Spark [25]) and eight components from OpenStack (Nova [23], Swift [26], Neuron [21], Keystone [18], Glance [12], Placement [24], Ironic [17], and Cinder [10]). Each component is a stand-alone project but is typically used with other components to compose large-scale distributed systems. We chose these 16 projects because they are widely-used and studied; their configuration design and implementation represents the state-of-the-art in modern cloud systems, and each one exposes many configuration parameters, as shown in Table 1.

	Project	Lang.	Desc.	LOC	# Param.
Hadoop	HCommon [14]	Java	Hadoop core lib/runtime	268K	320
	HDFS [16]	Java	Distributed file system	644K	431
	Yarn [28]	Java	Resource management	639K	397
	HBase [13]	Java	Distributed database	755K	202
	Alluxio [9]	Java	In-memory storage	459K	332
	ZooKeeper [29]	Java	Distributed coordination	105K	51
	MapReduce [20]	Java	Data processing	220K	202
	Spark [25]	Scala	ML and data processing	586K	348
OpenStack	Nova [23]	Python	Compute service	365K	708
	Swift [26]	Python	Object storage	216K	405
	Neutron [21]	Python	Networking service	223K	222
	Keystone [18]	Python	Authentication	105K	202
	Glance [12]	Python	Image management	62K	193
	Placement [24]	Python	Resource tracking	15K	22
	Ironic [17]	Python	Machine provisioning	123K	509
	Cinder [10]	Python	Block storage	364K	769

Table 1: Studied software systems and their descriptions.

3.2 Data Collection and Analysis

Ideally, configuration dependencies would be collected automatically, e.g., by using program analysis. However, that was difficult for us because there was no prior study of the types of configuration dependencies. Therefore, as described in §3.2.1, we manually collected configuration dependencies based on two text-based data sources (henceforth, text sources) where dependency information is sometimes documented: *configuration metadata* (e.g., in XML based default configuration files) and *user manuals*. While text sources do not document the complete set of configuration dependencies, they provide a starting point.

We are aware that user manuals and other documents often miss important information [50, 53, 66, 73, 86]. In fact, our automated tool, cDep, finds many *undocumented* configuration dependencies (§6). We are also aware that text sources could be outdated or even incorrect [50, 53, 66, 73, 86]. So, we do not treat the dependencies that we collect from text sources as ground truths. *Rather, we manually validated every collected configuration dependency by understanding how the dependency occurs in the code* (§3.2.2).

3.2.1 **Collecting Configuration Dependencies.** We describe our heuristics for exhaustively searching for potential configuration dependencies in the two text sources. We prioritized completeness over precision—our heuristics-based text analysis is effective in discovering configuration dependencies but also introduces false positives. False positives are acceptable at this stage; all collected data are subsequently manually inspected and validated. Our data collection does not differentiate intra- vs. inter-component dependencies. We identify any interdependent configuration parameters which could come from one or multiple components.

Collecting Potential Configuration Dependencies from Structured Configuration Metadata. All 16 software systems that we studied manage structured descriptions and other metadata about configuration parameters, which are organized in different forms, e.g., manual entries [8, 22, 30, 31] and default (XML) configuration files [11, 15, 19, 27]. Ideally, the description of a configuration parameter should mention its dependencies on other parameters (if any) but we rarely found such configuration dependency information in these structured configuration metadata.

We use the following heuristic to search for potential configuration dependencies: if the description of one parameter mentions another parameter, there is a likely dependency between both parameters. We implement this heuristic by searching for other parameters in the description of each parameter. Note that the search is not limited to strict string matching on parameter names; we implement fuzzy search using a series of natural language processing techniques, including tokenization, lowercase and camel-case filtering, and stemming. As previously pointed out [52, 88, 97], textual descriptions may not contain the exact strings of parameter names, but may contain similar text that describes parameters.

Collecting Potential Configuration Dependencies from Unstructured Manual Pages. Configuration dependencies are sometimes also described in unstructured manual pages (e.g., [5–7]). We use the following heuristic to identify potential configuration dependencies from unstructured texts: if two parameters are mentioned in the same paragraph, they may be dependent and we liberally track them as such. We record the paragraph and the manual page for further validation. Note that for manual pages, we search for exact parameter names.

3.2.2 **Validation and Analysis.** We validate each potential configuration dependency by inspecting each portion of text that contains a likely configuration dependency. We filter out any false positives that we encounter. Each case is inspected by two inspectors (co-authors of this paper). One inspector first manually examined each dependency in detail, with the goal to answer these questions: (1) What are the dependent parameters? (2) Is it an intra- or inter-component dependency? (3) How are these parameters dependent? The second inspector then manually verified all the results from the first inspector. In the end, we had 521 dependencies and categorized them by the types in §4. Two inspectors spent six months validating. For each of the 521 validated configuration dependencies, we further analyze the source code to answer three other questions: (4) What are the code patterns exhibited by different dependency types? (§4.1, §4.2) (5) How are configuration dependency violations checked in source code? (§5.1) (6) How are detected violations of configuration dependencies handled in source code? (§5.2)

4 CONFIGURATION DEPENDENCY TYPES

We define four types of functional configuration dependencies that we found, provide examples, and describe commonly-occurring code patterns. We neither imposed a taxonomy *ex ante* nor defined types admissible by the definition in §2.3 but which do not occur in our data set. We also provide more details and examples of behavioral configuration dependencies. Lastly, we describe the results of categorizing the configuration dependencies in our data set according to the types described in this section.

4.1 Types of Functional Dependencies

Recall from §2.3 that a *functional configuration dependency* is one in which a parameter value is influenced by other parameter values, defined as a pair (M, f). M maps a dependent parameter p to the set of parameters $Q = \{q_1, ..., q_n\}$ such that the scope or value of $c(p)$

is determined by $f(c(q_1), ..., c(q_n))$, where c looks up parameters and returns values. The type of M is the same in all functional configuration dependency types defined below; f varies. For brevity, we will sometimes write Q for $[c(q_1), ..., c(q_n)]$, the list of values of the parameters in Q. Although Q has one element in all but four of 521 configuration dependencies that we found, below we give general definitions in which Q is a multi-element set.

4.1.1 **Control Dependency.** In a control dependency, whether a dependent parameter p value can be used or not depends on the value of other parameters—$f(Q)$ determines whether p is in scope.

Example. The most common form of control dependency that we found is that $\{q_1, ..., q_n\}$ enables or disables the execution of the only parts of code where p is used. That is, $c(p)$ is used only when $c(q_1) \wedge ... \wedge c(q_n)$ is `true`. In a concrete example from HDFS, p = `rpc.metrics.percentiles.intervals` and Q = {`rpc.metrics.quantile.enable`}. Q controls whether to measure percentile latency as a RPC metric, p specifies percentiles to measure.

