An Automatic Verification Framework for Software Configurations

Abstract

System failures resulting from configuration errors have become the major causes affecting the availability and reliability of today's software systems. Although many misconfiguration handling techniques, *e.g.*, misconfiguration checking, troubleshooting and repair, have been proposed, offering automatic verification for configuration files – like what we did to regular programs – is still an open problem. This is because software configurations are typically written in poorly structured and untyped "languages", and specifying constraints and rules for configuration verification is non-trivial in practice.

This paper presents, ConfigV, the first automatic verification framework for general software configurations. ConfigV verifies a target configuration file F through three steps. First, ConfigV analyzes a dataset containing many sample configuration files belonging to the same system as F, and translates these sample files to a well-structured and probabilistically-typed intermediate representation. Second, ConfigV derives rules and constraints by analyzing this intermediate representation, thus building a sophisticated language model. Finally, ConfigV uses the resulting language model to verify F. ConfigV framework is highly modular, does not rely on the system source code or templates and can be applied to any new configuration file type with minimal user input.

ConfigV is capable of detecting various tricky errors that cannot be detected by previous efforts, including ordering errors, fine-grained value correlation errors, entry missing errors, and environment related errors. **Ennan:** [One or two sentences here describe the most important experimental results. For example, we verify real-world misconfiguration files and find 95% errors are detected.]

1 Introduction

Configuration errors are one of the most important root causes of today's software system failures [22, 23]. For example, Yin *et al.* [23] reveal that about 31% system failures were caused by misconfiguration problems. Misconfigurations, in practice, may result in various system-wide problems, such as security vulnerabilities, application crashes, severe disruptions in software functionality, and incorrect program executions [20, 21, 24, 25].

While many efforts have been proposed to check, troubleshoot, diagnose and repair configuration errors [6, 17, 19], offering automatic verification to configuration files – like what we did for verifying regular programs – is still highly desirable [18, 22, 25]. Nevertheless, the main obstacles to the automatic configuration verification are: 1) entries in configuration files are untyped entries, 2) there is no explicit structure policy for the entries in configuration files, and 3) there are surprisingly few rules specifying the entries' constraints.

In order to overcome these obstacles, researchers have proposed statistical analysis and learning based approaches [18, 24, 25]. These efforts build checking policies by learning a sample data set, rather than explicitly specifying entries' types or rules. In particular, for each entry in a certain configuration file, if it deviates from the common values used in a large collection of configurations (*i.e.*, the data set for training), it is typically suspected as a potential configuration error. However, because these learning efforts *either* are limited to simplistic configuration errors (*e.g.*, type errors and syntax errors), *or* heavily rely on template-based inference [25], many sophisticated configuration errors, *e.g.*, hard to be templated in practice, cannot be detected. For example, if `extension = mysql.so` appears before `extension = recode.so` in PHP configuration file, it would lead to a crash error, since the correct ordering should be `extension = recode.so` before `extension = mysql.so` [23]; however, such an error cannot be detected by existing learning efforts, since it is hard to build a template for this error [22].

In order to truly achieve automatic configuration verification, we argue that we need to leverage a collection of more powerful learning algorithms (not necessarily depending on templates) to derive a more sophisticated language model (with abundant rules covering tricky configuration errors), and then check the configuration files of interest through the learned language model.

Based on the above argument, this paper presents ConfigV, the first automatic verification framework for general software configurations. In particular, ConfigV's workflow to verifying a given configuration file could be looked as a three-step methodology. First, ConfigV analyzes a dataset containing sample configuration files, thus generating a well-structured and probabilistically-typed intermediate representation. Second, ConfigV de-

rives rules and constraints by analyzing the intermediate representations, thus building a language model. Finally, ConfigV uses the resulting language model to verify the given configuration file and detect potential errors. Compared with previous efforts, ConfigV does not necessarily rely on users' templates, and is capable of detecting more tricky configuration errors that cannot be identified by existing work (details in §2).

Building such an automatic verification framework for configuration files, nevertheless, requires addressing several challenges. First, in order to formulate a correct language model, we need to infer each entry's type in a given configuration file; however, the type of a variable cannot always be fully determined from a single value. For example, an entry `foo = MAX_SIZE` is most likely an integer rather than string; however, existing type inference work would report this is an error, because foo should be assigned an integer [25]. We address this problem by introducing so called *probabilistic types*. Rather than assigning only one variable to a single type, we assign several types with their probability distributions. The entry in the above example might be assigned a probabilistic type like `{foo, MAX_SIZE, [(Int, 60%), (String, 40%)]}`. With such probabilistic types in hand, we can generate a more accurate language model, thus significantly improving our checking capability.

