1 Motivation

Configuration errors are one of the most important root causes of today's software system failures [20, 21]. We read in the news that recently there was a problem in accessing Facebook and Instagram [1] and a Facebook spokeswoman that this was due to a change to the site's configuration systems. While program verification is mostly focused on detecting errors in code, in their empirical study Yin *et al.* [21] report that about 31% of system failures were caused by configuration errors problems and only 20% were caused by bugs in program code. Configuration errors are commonly known in literature under the name misconfigurations.

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 [23, 22, 19, 17].

The systems research community has recognized this as an important problem. In fact, at this year's OSDI conference (Operating Systems Design and Implementation, a top-tier system conference) a paper on detecting configuration errors [18] received the best paper award. While many efforts have been proposed to check, troubleshoot, diagnose, and repair configuration errors [5, 14, 16], those tools mainly try to understand *what* caused the error – they are still not on a level of automatic verification tools used for regular program verification [10, 12, 6] that can detect errors without executing the code.

1.1 Examples

We start by presenting two non-trivial configuration errors extracted from real-world examples. Although the errors are relatively simple, we call them non-trivial, because the majority of existing tools, *e.g.*, learning-based checking tools [23, 15], cannot detect these configuration errors. Most of the presented examples were found on StackOverflow, a popular question and answer website for programmers.

Example 1: Ordering errors. When a user configures PHP to run with the Apache HTTP Server, most likely the user will take some already existing configuration files and adapt them to suit her needs. The configuration file might contain, among others, the following lines:

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

In that case the configuration file will cause the Apache server to fail to start due to a segmentation fault error. This is because, when using PHP in Apache, the extension `mysql.so` depends on `recode.so`, and their relative ordering is crucial. This is an example of so-called *ordering error*. Yin *et al.* report that ordering errors widely exist in many system configurations, *e.g.*, PHP and MySQL, and typically lead to multiple system crash events. However, no existing tool in the systems research can effectively solve or detect this problem [23, 20, 19].