Code Patterns. Essentially, a control dependency occurs when control flows to all uses of $c(p)$ are guarded by Q. We found two control dependency code patterns: (1) *Branch condition.* Branching depends on Q and the dependent parameter value $c(p)$ is used in only one branch. The following code snippet shows the control dependency of the example described above, in which the value of `rpc.metrics.quantile.enable` (in `rpcQuantileEnable`) controls the use of `rpc.metrics.percentiles.intervals`'s value (in `intervals`).

```
if ( rpcQuantileEnable ) {
  rpcQueueTimeMillisQuantiles = new MutableQuantiles[ intervals ];
  for (int i = 0; i < intervals; i++) { ... }
}
```

(2) *Object creation.* Q is used to initialize an object and $c(p)$ is only used inside the created object. The following code snippet shows an example of such pattern from HDFS. The value of `dfs.datanode.available-space-volume-choosing-policy.balanced-space-preference-fraction` is only used when the value of `dfs.datanode.fsdataset.volume.choosing.policy` is `AvailableSpaceVolumeChoosingPolicy`, since the former parameter is only used inside the class `AvailableSpaceVolumeChoosingPolicy` [1].

```
VolumeChoosingPolicy<...> blockChooserImpl =
  ReflectionUtils.newInstance(conf.getClass(
  "dfs.datanode.fsdataset.volume.choosing.policy"), ...);

public class AvailableSpaceVolumeChoosingPolicy<...>
  implements VolumeChoosingPolicy{
  balancedPreferencePercent = conf.getFloat(
  "dfs.datanode.available-space-*.balanced-space-preference-fraction",...);
  ... }
```

4.1.2 **Default Value Dependency.** The default value of the dependent parameter p is a function of Q if and only if p is not currently assigned a value:

$$c(p) = \begin{cases} h(c(q_1), ..., c(q_n)), & \text{if } c(p) == \texttt{null} \\ c(p), & \text{otherwise} \end{cases} \quad \text{(A)}$$

where `null` means that a parameter is not mapped to a value.

[1] `dfs.datanode.available-space-volume-choosing-policy.balanced-space-preference-fraction` is abbreviated to save space.

Example. In one HDFS example, p = `dfs.namenode.edits.dir`, Q = {`dfs.namenode.name.dir`} and h is the identity function. `dfs.namenode.edits.dir` and `dfs.namenode.name.dir` specify the filesystem locations to store name tables and transactions, respectively. The latter serves as the default value of the former.

Code Patterns. Two code patterns that matched the cases in (A): (1) *In-file substitution.* One parameter value is explicitly used as the default value for the dependent parameter in the configuration file, as shown in the following example.

```
<name> dfs.namenode.checkpoint.edits.dir </name>
<value>${ dfs.namenode.checkpoint.dir }</value>
```

(2) *In-code substitution.* During execution, if the value of the dependent parameter is `null`, it is set to the value of another parameter. An example:

```
public static List<URI> getNamespaceEditsDirs(...){
  if ( editsDirs .isEmpty()) {  //dfs.namenode.edits.dir
    return getStorageDirs(conf, "dfs.namenode.name.dir");
  }
}
```

When `editDirs` (storing the value of `dfs.namenode.edits.dir`) is empty (i.e., not set), the value of `dfs.namenode.name.dir` is returned.

4.1.3 **Overwrite Dependency.** When multiple components are used together, some values for parameters defined in one component may be overwritten to be consistent with the parameter values in another component. Hence, in an overwrite dependency, at some point after p was initialized, $c(p) = h(c(q_1), ..., c(q_2))$. Overwrite dependencies in our data set are often inter-component dependencies and crashes can occur when an expected overwrite dependency does not hold. However, users may not be aware of overwrite dependencies, so there can be confusion as to why the system does not use the parameter values that users set.

Example. An example overwrite dependency from YARN and HDFS had p = `dfs.client.retry.policy.spec`, $Q = \{q\}$ where q =`yarn.resourcemanager.fs.state-store.retry-policy-spec`, and h as the identity function. p defines the timeouts and retries for HDFS clients; YARN uses HDFS as a distributed file system and overwrites p with its own parameter q.

Code Patterns. We identified two code patterns: (1) *Explicit overwrites.* The variable holding the dependent parameter's value is directly re-assigned. The following code snippet shows the overwrite dependency described above,

```
retryPolicy = conf.get(
      "yarn.resourcemanager.fs.state-store.retry-policy-spec", ...);
conf.set("dfs.client.retry.policy.spec", retryPolicy);
```

in which the `get` and `set` methods are used to read and overwrite configuration values, respectively. (2) *Implicit overwrites.* Multiple parameters are used to set the environmental variables and different environmental variables possess different priorities which form the overwriting relation implicitly. The following shows an example from MapReduce:

```
log4jPropertyFile = conf.get("mapreduce.job.log4j-properties-file");
vargs.add("-Dlog4j.configuration="+log4jPropertyFile);
logLevel = conf.get("yarn.app.mapreduce.am.log.level");
```

```
4 vargs.add("-Dhadoop.root.logger=" + logLevel + ",CRLA");
```

The environment variable `log4j.configuration` set by `mapreduce.job.log4j-properties-file` implicitly has higher priorities over the environment variable `hadoop.root.logger` set by `yarn.app.mapreduce.am.log.level`. Thus, `mapreduce.job.log4j-properties-file` overwrites `yarn.app.mapreduce.am.log.level`.

4.1.4 **Value Relationship Dependency.** The value of the dependent parameter p is constrained by the values of parameters in Q. We observed three kinds of such constraints: (1) *Numeric.* $c(p) = (A_1 \cdot c(q_1) \diamond ... \diamond A_n \cdot c(q_n)) + \varepsilon$, where $\diamond$ is any arithmetic operator, $A_1, ..., A_n$ are numeric coefficients, and ε is a positive or negative constant, or zero. (2) *Logical.* $c'(p) = c'(q_1) \circ ... \circ c'(q_n)$, where $\circ$ means any logical operator and c' is a special getter method that returns `true` or `false` depending on the value of a non-boolean q_i or $c(q_1)$ if q_i is boolean. It means the logical value $c'(p)$ should be equal to the logical value of $c'(q_1) \circ ... \circ c'(q_n)$. (3) *Set.* $c(p) \subseteq c(q_1) \odot ... \odot c(q_n)$, where $\odot$ can be any set operator. Failure to satisfy the constraints of value relationship dependencies can cause abnormal program behavior, including exit/abort, exceptions, and performance degradation. Constraints in a value relationship dependency are checked during component startup or during execution.

Example. In an Alluxio numeric value relationship dependency, p = `alluxio.master.worker.threads.max`, $Q = \{q\}$ where q =`alluxio.master.worker.threads.min`, and $\varepsilon \geq 0$. p and q define the max and min values of the thread pool size; hence $p \geq q$.