Second, without template, how to learn rules and constraints present a difficult. **Ennan:** [Here, we need to add one-paragraph description to illustrate how we learn rules without templates, or why we say our learning results are better than existing learning approach, *e.g.*, EnCore.]

From a practical perspective, ConfigV introduces no additional burden to the users: they can simply use ConfigV to check for errors in their configuration files. However, they can also easily extend the framework themselves. The system is designed to be highly modular. If there is a class of rules that ConfigV is not currently learning, the user can develop their own templates and learners for that class. The new learner can be added to ConfigV and this way it can check additionally a new set of errors.

Our ConfigV prototype still has a few limitations: for example, we cannot handle configuration errors that can be only triggered in system execution time. Nevertheless, we believe ConfigV may suggest a practical path toward automatic and modular language-based configuration verification. To summarize, this tool paper makes the following contributions:

1. We propose the first automatic configuration verification framework, ConfigV, that can learn a language model from an example set of correct configuration files, and then uses this language model to verify interested configuration files.

2. ConfigV proposes probabilistic types to assign a confidence distribution over a set of types to each entry, while generating the intermediate representation.

3. ConfigV employs a collection of machine learning algorithms to enable powerful rule and constraint inference without the assistance from any pre-defined templates.

4. ConfigV is capable of detecting various tricky errors that cannot be detected by previous efforts, including ordering errors, fine-grained value correlation errors, entry missing errors, and environment related errors.

5. We implement a ConfigV prototype and evaluate it by conducting comprehensive experiments.

2 Motivating Examples

In this section, we present several *tricky* configuration errors extracted from real-world examples. The reason we call them tricky is most of existing efforts, *e.g.*, learning-based checking efforts [18, 25], cannot detect these configuration errors. On the other hand, we also investigated many misconfiguration problems in practice were caused by these tricky configuration errors, *e.g.*, reported by Stack Overflow.

Example 1: Ordering errors. The first example configuration error is first reported by Yin *et al.* [23]. For instance, when a user configures PHP to run with the Apache HTTP Server, this user may write, among others, the following lines in the configuration file.

```
extension = mysql.so
...
extension = recode.so
```

Such a configuration file will cause the Apache server to fail to start due to a segmentation fault error [23]. This is because when using PHP in Apache, the extension `mysql.so` depends on `recode.so`, and the relative ordering of two of them is crucial. We call the above example misconfiguration situation as *ordering errors*. Yin *et al.* find that ordering errors widely exist in many system configurations, *e.g.*, PHP and MySQL, and typically lead to multiple system crash events; however, there is no existing effort that can effectively solve or detect this problem [21, 22, 25].

If this user is using ConfigV, she can avoid such a configuration error. In particular, ConfigV is able to inform the user that `recode.so` should appear before `mysql.so`, and reports the error (as shown below).

```
ORDERING ERROR: Expected "extension" "recode.so"
BEFORE "extension" "mysql.so"
```

Example 2: Fine-grained value correlation errors. In a real-world misconfiguration example [1], a user configures his MySQL as the following:

```
key_buffer_size = 384M
max_heap_table_size = 128M
max_connections = 64
thread_cache_size = 8
...
sort_buffer_size = 32M
join_buffer_size = 32M
read_buffer_size = 32M
read_rnd_buffer_size = 8M
...
```

This user complained his MySQL's load is very high and the website's respond speed is very slow. In this case, `key_buffer_size` is used by all the threads cooperatively while `join_buffer` and `sort_buffer` are created by each thread for private use; thus, the maximum amount of used key buffer, *i.e.*, `key_buffer_size` should be larger than `join|sort_buffer_size` * `max_connections`. Clearly, in the above example, it does not hold, so that this misconfiguration causes MySQL loads very slow.

We call the above situation as a *fine-grained value correlation error*, which is a more challenging case than *value correlation error* reported by some existing investigations [23, 25]. A typical value correlation error means one entry's value should have a certain correlation with another entry's value. For example, in MySQL, the value of `max_connections` should be higher than `mysql.max_persistent`. Although some work, *e.g.*, EnCore [25], can detect the normal value correlation case, none of existing efforts can detect the fine-grained value correlation errors, which "hides" an equation-aware correlation, as shown in the above example. As mentioned by Xu *et al.* [20], detecting the fine-grained value correlation errors present a much more challenging task than normal value correlation problems.