Example 2: Fine-grained value correlation error. Next example also comes from a discussion on StackOverflow [2]. The user has configured her MySQL as in 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
...
```

The user then complained that her MySQL load was very high, causing the website's response speed to be 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 this misconfiguration causes MySQL to load very slowly. This type of error is more sophisticated than the simple value correlation that some tools can detect [21, 23].

1.2 Challenges

We believe there are two main obstacles why we cannot simply apply the existing automatic tools and techniques to verification of configuration files are:

- a lack of a specification which would describe properties of configuration files
- a program structure of configuration files – they are mainly a sequence of entries assigning some value to system variables.

The language in which configuration files are written does not adhere to a specific grammar or syntax. In particular, the entries in configuration files are untyped. Moreover, there are surprisingly few rules specifying constraints on entries and there is no explicit structure policy for the entries. We believe that those are the reasons why there is no automated verification of configuration files although the system community expressed that that would be highly desirable [15, 23, 20].

1.3 Automated Verification of Configuration Files

The goal of this proposal is to develop a fully **automated verification framework for general software configurations**. We plan to overcome the above obstacles by first automatically inferring a specification for configuration files. It is unrealistic to expect the users to write a specifications for configuration files on their own. This process can easily lead to incomplete or even contradictory specifications. Instead, we will learn specification from a large sample of configuration files. We start a learning process that takes as input this sample. The process will be language-agnostic and should work for any kind of configuration files, but all of the files in the sample need to be of the same kind (such as MySQL or HTTPD configuration files). From that sample, we will learn an abundant set of rules specifying various properties that hold on the given sample. The rules, in general,

specify which properties variables in configuration files need to satisfy. One can see this learning process as a way of deriving a specification for configuration files. With these rules we can efficiently check the correctness of the configuration files of interest and detect potential errors. Errors are reported if the configuration file does not adhere to a specification.

For practical purposes we plan to use a real-world dataset [3]. The files in the sample might contain errors, but they are typically different errors and only appear in a small percentage of files. We will, therefore, use probabilistic learning to derive a set of accurate rules.

Building such an automatic verification framework for configuration files requires addressing several challenges. First, in the process of inferring a specification, we have to analyze mainly with an untyped, unstructured sequence of assignments. Thus, we need to develop a suitable language model. We do that by "guessing" a type of variables and then deriving formulas that describe relationships between these variables. However, the type of a variable cannot always be fully determined from a single value. For example, an entry `something = 1` assigns **Ruzica:** [here, something is 1 which is int, but it is actually a Boolean] **Ruzica:** [more story here] Some existing type inference work would report this is an error, because `temp_dir` should be assigned an integer [23]. We address this problem by introducing the concept of *probabilistic types*. Rather than assigning only one variable to a single type, we assign several types over a probability distribution. Using probabilistic types, we aim to generate a more accurate language model, thus significantly improving our verification capabilities.

Second, we will learn patterns describing variables and their relations. We will associate with each type a set of very general templates. The user does not need to provide any templates – they are internally associated to the types. Nevertheless, in addition to those general templates, we still need specific algorithms to learn rules that cannot be easily templated.

From a practical perspective, we do not want to introduces any additional burden to the user: 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 her own templates and learners for that class. The new learner can be added to ConfigV and this way it can check an additional new set of errors.

Our ConfigV prototype still has many limitations: for example, we cannot handle configuration errors that can be triggered during 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 a sample dataset, and then use this language model to verify configuration files of interest.
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.
4. ConfigV is capable of detecting various tricky errors that cannot be detected by previous efforts, including entry ordering errors, fine-grained value correlation errors, missing entry errors, and environment-related errors.
5. We implement a ConfigV prototype and evaluate it by conducting comprehensive experiments on real-world dataset.

2 Our Previous Work: ConfigC