Code Patterns. Commonly, (1) numeric value relationship dependencies constrain a parameter value to not be greater (respectively, less) than a *max* (respectively, *min*) value specified by another parameter. The following shows an example from Alluxio,

```
1 mMinThreads=conf.getInt( "alluxio.master.worker.threads.min" );
2 mMaxThreads=conf.getInt( "alluxio.master.worker.threads.max" );
3 Preconditions.checkArgument(
4     mMaxThreads >= mMinThreads, ...);
```

(2) Logical value relationship dependencies are often used to specify that parameters need to be simultaneously enabled. The following shows an example from Spark,

```
1  if ( dynamicAllocationEnabled ) { //spark.dynamicAllocation.enabled
2    ExecutorAllocationClient =>
3    Some(new ExecutorAllocationManager(...)
4  }
5  private[spark] class ExecutorAllocationManager(
6    private def validateSettings(): Unit = {
7      if (!conf.get( "spark.shuffle.service.enabled" ) && !testing)
8        throw new SparkException("...")
9    }
10 }
```

If `spark.dynamicAllocation.enabled` (stored in `dynamicAllocationEnabled`) is `true`, Spark will create an `ExecutorAllocationManager` object which requires `spark.shuffle.service.enabled` to be `true`. Otherwise, an exception will be thrown. In short, `spark.dynamicAllocation.enabled` and `spark.shuffle.service.enabled` have to be enabled at the same time. (3) As the name implies, set relationship dependencies are often used to enforce that the value of one parameter must be the subset of values specified by another parameter. The following shows an example from YARN and Mapreduce.

```
1  Collection<String> shuffleProviders = conf.getStringCollection(
2      "mapreduce.job.shuffle.provider.services" );
3  Collection<String> auxNames = conf.getStringCollection(
4      "yarn.nodemanager.aux-services" );
5  for (String shuffleProvider: shuffleProviders)
6    if (auxNames.contains(shuffleProvider)) {
7      ...
8    } else {
9      throw new YarnRuntimeException();
10   }
```

Variable `auxNames`, which stores `yarn.nodemanager.aux-services` must be a subset of `shuffleProviders`, which stores `mapreduce.job.shuffle.provider.services`; otherwise, a runtime exception will be thrown.

4.2 Behavioral Dependencies

In a *behavioral configuration dependency*, there is no dependent parameter p whose value or scope depends on the values of other parameters. Rather the values of multiple parameters "co-operate" to influence some behavior of the system. More specifically, a set of parameters $P = \{p_1, ..., p_n\}$ have a behavioral dependency if they are used together in the same operation (such as a library call, system call or method call), such that changing the value of some $p \in P$ can alter a component's behavior.

Example. An example behavioral dependency in Hadoop Common is `connect(A, B)`, where $P = \{A, B\}$, A = `fs.ftp.host` and B = `fs.ftp.host.port`. Changing A but not B (and vice versa) could result in an attempt to connect to an IP address at a port that is not allowing connections—the effect of A is bounded by B.

Code Patterns. The code patterns for behavioral dependencies are (1) The library/system/method call has P as arguments, e.g.,

```
1 String host = conf.get( "fs.ftp.host" );
2 int port = conf.getInt( "fs.ftp.host.port" );
3 client.connect(host, port);
```

(2) The result of an arithmetic operation on elements in P is an argument to the library/system/method call. An example from HDFS,

```
1 editLogRollerThreshold =
2   conf.getLong( "dfs.namenode.checkpoint.txns" ) *
3   conf.getFloat( "dfs.namenode.edit.log.autoroll.multiplier.threshold" );
4
5 nnEditLogRoller = new Daemon(new NameNodeEditLogRoller(
      editLogRollerThreshold,...));
6 nnEditLogRoller.start();
```

`dfs.namenode.edit.log.autoroll.multiplier.threshold` determines the threshold, which in turn determines when an active node rolls its own edit log; `dfs.namenode.checkpoint.txns` controls after how many transactions a checkpoint will be created. These parameters are multiplied to obtain the threshold for rolling logs. They control how the function `start()` works.

4.3 One-off Code Patterns

We identify 30 configuration dependencies which fall in one of the dependency types defined in §4.1 and §4.2, but do not have the *common* code patterns described in §4.1 and §4.2. Among these one-off cases, 90% (27/30) are value relationship dependencies, while

	Hadoop		OpenStack	
Dependencies	**Intra**	**Inter**	**Intra**	**Inter**
Control	125	20	118	0
Value Relationship	46	54	60	3
Overwrite	5	11	0	0
Default Value	32	6	18	0
Behavioral Dependency	11	3	9	0

Table 2: The number of dependency types for intra- and inter-component dependencies in Hadoop and OpenStack.

Hadoop			OpenStack		
Component	**Intra**	**Inter**	**Component**	**Intra**	**Inter**
HCommon	38	36	Nova	65	0
HDFS	46	26	Swift	11	0
Yarn	46	57	Neutron	17	1
HBase	14	11	Keystone	42	1
Alluxio	10	8	Glance	16	2
Zookeeper	4	9	Placement	1	0
MapReduce	28	27	Ironic	28	2
Spark	33	14	Cinder	25	0
Total	219	94	Total	205	3

Table 3: The number of intra- and inter-component configuration dependencies found in each system. Inter-component dependencies count one for each component involved—the sum of unique inter-component dependencies is therefore half of the sum of the number of involved components.

10% (3/30) are behavioral dependencies; 76.7% (23/30) are intra-component and 23.3% (7/30) are inter-component.

We provide two examples. The first example involves parameters `dfs.hosts` and `dfs.hosts.exclude` in HDFS. The former specifies allowed data node addresses, while the latter specifies blocked data node addresses. That is, the intersection of values specified by these two parameters should be empty. However, from inspecting the code, we did not find how they are related. A second example, also in HDFS, involves the parameters, `dfs.namenode.replication.min` and `dfs.namenode.safemode.replication.min`. Both parameters control replication numbers: the former controls the replication number in normal mode while the latter controls the replication number in safe mode. Thus, the latter should be larger than the former (to be safe), but there is no code to check this dependency. Our future direction is to infer those logical dependencies that miss structural dependencies in code using other types of coupling measures (e.g., evolutionary coupling [46]).

4.4 Results of Grouping Dependencies by Type

Tables 2 and 3 show the results of grouping the configuration dependencies in our study of text sources by the types discussed in §4.1 and §4.2. We highlight four main observations from Table 2, which shows the intra- and inter-component configuration dependencies of various types in Hadoop and OpenStack. First, majority (95.6%) of configuration dependencies that we studied are functional. Second, control dependencies are the most common form of (functional) configuration dependencies, comprising 50.5% of the 521 dependencies that we studied. Third, across all dependency types, there are many more intra-component dependencies than inter-component dependencies, as expected (parameters should be used more inside the components in which they are defined). We further investigated the inter-component dependencies and found that the components that interacted the most were MapReduce and YARN with 22 inter-component dependencies. Finally, OpenStack has much fewer inter-component dependencies than Hadoop because components in OpenStack are much loosely coupled—each component provides independent services and uses RESTful APIs to communicate. Hence, in building cDep, we decided to focus on Java, in order to discover dependencies in Hadoop.