To the best of our knowledge, ConfigV is the first effort that is able to check such fine-grained value correlation problem. If we run ConfigV on this configuration file, ConfigV would return:

```
INTEGER RELATION ERROR:
Expected "key_buffer_size" >= "max_connections" * "
    sort_buffer_size"
```

Note that existing effort, EnCore, aims to detect value correlation errors. However, different from ConfigV, EnCore can only detect simple correlation, *e.g.*, `mysql.max_persistent` < `max_connections`, rather than the fine-grained computation-based value correlation, like `key_buffer_size >= max_connections * sort_buffer_size` identified by ConfigV.

Example 3: Entry missing errors. Many critical system outages result from the fact that important entry was missing in the configuration file. We call such a problem as *entry missing error*. In a public misconfiguration dataset [2], the major problems of MySQL failure reports were caused by entry missing errors. Below is a real-world entry missing error example [23], when a user wants to use OpenLDAP to enable her directory access protocol, she needs to use the password policy overlay. This is usually done through the following entries in the OpenLDAP configuration file:

```
include schema/ppolicy.schema
overlay ppolicy
```

When using the password policy overlay in OpenLDAP, users have to first include the related schema. Leaving out the "include" entry, which have been left by many users [23], will cause the failure of this LDAP. If this user has ConfigV, she can run ConfigV on such a misconfiguration file, and ConfigV would return:

```
MISSING KEYWORD ERROR: Expected "overlay" "ppolicy"
in the same file: "include" "schema/ppolicy.schema"
```

Ennan: [Example 4: This should be the singular value error. We need an example from Stack Overflow to illustrate the real-world problem. I guess the buffer size example is fine.]

Other errors. ConfigV can also deal with the configuration errors that can be detected by existing work, such as EnCore [25] and CODE [24]. For a configuration error reported recently made MySQL daemon failed to start [4]. One entry in a MySQL configuration file is written as `datadir=/root/appfinder/mysql`, and the type as well as format seem to be correct. However, the problem is this directory should not be initiated as a root directory, and the correct entry should be `datadir=/appfinder/mysql`. Such an error is called system environment related configuration error. Both ConfigV and EnCore can handle system execution related misconfiguration problems. Furthermore, ConfigV is also able to detect type errors and syntax errors in configuration files.

3 The ConfigV Framework Overview

We propose ConfigV, an automatic verification framework for software configuration files. In particular, ConfigV can solve many sophisticated configuration errors (*e.g.*, ordering errors, missing entry errors and fine-grained value correlation) that previous efforts cannot detect. As depicted in Figure 1, a typical ConfigV verification workflow has three steps: translation, learning and

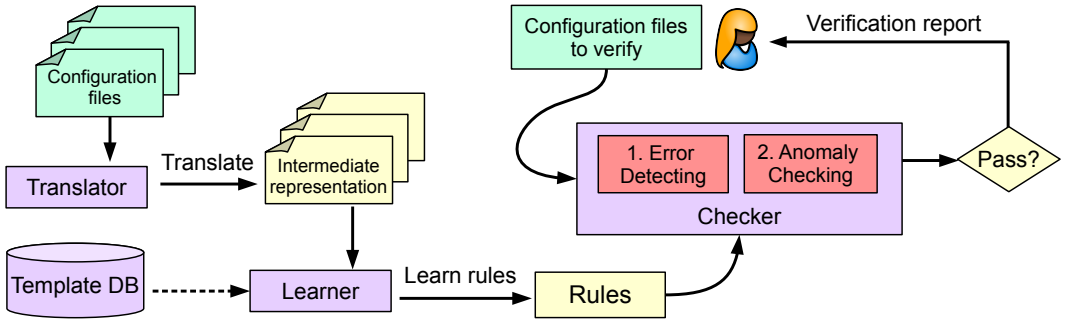

Figure 1: ConfigV's workflow. The green components represent configuration files, including both sample configuration datasets and users' input configuration files to verify. The purple components are the modules of ConfigV. Because template DB is not necessarily used, we use dashed arrow between it and the learner. Red boxes are sub-modules within the checker. The yellow components are results generated by ConfigV's modules.

checking. In this section, we briefly describe how does each step work.