We have proposed and developed a preliminary system, named ConfigC [13], which is (to the best of our knowledge) the first systematic effort capable of automatically verifying configuration files before the configuration files are deployed and installed. In particular, ConfigC is a framework that first automatically analyzes datasets of correct configuration files, and then derives rules for building a language model from the give datasets. Finally, the resulting language model could be used to verify new configuration files (called target configuration files) and detect errors in these target configuration files. The configuration errors ConfigC can deal with cover: entry missing errors, ordering errors, type errors and value correlation. To our knowledge, there is no existing effort that is able to detect the above errors, because they are very tricky to identify in practice.

ConfigC can detect the above errors by overcoming main technical challenges against automatic configuration verification. First, because writing specifications for configuration verification is difficult – especially for specifying the above tricky configuration errors, *e.g.*, missing entry and ordering errors, ConfigC employs a set of machine learning algorithms to automate the specification writing. Machine learning based approach employed by ConfigC enables the process of specification writing to become automatic. As long as a set of configuration files are provided, ConfigC can automatically generate a set of rules that could be used as specifications. Second, because existing configuration files do not have any language structures and grammar, it is difficult to verify configuration files. ConfigC proposes a new language model and transforms target configuration files into the proposed language model. In other words, ConfigC uses the language model as a uniform representation and parses the configuration files to verify into such a representation. With the uniform representation and specifications in hand, ConfigC is able to verify whether the given configuration files meet the specification.

Evaluations on ConfigC. We implemented a prototype system by following our ConfigC design [13]. In order to demonstrate the capability of ConfigC, we evaluated the ConfigC prototype based on real-world MySQL configuration files [3] containing tricky misconfiguration errors, such as entry missing errors, ordering errors, and value correlation errors. We extracted 20 MySQL configuration files to detect, and grouped them into four categories according to their error types:

missing entry, type error, ordering error and value correlation. Table 1 shows the evaluation results we use ConfigC to detect configuration errors in these 20 real-world configuration files. As shown in Table 1, ConfigC is able to successfully report the misconfiguration problems.

Table 1: Evaluation results on misconfiguration detection of ConfigC

Error Type	Passing Tests	False Positives
Missing Entry	5/5	1, 0, 0, 0, 4
Type Error	5/5	0, 0, 0, 0, 0
Ordering Error	5/5	0, 2, 1, 0, 6
Value Correlation	4/5	0, 0, 0, 1, 0

Limitations. ConfigC still has the following limitations.

1. ConfigC requires the datasets of configuration files for training have to be correct, which is hard to achieve in practice, since it is difficult for administrators or users to offer a set of 100% correct configuration files for training.
2. The language model of ConfigC is still a starting point. It is hard to formulate a uniform representation for diverse configuration files.**Ennan:** [we still need to put one more sentence here to distinguish why our current language model is better than ConfigC's language model]
3. ConfigC can only offer coarse-grained value correlation errors, *e.g.*, `max_connections` > `mysql.max_persistent`. In practice, many software outages were caused by more tricky, called fine-grained value correlation [2]. For example, in a MySQL configuration file, `key_buffer_size` should be higher than `max_connections`× `sort_buffer_size` [2]; otherwise, the MySQL load will be very high.
4. ConfigC cannot check singular value anomalies. For example, a parameter relating to memory in MySQL configuration files is setted too large, exhausting the RAM and causing extreme slowness or even a crash. Such an anomalous value does not violate any constraints (*e.g.*, type error and value ordering error), but it makes system behave wrongly.

This proposal aims to address the above limitations.

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 errors) that previous efforts cannot detect. As depicted in Figure 1, a typical ConfigV verification workflow has three steps: translation, learning, and checking. In this section, we briefly describe how each step works.

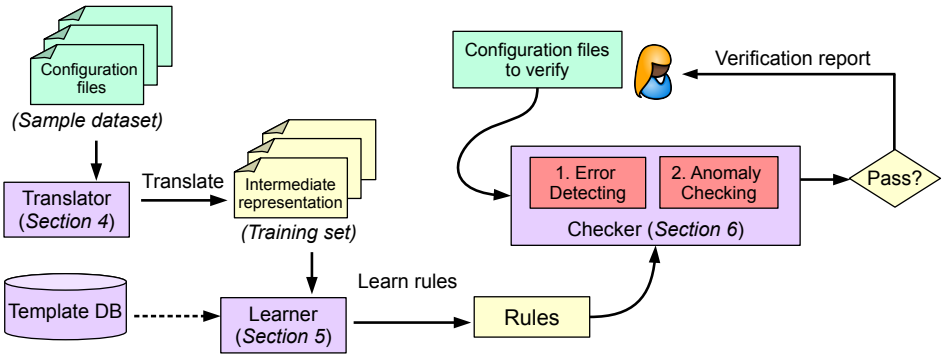

Figure 1: ConfigV's workflow. The green components represent configuration files, including both sample configuration datasets and users' input configuration files to verify. The purple components are the modules of ConfigV. Because template DB is not necessarily used, we use dashed arrow between it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

Initial phase. We start with the assumption that we are given a number of (not necessarily correct) configuration files, called *sample dataset*, belonging to the same system, such as MySQL or Apache. These files, therefore, follow similar patterns.

Translator. The translator module first parses the input sample dataset (containing both configuration files and system environment information), and then transforms them into a more structured and typed intermediate representation. When we infer the types of entries in a configuration file, the type of an entry cannot always be fully determined from a single value, since it is very hard to understand the purposes of key-value entries in modern software configuration files [17]. We address this problem by introducing *probabilistic types*. In particular, rather than giving a variable a single type, we assign several types over a probability distribution. We can later use these more structured files as a training set to learn the rules.