Table 3 shows how many intra-component dependencies and inter-component dependencies are in each of the 16 software systems that we evaluate. A key observation is that every software in our evaluation contains a configuration dependency, suggesting that configuration dependencies are widespread. On average a component has 33 configuration dependencies. Even though Placement is the smallest component with only 22 parameters, it has one configuration dependency.

We remind readers that the population of the configuration dependencies presented in Tables 2 and 3 should be taken with the specific systems and our methodology in mind.

4.5 Discussion

4.5.1 **Variables in Dependencies.** The majority of configuration dependencies only involve configuration values read from configuration file, while some configuration dependencies could also include variables whose values can only be evaluated at runtime. The following code snippet (from a value relationship dependency) illustrates the latter:

```
1 // "mapreduce.map.memory.mb"
2 Resource capability = getPerAllocationResource();
3 // "yarn.nodemanager.resource.memory-mb" - allocated_memory
4 Resource available = total_memory -allocated_memory
5 if (available > capability)
6   return new ContainerAllocation(pendingAsk, ALLOCATED);
7 else
8   return ContainerAllocation.LOCALITY_SKIPPED;
```

in which `capability` stores the requested memory (set by `mapreduce.map.memory.mb`), while `available` stores the total available memory, `total_memory` (set by `yarn.nodemanager.resource.memory-mb`). The variable `available` can only be evaluated at runtime.

4.5.2 **Dependencies that we do not cover.** In this paper, we mainly focus on configuration dependencies that are formed in *software programs*. We do not consider configuration dependencies that are formed externally in the *deployment environment*.

One such example is resource competition, in which different configuration parameters refer to external resources, such as CPU, memory, and operating system (OS) resources (e.g., IP addresses, ports, and file descriptors). In other words, p and Q (defined in §2.3) must satisfy external constraints enforced by the OS, virtual machine, or hardware that deploys the software systems. Take the example in §4.5.1; `yarn.nodemanager.resource.memory-mb` is constrained by the physical memory size and is competing with other co-running systems sharing the same machine. Resource competition is difficult to capture within the target software, without knowledge of the deployment environment. Therefore, we do not consider them in this paper.

5 DEPENDENCY HANDLING IN PRACTICE

In this section, we study how configuration dependencies are *checked*, *handled*, and *logged* in the software programs we study. We focus only on value relationship dependency types (§4.1.4) which have clear definitions of *violations* which occur when parameter values do not hold constraints. In principle, software programs should rigorously check that dependencies hold, handle any violations, and provide feedback to the system users [73, 86, 87, 89].

We also studied the other types of dependencies, such as control, default value, and overwrite dependencies. Unfortunately, we rarely found checking code or feedback messages in the program. For example, we observe that only 13 control dependencies have checking code or feedback messages. Our analysis follows the practice of source code inspection and violation injection [87, 89]. For a target configuration dependency, we start the system with crafted configuration values that intentionally violate the dependency, and then run workloads to exercise the code that uses the configurations; meanwhile, we observe the system behavior and logs to determine the impact of the violations.

5.1 Checking Configuration Dependencies

Checking that configuration dependencies hold has significant implications on the reliability, performance, and usability of software systems [89]. Without systematic and proactive checking, dependency violations would manifest as runtime exceptions, error code, failed assertions, or performance issues (discussed in §5.4).

Table 4 shows a break down of the three execution phases during which configuration dependency is checked in Hadoop and OpenStack: (1) *checking at initialization time*, (2) *checking at runtime* (after initialization), and (3) *no checks*. Note that neither "checking at runtime" nor "no check" is desirable—the former could raise runtime errors while the latter could degrade performance.

Observations. In Table 4, most dependencies (89% in Hadoop and 71.4% in OpenStack) have logic in the code to check that they hold. However, a significant percentage of dependencies (11% in Hadoop and 28.6% in OpenStack) have no checking logic—the program directly uses the dependent parameter even when the dependency is violated, as exemplified in Figure 1(c).

Moreover, 43% and 39.7% of the cases in Hadoop and OpenStack are checked after initialization, when it is often too late to prevent or recover from runtime exceptions or other failures and anomalous consequences [89]. The main reason is that not all modules are needed at the system's initialization phase; some modules are created on demand. Thus, in those on-demand modules, the checking code is only invoked when the module is created. Moreover, some dependency cases involve dynamic variables which can only be checked at runtime, as described in §4.5.1.

5.2 Handling Dependency Violations

We investigate how dependency violations detected by the checking logic (§5.1) are handled. Table 5a shows handling logic in three categories: (1) *exceptions:* the program does not recover from the violation; the violation is simply reported. Table 5a reports on *when* the exception is thrown, either at initialization time or runtime. (2) *correction:* the program enforces dependencies by correcting the violation; such correction could potentially lead to behavior that is different from what the users expect, due to deviation from the original parameter values set by users. Table 5a also reports whether the program logs its corrective actions as user notifications, (3) *logging only:* the program logs the dependency violation and continues its execution without invoking any handling logic.

Observations. Only 45% and 22.3% dependency violations are corrected in Hadoop and OpenStack, respectively. Of these corrected violations, 80% (32/40) and 70% (7/10) do not provide any log messages to users that parameter values were updated in Hadoop and OpenStack, respectively. The implication is that the software that we studied are missing many opportunities to correct dependency violations; they simply throw exceptions (40.4% of cases in Hadoop and 73.3% of cases in OpenStack) or log the violations (14.6% of cases in Hadoop and 4.4% of cases in OpenStack).

SW Stack	Init Time	Runtime	No Check	Total
Hadoop	46 (46.0%)	43 (43.0%)	11 (11.0%)	100
OpenStack	20 (31.7%)	25 (39.7%)	18 (28.6%)	63

Table 4: Checking practices of value relationship dependencies.

SW Stack	Exception		Correction		Logging only	Total
	Init Time	Runtime	w/ log	w/o log		
Hadoop	30 (33.7%)	6 (6.7%)	8 (9.0%)	32 (36.0%)	13 (14.6%)	89
OpenStack	18 (40.0%)	15 (33.3%)	3 (6.7%)	7 (15.6%)	2 (4.4%)	45

(a) Violation handling practices of value relationship dependencies.

SW Stack	Complete	Partial	Inadequate	None	Total
Hadoop	23 (25.8%)	17 (19.1%)	17 (19.1%)	32 (36.0%)	89
OpenStack	19 (42.2%)	10 (22.2%)	9 (20.0%)	7 (15.6%)	45

(b) Logging quality for violations of value relationship dependencies.

Table 5: Handling practices and feedback on dependency violations. We only include cases that have checking code in Table 4—"no check" cases are not handled.