Initial phase. We start with the assumption that we are given a number of (not necessarily correct) configuration files belonging to the same system, such as MySQL or Apache. These given files follow similar patterns, which we exploit in a learning algorithm to build rules that describe a language model for the files.

Translator. The translator module first parses the input sample dataset (containing both configuration files and system environment information), and then transforms them to a more structured and typed intermediate representation. When we run type inference on a configuration file, the type of a variable cannot always be fully determined from a single value. We address this problem by introducing so called *probabilistic types*. Rather than giving a variable a single type, we assign several types with their probability distributions. We can later use these more structured files as a training set to learn the rules.

Learning. The input of learner is a set of files that have been translated into well-structured and typed representations. The learner module employs a collection of learning algorithms to generate various rule and constraints, potentially used to handle different types of configuration errors. These rules are the output of the learner, and will be used by the checker to detect errors. By combining the translated representations and the learned rules together, we already build a language model for configuration verifications. Because the translator outputs probabilistic typed entries, the learner is responsible for determining a type for each entry.

Different from previous efforts, *e.g.*, EnCore [25], which needs users or developers to provide explicit templates, ConfigV's learner encodes some predefined error-patterns to generate rules. As an illustration of a simple rule that we can learn, consider an encoded pattern $X_1 \leq X_2$, where X_1 and X_2 are integer variables. The learner may derive the rule stating that `mysql.max_persistent` $\leq$ `max_connections`. There is a classification and taxonomy of configuration errors in the existing work on automated configuration troubleshooting [2, 23]. Each class is looked as an error-pattern that ConfigV should handle: we consider integer constraints, ordering errors, typing errors, correlation errors, etc. **Ennan:** [I do know we are using in some sense templates, but we should use pattern or some words to persuade we are using an implicit "templates"; otherwise, too similar to what EnCore did.]

Although the learner does necessarily rely on templates, ConfigV still offers a database to contain many templates, as shown in Figure 1. Some of these templates are responsible for offering specific system executional environment information. It is widely known that any existing learning algorithm cannot derive rules checking environment violation, *e.g.*, whether current account is the owner of a certain path. In order to deal with comprehensive misconfiguration problems, the learner needs a template DB to provide environment information, thus detecting system environment related configuration errors.

Checking. The checker is used to detect the rule violations in the configuration files of interest. The inputs of checker are the learned rules and the target configuration file to verify. Its outputs a report (as shown in §2) about whether it finds any error, *e.g.*, rule violations, or a suspicious value. As shown in Figure 1, there are two sub-modules in the checker. They are responsible for checking rule violations and suspicious values, respectively. **Ennan:** [I really wanted to say we are using MaxSMT to check and rank these errors ...] In our experience, we find learned rules can be significantly reused to check different configuration files, thus improving our usability.

4 Translator

The responsibility of the translator is to transform a given set of configuration files into another set of files written in typed and well-structured representations. In some sense, the translator looks like a parser, and it is only used to generate an intermediate representation for the post process, such as learning rules (see §5). Coupled with rules generated by learner module, we will have a complete language model to verify configuration files of our interest.

Translating or parsing is system dependent. In other words, for MySQL and HDFS, we need to develop different parsers to handle each of them, respectively. ConfigV allows users to provide extra help to the translator for their specific system configuration, such as specifying a comment character. Each of the entries in the intermediate representations must be assigned types, and these types would guide the learning modules later.

Ennan: [We need a kind of formal definition (using a figure) to define the grammar of our intermediate representation. The basic view of this definition should be similar to define a type language grammar.]

Introducing probabilistic types. Based on a single example value of v, the translator cannot fully determine the type of k. Consider for instance the following example:

```
foo = 300
bar = 300.txt
```

Most likely `foo` should be an integer, but it could also be a string. In this case, we want to learn the rule like `foo` $\in$ substrings(`bar`). The translator, therefore, assigns a distribution of types to a value, an idea closely related to existentially quantified types [12]. We introduce *probabilistic types* to address this issue.

Let $\mathscr{T}$ be a set of basic types. In ConfigV set $\mathscr{T}$ contains strings, integers, file paths, sizes and IP addresses. A probabilistic type built from $\mathscr{T}$ is a list of pairs $[(\tau_1, p_1), \ldots, (\tau_n, p_n)]$, such that $\tau_i \in \mathscr{T}$, $0 \leq p_i \leq 1$ and $\Sigma p_i = 1$. These probabilities are updated each time a new example value for a key is encountered.

When a value has a probabilistic type, we generate rules for all its types. This means that by assigning `foo` a probabilistic type (*e.g.*, `(foo, 300, [(Int,90%), (String,10%)])`, we would generate rules for both strings and integers. Once the type inference can uniquely determine the type, the probability of all other types is set to zero, and the associated rules are withdrawn.

Note that typing is also a system module than can be easily extended to support more types. In that case the user will need to provide rules for type inference and probability distributions for values where type inference is ambiguous.

5 Learner

Ennan: [This section details how do we extract rules from the intermediate representations. This section may should have the following two subsections: Aaron's part (including Xinyu's fine-grained value correlation rule learning) and Jonathon's part. In addition, we may need to add one more subsection about learning system environment information related rules, *e.g.*, for paths and users.]

6 Ranges

We discuss the technique we use to detect anomalous values for numerical attributes. For a given test file, let A be the set of attributes contained in the file. Let A_n be the subset of attributes of A which are numerical types. Then, for each attribute $a \in A_n$, we construct a vector v_a of the values corresponding to attribute a seen over the entire data-set. For each v_a, we compute an interval

$$[\hat{v_a} - 3 * MAD(v_a), \hat{v_a} + 3 * MAD(v_a)],$$

where $\hat{v_a}$ represents the median over the values in v_a and $MAD(v_a)$ refers to the median absolute deviation. This is a standard outlier detection test, namely the Hampel identifier. [1] If the value for a numerical attribute in a test file falls outside of this interval, a warning is printed to the user indicating the violating value, the attribute, and the upper or lower Hampel threshold.

The intuition for this test is that if the user has input a value that falls outside of an interval containing values that are considered "normal" over the entire data set, that value will probably cause an error, in particular for performance. We cannot know for sure if this value will cause an issue. For instance, a user might have a machine with particularly high-end hardware, in which case a value beyond the upper Hampel threshold may be appropriate.

7 Checker

With the learned rules and constraints in hand, ConfigV checks whether any entry in a target configuration file violates the learned rules. For a given configuration file, ConfigV parses it using the same way employed in the translator, thus obtaining a structured and typed representation for the target configuration file. Then, the

[1] Mathematically, $MAD(v_a) = 1.4826 * median(|v_a - \hat{v_a}|)$, estimating standard deviation for a normal distribution.

checker uses two sub-modules to check the following aspects of the target configuration file.

Ennan: [I am still working on the following stuff ...] **Ordering violation.** ConfigV can check whether two or more entries have the ordering problem, which means whether the order of some entries in the target configuration follows the rules output by the learner.

Correlation violation. ConfigV checks if the target configuration file follows the correlation rules (including fine-grained value correlation rules) learned from the training set. The rule would be ignored if the involved entries are absent in the target configuration file.

Data type violation. For each entry to be checked in the target configuration file, the checker reads its type information inferred from the training set, and uses the generated language model to verify whether the types are matched. A type violation is reported if the verification fails.

Suspicious warning. This checking occurs in the second sub-module of the checker. Different from the previous checking tasks, suspicious warning is just to detect whether some value is too different (or distinguished) from the same entries in the training dataset. Even if some values are statistically different from the ones in the training dataset, it does not mean such a value is incorrect; thus ConfigV, in this case, throw out a warning to the user who enters the target configuration file, and a report containing normal values in the training dataset. ConfigV allows users to choose whether they want to change the values according to the ones in the training configuration files or not.

8 Discussion and Limitations

This section discusses a few ConfigV's limitations and possible solutions.

Legal misconfigurations. While ConfigV can check diverse configuration errors without human participations, most of the existing proactive misconfiguration detection techniques, including ConfigV, cannot handle configuration errors resulting from events occurred during the system runtime. Such configuration errors are referred to as *legal misconfigurations* [23]. In particular, many parameter misconfigurations have perfectly legal types and values, but do not deliver the functionality intended by users. For example, after a website's traffic significantly increases, the parameter `Max_key_buffer` in MySQL may not be able to handle increasingly more data traffic, thus leading to the outage of the whole system. These cases are more difficult to detect by automatic checkers and may require more user training or better configuration design. A potential solution is to combine existing misconfiguration diagnosis tools, *e.g.*, X-ray [5] and ConfAid [6], with ConfigV in order to enhance the misconfiguration checking capability.