5.3 Giving Feedback on Dependency Violations

We systematically examined the quality of log/error messages produced during the handling of dependency violations (§5.2). Table 5b shows four categories of feedback quality that we found: (1) *Complete:* the log message contains all parameters in the dependency and also describes the dependency, e.g.,

```
Preconditions.checkArgument(mMaxWorkerThreads >=
      mMinWorkerThreads, "alluxio.master.worker.threads.min" +" can
      not be less than "+ "alluxio.master.worker.threads.max" )
```

(2) *Partial:* the log message contains some but not all parameters in the dependency. It is hard to understand the dependency directly from the log message. The following is an example:

```
if ( recoveryEnabled ) { //mapreduce.jobhistory.recovery.enable
  storeClass = conf.getClass("mapreduce.jobhistory.recovery.
       store.class");
  if (storeClass == null)
    throw new RuntimeException("Unable to locate storage class,
         check mapreduce.jobhistory.recovery.store.class ");}
}
```

SW Stack	Usability	Startup Failure	Runtime Failure	Perf. Issues	Service Degrad.	Total
Hadoop	44 (44.0%)	30 (30.0%)	7 (7.0%)	12 (12.0%)	7 (7.0%)	100
OpenStack	24 (38.0%)	18 (28.6%)	15 (23.8%)	3 (4.8%)	3 (4.8%)	63

Table 6: Impact of violations of value relationship dependencies.

The message only pinpoints `mapreduce.jobhistory.recovery.store.class`, but does not mention `mapreduce.jobhistory.recovery.enable` (stored in `recoveryEnabled`) which can be disabled to fix the exception. (3) *Inadequate:* the log message contains no parameter. An example:

```
try:
  scheme = CONF.enabled_backends[store_id]
except KeyError:
  msg = _("Store for identifier %s not found") % store_id
  raise exceptions.UnknownScheme(msg)
```

the log message will only tell users identifier is not found, while telling neither parameter names. (4) *No message:* This mostly occurs when the program overrides the configuration values to enforce dependencies.

Observations. Majority of dependency violation handling logic (74.2% in Hadoop and 57.8% in OpenStack) do not provide complete log messages. 55.1% of log messages in Hadoop and 35.6% of log messages in OpenStack are in the "inadequate" or "none" categories. These results suggest that log enhancement tools [93, 94, 97] could be enhanced with configuration dependency information to improve the quality of these messages.

5.4 Consequences of Dependency Violations

Based on the analysis in §5.1—§5.3, we turn to the question, "what are the (potential) consequences of configuration dependency violations?" We find that violations of configuration dependencies can have several consequences, including (1) runtime failures (2) startup failures, (3) performance issues (4) usability issues, and (5) service degradation. Table 6 shows a breakdown of these categories of potential consequences for control and value dependencies.

Usability issues refer to systems ignoring or overwriting user-inputted configurations without providing users with feedback (e.g., logging). The typical cases are (1) changing a configuration parameter takes no effect due to not satisfying control dependencies, and (2) the user-inputted configuration is overwritten to resolve dependency violations without feedback. Although it keeps the system running, the overwritten value may not reflect users' intention which influences usability. Performance issues are defined by performance metrics and do not consider system feedback. We do not have a case that fits in both usability and performance.

49.0% and 57.2% of consequences are severe (i.e., failures or performance issues) for Hadoop and OpenStack, respectively. The following code snippet from HDFS shows an example in which a runtime exception is thrown when `dfs.replication` is less than `dfs.namenode.replication.min`,

```
replication = conf.get("dfs.replication");
minReplication = conf.get("dfs.namenode.replication.min");
if (replication < minReplication)
  throw new IOException(...);
```

Dependency violations can also lead to performance degradation. For example, when `mapreduce.map.cpu.vcores` exceeds `yarn.nodemanager.resource.cpu-vcores`, YARN will not grant more CPUs to MapReduce, slowing down the system. Many dependency violations could potentially lead to usability issues as the software either *silently* ignores or overwrites user-specified parameter values. As discussed in §2.1, configuration dependencies often lead to user confusion and questions in reality. We also observe service degradation such as log truncation and stale data.

6 AUTOMATED DEPENDENCY DISCOVERY

As discussed in §3.2.2, manual discovery of configuration dependencies is time-consuming. It took us 20 person months to discover, analyze, and validate the dependencies described in the documents for the 16 software projects, despite extensive scripting (§3.2.1). However, the understanding that we gained, including the definition and source code patterns, inspired our automated solution for discovering configuration dependencies. We present cDep, a tool for automatically discovering various types of configuration dependencies by statically analyzing the target software programs. cDep is built on the Soot compiler framework [32]. It analyzes Java bytecode and thus works for both Java and Scala programs. cDep takes the bytecode of multiple programs as input and outputs the configuration dependencies—the configuration parameters involved and the dependency types.

6.1 cDep Design and Implementation

The basic idea of cDep is intuitive. cDep first "colors" each program variable that stores a configuration parameter value based on static taint analysis—variables associated with different parameters have different colors and one variable could have multiple colors if its value is derived from multiple parameters. cDep then analyzes the dependencies between the colored variables using the source code patterns summarized in §4. If the variables match the patterns of a specific configuration dependency type, cDep records the corresponding configuration parameters and reports a dependency between them. Figure 2 shows the workflow of cDep.

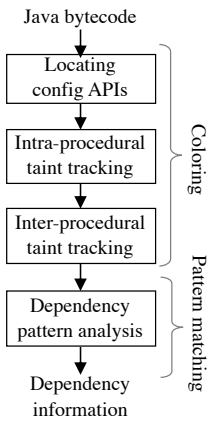

Figure 2: Workflow of the cDep tool.

6.1.1 Coloring. cDep colors program variables based on an implementation of static taint analysis on top of the Soot compiler infrastructure. Different parameters correspond to different taint colors. cDep taint analysis is interprocedural (to track values across methods), field sensitive (configuration values could be stored in a field of a class), and context sensitive (recording the calling context) (see the model in §2.2).

The initial taints are values read from the configuration getter APIs identified by cDep (§2.2). Taints are then propagated along the data-flow paths, through assignments, arithmetic operations, and string operations, until they reach sink statements. cDep supports

taint propagation through RPCs (Remote Procedure Calls) by mapping the caller stub interface to the callee implementation. A sink statement only consumes the parameter value; it does not further propagate the value, e.g., by using the value as a branch condition or passing the value to an external library or system call. We do not propagate taints via *all* control flows, to avoid over-tainting [74]. We do, however, analyze *some* control-flow dependencies of tainted variables to identify dependency types such as control dependency and logical value relationship.