Misconfiguration across software components. As exposed by Yin *et al.* [23], cross-software configuration correlation problems also account for a considerable number of misconfiguration cases. For example, in a LAMP-based Web server, one entry in PHP configuration file, `mysql.max_persistent = 400` may make users encounter a "too many connections" error, because a correlated entry in the underlying MySQL's configuration file assigns `max_connection` to 300, which is less than the MySQL connection numbers in PHP's configuration file (*i.e.*, 400). It is quite difficult to detect such a type of tricky error through leaning approaches, because not only users or engineers are not aware of the hidden interactions [22], but also it is hard to obtain a global knowledge to the entire configurations due to the business privacy concerns of each software provider. One possible solution to this problem might be to introduce some cryptographic protocol, *e.g.*, private set intersection [11], to privately extract the overlapping entries, *e.g.*, `mysql.max_connection` in the above MySQL and PHP case, for double-checking.

Network configuration verification. ConfigV mainly focuses on software configurations, *e.g.*, MySQL and Apache, so that our approach is limited to support network configuration verification. This is because network configurations have quite different representations, format and rules from software configurations, since network configurations are typically written in more domain-specific policy languages. In fact, many network verification tools, *e.g.*, NoD [14] and Dobrescu *et al.* [8], have been proposed to check whether network configurations meet their specifications.

More complex configuration structure. The current ConfigV mainly targets key-value configuration files, but in practice many systems, *e.g.*, OpenStack [3], employ very complex configuration format and structure, which ConfigV cannot handle. Verifying such structurally complex configurations typically needs ConfigV to learn a much more sophisticated language model, which is challenging in practice. Recent efforts [15, 16] may present a possible solution on this limitation. These techniques can learn a call-graph from a training program set, and check a new program based on properties extracted from this generated call-graph. If we look a structurally complex configuration file as a program in these tools, we may be able to use a similar way to verify whether the configuration file violates any property of our interest.

9 Implementation

Ennan: [We need to describe how do we implement each module. We may need to have three paragraphs: one is for translator, one is for learner and one is for checker.]

10 Evaluations

We mainly conduct three parts of experiments to evaluate our ConfigV prototype. We want to answer several questions:

- Whether ConfigV can detect real-world configuration errors?
- How does the size of training dataset impact on the correctness of ConfigV?
- What is the rule inference accuracy?
- How long can we run the verification?

10.1 ConfigV's effectiveness

In order to evaluate the effectiveness of ConfigV, we extract 20 misconfiguration files from a MySQL dataset [20], and perform ConfigV on them. **Ennan:** [Here, we first need a table to list each configuration error with error type, description, and whether ConfigV correctly reports the issues.]

Next, we inject some errors to correct configuration files, and run ConfigV to check whether our framework can correctly detect these injected errors. **Ennan:** [We also need a table here, and list each injected problem.]

10.2 Training Set Impact

In this experiment, we vary the number of training configuration files, and observe whether the correctness of ConfigV would increase accordingly. **Ennan:** [We may need a table whose x-axis is the number of entries in the training configuration files, and y-axis is the accuracy or something close.]

10.3 The accuracy of Rule Inference

Ennan: [I am still trying to think about this part ...]

10.4 The Run-Time of Verification

Ennan: [Here, we need three pictures. In the first figure, x-axis should be the number (or the size) of entries in the training dataset, and y-axis should be the run-time of parsing. In the second figure, we need to measure the time of generating rules.]

11 Related Work

Providing language support has been considered as a promising means of tackling configuration problems [22]. Nevertheless, practical language-based misconfiguration detection approach still remains an open problem.

Configuration languages. There have been several language-support efforts proposed to prevent configuration errors introduced by fundamental deficiencies in either untyped or low-level languages. For example, in network configuration management area, it is easy for administrators to produce configuration errors in their routing configuration files. PRESTO [9] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [13] adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [7] constructs a language to describe domain knowledge about network devices and services to convenient network reasoning and management.

Compared with these existing efforts, our work mainly focused on software systems, *e.g.*, MySQL and Apache, rather than network configurations. In addition, we do not need the user of ConfigV to manually write a configuration file with the proposed language, since ConfigV can automatically parse a target configuration file into our proposed representation.

Misconfiguration detection. Misconfiguration detection techniques aim at checking the configuration efforts before the system outages occur. Most of existing detection approaches check the configuration files against a set of predefined correctness rules, named constraints, and then report the errors if the checked configuration files do not satisfy these rules.