6.1.2 Pattern Matching. With colored variables, cDep searches for patterns described in §4 to discover different types of dependencies: (1) *Control dependency.* If a branch condition uses variables from parameters $Q = \{q_1, ..., q_n\}$ and the branch condition dominates the sink statements of a parameter p, then cDep reports a control dependency between p and Q. cDep also finds, as an object creation pattern, if Q is used to initialize an object within which p is used (§4.1.1). (2) *Default value dependency.* cDep leverages the semantics of common configuration getter APIs in which the default value needs to be provided as an argument, e.g.,

```
<T> get(Class<T> class, String parameterName, T defaultValue);
```

If the default value of parameter p is tainted by other parameters in Q, a default value dependency is found. Moreover, cDep checks the pattern in which $c(p)$ is overwritten by parameters from Q, after checking p is not set (i.e., `NULL` or `isEmpty`). (3) *Overwrite dependency.* cDep captures explicit overwrites and identifies all the configuration rewrite APIs (in the form of setter methods) as shown in §4.1.3. We do not handle implicit overwrites, as they are not common (0.96% in our dataset) and it requires cDep to understand the parsing code that reads the values loaded into corresponding variables. (4) *Value relationship dependency.* For the numeric and set values, cDep identifies colored variables used in binary operator $\diamond \in \{\leq, \geq, <, >, =, \neq\}$ and set operations (e.g., `contains`). It outputs the operators if the numeric/set relationship is enforced in the program. cDep also identifies tainted variables used in `max/min` methods, which indicate numeric value relationships. For logical values, cDep searches for all tainted variables used in a logical expression. (5) *Behavioral dependency.* cDep identifies results of applying arithmetic operators $\diamond \in \{+, -, *, /\}$ to tainted variables which are then used in subsequent library/system/method calls. These are output as behavior dependencies. Moreover, if tainted parameters are used in Java's core library APIs, they are also output as behavior dependencies.

6.2 Evaluation

We applied cDep to the eight Java and Scala software components in the Hadoop-based stack (Table 1). Overall, cDep discovered 723 true configuration dependencies of the five target types, with a 6.0% average false positive rate. The breakdown based on dependency types is shown in Table 7. Two authors manually verified each dependency discovered by cDep based on the definition of the dependencies (§4) and the code location pointed by cDep; a third author will be looped in, whenever the two inspectors needed additional opinions for consensus. The verification process was the same as the process used in our study, described in §3.2.2.

Dependencies	Discovered	Known TP	New TP	FP
Control	372	143/145 (98.6%)	211	4.8%(18/372)
Value Relationship	155	80/100 (80.0%)	57	11.6%(18/155)
Overwrite	19	3/16 (18.8%)	16	0%(0/19)
Default Value	97	38/38 (100.0%)	59	0% (0/97)
Behavioral	126	11/14 (78.6%)	105	7.9%(10/126)
Overall	769	275/313 (87.9%)	448	6.0%(46/769)

Table 7: Evaluation results of applying cDep to the eight software projects in the Hadoop-based stack (Table 1). TP and FP stand for True and False Positives, respectively.

Among the 723 true dependencies that cDep discovered, 448 were not in our dataset collected from the documents (§3)—we were not aware of these until cDep discovered them. There are two reasons for the surprisingly large number of undocumented dependencies. First, many of the dependencies are control dependencies and default value dependencies as shown in Table 7; those dependencies do not lead to crashes or runtime exceptions. So, developers may not carefully document them even though they can lead to usability problems. Second, there is currently no systematic practice of discovering subtle configuration dependencies. Some dependencies are obvious but many of those discovered by cDep are subtle and even counter-intuitive: we ourselves did not understand some dependencies until we manually validated them.

We reverse-checked all the 448 dependency cases found by cDep that were not included in our study. We find that 94.4% (423) of the cases are not documented, while the rest 5.6% (25) are documented but are missed in our data collection methods (§3).

We also investigated the 38 false negatives and 46 false positives from cDep. As shown in Table 7, cDep identified 87.9% (275 out of 313) of the dependencies in our dataset. There are three reasons why cDep missed the remaining 12.1%: (1) 14 dependencies do not have common code patterns as discussed in §4.3—the patterns used by cDep cannot capture those dependencies. (2) Some projects use *ad hoc* means to overwrite parameters instead of the standard configuration APIs, which contributes to the 13 missing cases of overwrite dependencies. For example, HBASE uses substring matching to overwrite ZooKeeper parameters. (3) The remaining cases are dependencies that involve through external libraries, which cDep does not analyze. The false positives are mainly caused by over-tainting due to cDep's analysis not being path-sensitive—some variables should not be tainted as the variables will not store parameter values at runtime.

7 DISCUSSION

*Eliminating configuration dependencies*A solution to the complexity caused by configuration dependencies is to eliminate them via better configuration design. We believe that some dependencies are not necessary but result from poor configuration design. For example, the dependency between `dfs.hosts` and `dfs.hosts.exclude` in §4.3 should be eliminated—if a host string is in both, it is unknown whether it will be allowed or blocked. Also, min and max value dependencies can be designed as values in a range type to help users keep track of dependent parameters. On the other hand, many dependencies exist for good reasons, e.g., `mapreduce.map.memory.mb`

and `yarn.scheduler.maximum-allocation-mb` both control memory allocation at different levels. So, it is important to investigate radical new designs to eliminate unnecessary dependencies and to effectively manage existing ones.

Better Handling. Configuration dependencies are often not systematically handled w.r.t. checking, error handling, and feedback §5. Testing and analysis tools are needed to detect deficiencies in handling and to improve usability and reliability of configurable software. cDep can provide dependency information to enhance misconfiguration injection testing [87, 97] configuration checking/validation [37, 49, 70, 71, 79, 89, 95], and configuration-aware software testing [42, 45, 53, 60, 62–64, 69, 75–77].

Applying NLP to discover configuration dependencies. A potential direction is to add NLP (Natural Language Processing) to discover new types of configuration dependencies. Some dependencies collected from documents do not have common source code patterns (§4.3) and would be hard to find using program analysis. Using the definitions in §4, text features can be built with focus on descriptions of dependencies. Recent work has shown promises of applying NLP techniques to infer configuration constraints [83] but do not consider configuration dependencies between multiple configuration parameters. Further, data mining based methods can be used to discover more dependencies from large-scale datasets [85].

Threats to Validity. Manually finding configuration dependencies from text sources is error prone, and we may miss or mis-classify dependencies. To reduce this risk, two inspectors double-checked the results. Our results may not generalize to other systems; we only studied (1) software for cloud and datacenter systems, and (2) software with well defined configuration APIs. cDep can only find configuration dependencies with code patterns that we manually identified. Thus, we cannot claim to have found all the configuration dependencies in the projects studied. However, cDep proved our concept, and showed that automatic configuration dependency discovery is feasible and should be improved more in the future.

8 RELATED WORK

The prevalence and severity of software misconfigurations have driven the design and development of a number of detection and diagnosis techniques [34, 35, 37, 43, 49, 59, 70, 71, 79–82, 87, 89, 95, 96]. Detection aims at detecting misconfiguration before deployment, while diagnosis identifies root-causes of misconfigurations that caused failures, performance issues, and incorrect results.

Most existing techniques either implicitly assume or are explicitly scoped to find misconfigurations of individual parameters. However, prior studies show that 23.4%–61.2% of real-world misconfigurations involve multiple interdependent configuration parameters [92]. In those cases, each parameter value is correct in isolation, but the value combination violates dependency constraints. Hence, techniques for single-parameter misconfigurations cannot deal effectively with problems caused by configuration dependencies. cDep is one step towards enhancing existing techniques to make them reason about dependencies.

A few prior studies consider specific types of configuration dependencies [41, 68, 70, 71, 87, 95]. Most of these apply machine learning or data mining techniques to infer the dependent parameters from a large number of configuration files. As *a priori* knowledge, these techniques take configuration dependency types as inputs, either as learning templates [68, 95] or as language grammars [70, 71]. For example, if A is larger than B in a hundred of configuration file, those techniques infer a value relationship dependency, $A > B$.

Our work complements the above studies in the following three aspects. First, we cover more dependency types (e.g., default value, overwrite, and behavior dependencies) which are not covered by prior studies. Second, for control and value relationship dependencies presented in [87], our study shows more diverse code patterns. Third, no prior work provides formal definitions of configuration dependencies. By filling the knowledge gap, we believe that our work can significantly enhance existing tools to learn more types of configuration dependencies.

cDep is most related to Spex [87]. Spex attempts to automatically discover configuration dependencies from source code, including control dependencies, and numeric value relationships between two parameters. cDep differs from Spex in at least two aspects: (1) cDep is able to discover more dependencies with different types and different code patterns, benefiting from the systematic understanding of configuration dependencies in our study, and (2) cDep is generic to dependencies among more than two parameters, while Spex is hardcoded to two-parameter code patterns.

The notion of dependencies as a source of complexity has been studied in other domains. For example, dependencies of network router configurations are considered a key source of complexity of network management [39, 40] and software product lines [48, 55]. Our work focuses on configuration dependencies introduced and enforced by *software programs*, not networks or product lines.

Prior work [72] has studied cross-stack configuration errors, referred to as errors in one component caused by misconfigurations of other components. The concept is fundamentally different from configuration dependencies defined and studied in our paper.

9 CONCLUSION

This paper presents our study of, and tool for finding, configuration dependencies within and across software components. We define five types of configuration dependencies and identify their common code patterns. We also report on existing practices for handling these consequences, which are often deficient and *ad hoc*. Our tool, cDep is effective: it discovers known dependencies with high precision and recall and also finds 448 previously undocumented configuration dependencies. These results show that configuration dependencies are prevalent and diverse, that it is feasible to automatically discover them, and that they should henceforth be considered a first-class issue in software configuration engineering.

All the data and the cDep tool are made publicly available at:

https://github.com/xlab-uiuc/cdep-fse

ACKNOWLEDGEMENT

We thank Xudong Sun, Sam Cheng, Jack Chen, and Elaine Ang for useful discussions and paper proofreading. Xudong Sun and Wenyu Wang helped validate the reproduction of the artifacts for the paper rebuttal. Xuan Peng contributed to the study of OpenStack.

REFERENCES

[1] Apache Users Mailing List. heartbeat timeout doesn't work. http://mail-archives.apache.org/mod_mbox/hadoop-user/201407.mbox/<53BA5D42.7080202@oss.nttdata.co.jp>, 2014.

[2] Apache Users Mailing List. block replication. http://mail-archives.apache.org/mod_mbox/hadoop-user/201401.mbox/<20140101182042490212 5@jd.com>, 2014.

[3] HBASE ISSUE 13200. Improper configuration can leads to endless lease recovery during failover. https://issues.apache.org/jira/browse/HBASE-13200/, 2015.

[4] MAPREDUCE ISSUE 4604. In mapred-default, mapreduce.map.maxattempts & mapreduce.reduce.maxattempts defaults are set to 4 as well as mapreduce.job.maxtaskfailures.per.tracker. https://issues.apache.org/jira/browse/MAPREDUCE-4604/, 2015.

[5] HDFS Short-Circuit Local Reads. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html, 2018.

[6] Hadoop in Secure Mode. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-common/SecureMode.html, 2018.

[7] The YARN Timeline Server. https://hadoop.apache.org/docs/r2.9.2/hadoop-yarn/hadoop-yarn-site/TimelineServer.html, 2018.

[8] Keystone configuration options. https://docs.openstack.org/keystone/stein/configuration/config-options.html, 2019.

[9] Alluxio 1.8 - Overview. https://docs.alluxio.io/os/user/1.8/en/Overview.html, 2019.

[10] OpenStack Docs (Stein) : OpenStack Block Storage (Cinder) documentation. https://docs.openstack.org/cinder/stein/, 2019.

[11] Hadoop Common Configurations. core-default.xml. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-common/core-default.xml, 2019.

[12] OpenStack Docs (Stein) : Welcome to Glance's documentation! https://docs.openstack.org/glance/stein/, 2019.

[13] Apache HBase 2.2.1 - Reference Guide. https://hbase.apache.org/book.html, 2019.

[14] Apache Hadoop 2.9.2 - Hadoop: CLI MiniCluster. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-common/CLIMiniCluster.html, 2019.

[15] HDFS Configurations. hdfs-default.xml. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml, 2019.

[16] Apache Hadoop 2.9.2 - HDFS Architecture. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2019.

[17] OpenStack Docs (Stein) : Welcome to Ironic's documentation! https://docs.openstack.org/ironic/stein/, 2019.

[18] OpenStack Docs (Stein) : Keystone, the OpenStack Identity Service. https://docs.openstack.org/keystone/stein/, 2019.

[19] Hadoop MapReduce Configurations. mapred-default.xml. https://hadoop.apache.org/docs/r2.9.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml, 2019.

[20] Apache Hadoop 2.9.2 - MapReduce Tutorial. https://hadoop.apache.org/docs/r2.9.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, 2019.

[21] OpenStack Docs (Stein) : Welcome to Neutron's documentation! https://docs.openstack.org/neutron/stein/, 2019.

[22] Nova configuration options. https://docs.openstack.org/nova/stein/configuration/config.html, 2019.

[23] OpenStack Docs (Stein) : OpenStack Compute nova. https://docs.openstack.org/nova/stein/, 2019.

[24] OpenStack Docs (Stein) : Placement. https://docs.openstack.org/placement/stein/, 2019.

[25] Spark 2.4.0 - Overview. http://spark.apache.org/docs/2.4.0/, 2019.

[26] OpenStack Docs (Stein) : Welcome to Swift's documentation! https://docs.openstack.org/swift/stein/, 2019.

[27] Yarn Configurations. yarn-default.xml. https://hadoop.apache.org/docs/r2.9.2/hadoop-yarn/hadoop-yarn-common/yarn-default.xml, 2019.

[28] Apache Hadoop 2.9.2 - Apache Hadoop YARN. https://hadoop.apache.org/docs/r2.9.2/hadoop-yarn/hadoop-yarn-site/YARN.html, 2019.