Huang *et al.* [10] proposed a specification language, ConfValley, to validate whether given configuration files meet administrators' "belief" in mind. Different from ConfigV, ConfValley itself does not have inherent misconfiguration checking capability, since it only offers a language representation. In addition, administrators have to manually write specifications with ConfValley, which is an error-prone process; on the contrary, ConfigV does not need users to manually write anything.

Several machine learning-based misconfiguration detection efforts also have been proposed [24, 25]. EnCore [25] is the most close work to ConfigV. It introduces a template-based learning approach to improve the accuracy of their learning results. The learning process is guided by a set of predefined rule templates that enforce learning to focus on patterns of interests. By this way, EnCore filters out irrelevant information and reduces the false positives; moreover, the templates are able to express system environment information that other machine learning techniques cannot handle. Compared with

EnCore, ConfigV has the following advantages. First, ConfigV does not rely on whether the files in given configuration set are 100% correct. Second, ConfigV not only can cover much more types of misconfigurations, but also introduces probabilistic type. Finally, ConfigV is a language framework, which could even be used to write configuration files, but EnCore is only a misconfiguration detection tool.

Misconfiguration diagnosis. Many misconfiguration diagnosis approaches have been proposed [5, 6]. For example, ConfAid [6] and X-ray [5] use dynamic information flow tracking to find possible configuration errors that may result in failures or performance problems. AutoBash [17] speculatively executes processes and tracks causality to automatically fix misconfigurations. Different from ConfigV, most of misconfiguration diagnosis efforts aim at finding out errors after the system failures occur, which typically lead to prolonged recover time.

Misconfiguration tolerance. There have been several efforts proposed to test whether systems are tolerant to misconfigurations [21]. SPEX [21] takes a white-box testing approach to automatically extract configuration parameter constraints from source code and generates misconfigurations to test whether systems can tolerate the potential configuration errors.

Making systems gracefully handle misconfigurations and eliminating configuration errors are two orthogonal directions. The former helps improve the robustness of systems and make diagnosis easier. This is especially important for software that will be widely distributed to end users. Our work belongs to the latter case, which is used to prevent configuration errors before the system failures occur.

12 Conclusion

In this paper, we introduce ConfigV, a highly-modular framework that allows verification of configuration files, even without a language model of the file. New verification properties require only a small amount of code and are not language specific, all indicating that ConfigV could be widely adopted by system administrators. Such a verification tool that scales in both performance and expressivity can revolutionize configuration file checking, reducing the cost of system maintenance and failure dramatically.

References

[1] Fine-grained value correlation error. http://serverfault.com/questions/628414/my-cnf-configuration-in-mysql-5-6-x.

[2] Misconfiguration dataset. https://github.com/tianyin/configuration_datasets.

[3] OpenStack. http://www.openstack.org/.

[4] Problem moving a MySQL data directory. http://serverfault.com/questions/281217/problem-moving-a-mysql-data-directory-to-a-new-drive.

[5] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[6] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[7] Xu Chen, Yun Mao, Zhuoqing Morley Mao, and Jacobus E. van der Merwe. Declarative configuration management for complex and dynamic networks. In *ACM CoNEXT (CoNEXT)*, November 2010.

[8] Mihai Dobrescu and Katerina J. Argyraki. Software dataplane verification. In *11th USENIX Symposium on Networked System Design and Implementation (NSDI)*, April 2014.

[9] William Enck, Patrick Drew McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert G. Greenberg, Sanjay G. Rao, and William Aiello. Configuration management at massive scale: System design and experience. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2007.

[10] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. Confvalley: A systematic configuration validation framework for cloud services. In *10th European Conference on Computer Systems (EuroSys)*, April 2015.

[11] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *25th Annual International Cryptology Conference (CRYPTO)*. Springer, August 2005.

[12] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Language Design and Implementation (PLDI)*, pages 24–35. ACM Press, 1993.

[13] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *ACM SIGCOMM (SIGCOMM)*, August 2005.

[14] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI)*, May 2015.

[15] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In *43rd ACM SIGPLAN-SIGACT (POPL) Symposium on Principles of Programming Languages*, January 2016.

[16] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *42nd ACM SIGPLAN-SIGACT (POPL) Symposium on Principles of Programming Languages*, January 2015.

[17] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: Improving configuration management with operating systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[18] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[19] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[20] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Key, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, August 2015.

[21] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[22] Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4):70, 2015.

[23] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[24] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2011.

[25] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.