[29] ZooKeeper 3.5.4 - Administrator's Guide. http://zookeeper.apache.org/doc/r3.5.4-beta/zookeeperAdmin.html, 2019.

[30] Ironic configuration options. https://docs.openstack.org/ironic/stein/configuration/config.html, 2020.

[31] Neutron configuration options. https://docs.openstack.org/neutron/stein/configuration/neutron.html, 2020.

[32] Soot: a Java Optimization Framework. http://sable.github.io/soot/, 2020.

[33] G. Amvrosiadis and M. Bhadkamkar. Getting Back Up: Understanding How Enterprise Data Backups Fail. In *Proceedings of 2016 USENIX Annual Technical Conference (ATC'16)*, June 2016.

[34] M. Attariyan and J. Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[35] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, October 2012.

[36] L. A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines (Third Edition)*. Morgan and Claypool Publishers, 2018.

[37] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci. Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17), Industrial Track*, December 2017.

[38] F. Behrang, M. B. Cohen, and A. Orso. Users Beware: Preference Inconsistencies Ahead. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, August 2015.

[39] T. Benson, A. Akella, and D. Maltz. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)*, 2009.

[40] T. Benson, A. Akella, and A. Shaikh. Demystifying Configuration Challenges and Trade-Offs in Network-based ISP Services. In *Proceedings of 2011 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'11)*, August 2011.

[41] W. Chen, H. Wu, J. Wei, H. Zhong, and T. Huang. Determine Configuration Entry Correlations for Web Application Systems. In *Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC'16)*, June 2016.

[42] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering (TSE)*, 34(5):633–650, September 2008.

[43] Z. Dong, A. Andrzejak, and K. Shao. Practical and Accurate Pinpointing of Configuration Errors using Static Analysis. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, September 2015.

[44] Z. Dong, A. Andrzejak, D. Lo, and D. Costa. ORPLocator: Identifying Read Points of Configuration Options via Static Analysis. In *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE'16)*, October 2016.

[45] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback Driven Adaptive Combinatorial Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*, July 2011.

[46] H. Gall, K. Hajek, , and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the 1998 International Conference on Software Maintenance (ICSM'98)*, November 1998.

[47] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*, October 2016.

[48] G. Holl, D. Thaller, P. Grünbacher, and C. Elsner. Managing Emerging Configuration Dependencies in Multi Product Lines. In *Proceedings of 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, January 2012.

[49] P. Huang, W. J. Bolosky, A. Sigh, and Y. Zhou. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th ACM European Conference in Computer Systems (EuroSys'15)*, April 2015.

[50] A. Hubaux, Y. Xiong, and K. Czarnecki. A User Survey of Configuration Challenges in Linux and eCos. In *Proceedings of 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, January 2012.

[51] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008.

[52] D. Jin, M. B. Cohen, X. Qu, and B. Robinson. PrefFinder: Getting the Right Preference in Configurable Software Systems. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*, September 2014.

[53] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations Everywhere: Implications for Testing and Debugging in Practice. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[54] S. Kendrick. What Takes Us Down? *USENIX ;login:*, 37(5):37–45, October 2012.

[55] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. In *Proceedings of the 2020 ACM SIGMETRICS Conference (SIGMETRICS'20)*, June 2020.

[56] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-time Configuration Options. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*, September 2014.

[57] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering (TSE), Early Access*, September 2017.

[58] B. Maurer. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM*, 58(11):44–49, November 2015.

[59] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar. Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, February

2020.

[60] T. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter. iGen: Dynamic Interaction Inference for Configurable Software. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, November 2016.

[61] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, March 2003.

[62] X. Qu. Configuration Aware Prioritization Techniques in Regression Testing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, May 2009.

[63] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*, July 2008.

[64] X. Qu, M. Acharya, and B. Robinson. Impact Analysis of Configuration Changes for Test Case Selection. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE'11)*, November 2011.

[65] A. Rabkin and R. Katz. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, November 2011.

[66] A. Rabkin and R. Katz. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, May 2011.

[67] A. Rabkin and R. Katz. How Hadoop Clusters Break. *IEEE Software Magazine*, 30 (4):88–94, July 2013.

[68] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury. Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications. In *Proceedings of the 6th International Conference on Autonomic Computing and Communications (ICAC'09)*, June 2009.

[69] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32rd International Conference on Software Engineering (ICSE'10)*, May 2010.

[70] M. Santolucito, E. Zhai, and R. Piskac. Probabilistic Automated Language Learning for Configuration Files. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, July 2016.

[71] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac. Synthesizing Configuration File Specifications with Association Rule Learning. In *Proceedings of 2017 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*, October 2017.

[72] M. Sayagh, N. Kerzazi, and B. Adams. On Cross-stack Configuration Errors. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, May 2017.

[73] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo. Software Configuration Engineering in Practice: Interviews, Surveys, and Systematic Literature Review. *IEEE Transactions on Software Engineering (TSE), Preprint*, 2018.

[74] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE symposium on Security and privacy (S&P'10)*, 2010.

[75] C. Song, A. Porter, and J. S. Foster. iTree: Efficiently Discovering High-Coverage Configuration Using Interaction Trees. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[76] S. Souto, P. Barros, and R. Gheyi. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, May 2017.

[77] H. Srikanth, M. B. Cohen, and X. Qu. Reducing Field Failures in System Configurable Software: Cost-Based Prioritization. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering (ISSRE'09)*, November 2009.

[78] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, November 2020.

[79] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)*, October 2015.

[80] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, December 2004.

[81] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*, October 2003.

[82] C. Xiang, Y. Wu, B. Shen, M. Shen, H. Huang, T. Xu, Y. Zhou, C. Moore, X. Jin, and T. Sheng. Towards Continuous Access Control Validation and Forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*, November 2019.

[83] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*, July 2020.

[84] T. Xu and O. Legunsen. Configuration Testing: Testing Configuration Values as Code and with Code. *arXiv:1905.12195*, July 2019.

[85] T. Xu and D. Marinov. Mining Container Image Repositories for Software Configurations and Beyond. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18), New Ideas and Emerging Results*, May 2018.

[86] T. Xu and Y. Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys*, 47(4), July 2015.

[87] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP'13)*, November 2013.

[88] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, August 2015.

[89] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, November 2016.

[90] T. Xu, V. Pandey, and S. Klemmer. An HCI View of Configuration Problems. *arXiv:1601.01747*, January 2016.

[91] T. Xu, H. M. Naing, L. Lu, and Y. Zhou. How Do System Administrators Resolve Access-Denied Issues in the Real World? In *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)*, May 2017.

[92] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, October 2011.

[93] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, October 2012.

[94] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, October 2014.

[95] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*, March 2014.

[96] S. Zhang and M. D. Ernst. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[97] S. Zhang and M. D. Ernst. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*, July 2015.