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

Although the learner does not necessarily rely on templates, ConfigV still offers a database containing many templates, as shown in Figure 1. Some of these

templates are responsible for offering specific system executional environment information. A learning algorithm cannot derive rules related to environment violations, *e.g.*, whether the current account is the owner of a certain path, without information about the environment. 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 rule violations in the configuration files of interest. The inputs of the checker are the learned rules and the target configuration file to verify. It generates a report (as shown in §**??**) about whether it finds any error, *e.g.*, rule violations or some suspicious values (*e.g.*, singular value anomaly). As shown in Figure 1, there are two sub-modules in the checker. They are responsible for checking rule violations and suspicious values, respectively. In our experience, we found learned rules could be significantly reused to check different configuration files, thus improving our usability.

4 Language Model

The translator takes as input a sample dataset of configuration files and transforms it into another set of files written in typed and well-structured representations. The translator can be seen a parser, and it is only used to generate an intermediate representation for post-processing, such as learning rules (see §**??**). 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 configurations, but it is not required.

The majority of entries in a configuration files are assignments. Our first attempt was to simply translate every key-value entry `k = v` into a triple (k, v, τ), where τ is a type of v. However, sometimes we could not fully determine the type of key based on a single example value. For this reason, we introduced *probabilistic types*.

Consider the following example.

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

Most likely `foo` should be an integer, but it could also be a string. In the second case, we can learn the rule stating `foo` $\in$ `substrings(bar)`. Instead of assigning one type to a value, the translator assigns a distribution of types to a value, an idea closely related to existentially quantified types [**?**].

Formally, we define probabilistic types as follows: let $\mathcal{T}$ be a set of basic types (cf. Table 2). A probabilistic type built from $\mathcal{T}$ is a list of pairs $[(\tau_1, p_1), \ldots, (\tau_n, p_n)]$, such that $\tau_i \in \mathcal{T}$, $0 \leq p_i \leq 1$ and $\Sigma p_i = 1$. These probabilities are updated each time a new example value for a key is encountered.

Table 2: Table of types, along with associated templates, used in ConfigV

Type	Template
Integer	$X = Y, X \neq Y, X \leq Y, X < Y, X * Y < Z, \ldots$
String	$\mathsf{substr}(U, V), \mathsf{prefix}(U, V), \mathsf{suffix}(U, V), \ldots$
File Path	$\mathsf{isFile}(F), \mathsf{isDir}(D), \ldots$
Size	Similar to integers
Port	$P_1 > P_2$
IP Addr.	$\mathsf{sameSubnet}(addr1, addr2)$

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.

In ConfigV, the set $\mathcal{T}$ contains strings, integers, file paths, sizes, and IP addresses. With every type, we associate a list of templates that are used to learn the rules about those files. Table 2 contains the most important types and templates.

In general, the templates works as follows: if there are two entries `k1` = `v1` and `k2` = `v2`, let $t(X, Y)$ be a template that can be applied to those values (w.r.t. their types). If $t(v1, v2)$ holds, *i.e.*, the template condition is valid for those concrete values, then we will add a rule $t(k1, k2)$ to a list of potentially correct specifications. Note that the rule expresses a relation between variables.

With the help of probabilistic types, we addressed ambiguities during the parsing phase. Another problem that we need to solve is that configuration files sometimes contain a simple version of the `if-then`, such as in Apache HTTPD, that sets conditions for which an action should be applied. Therefore, we also add this guard to our entry during the parsing phase. We first check if the guard is true before deriving a rule; otherwise, we skip the entry.

To summarize, the translator translates every entity `k` = `v` in a configuration file into a quadruple entry

$$(g, k, v, \mathsf{List}[p_1 : \tau_1, \ldots, p_n : \tau_n])$$

For an entry e, we denote the corresponding guard by $g(e)$, the key by $k(e)$, the value by $v(e)$, and the probabilistic type list by $typelist(e)$.

5 Learning More Accurate Specification

Ruzica: [this is actually an OSDI summary and learning from incorrect files]

6 Extending ConfigV Via Version Space Learning

The current design of ConfigV does not have any assumptions on whether the configuration files in the training sets are correct or incorrect, thus making ConfigV more general and practical. Nevertheless, some cases in practice may allow

us to assume some of configuration files in the training sets are guaranteed to be correct. For example, TravisCI is a famous system connected to Github, and it allows programmers to automatically run their test suites on every code commit. Thus, a TravisCI user needs to add a configuration file to the repository that enables TravisCI and specifies build conditions, such as which dependencies are required, and a set of benchmarks to test. This ensures the tool to be automatically built correctly on a fresh machine. Such a configuration file management scheme, in fact, guarantees the configuration file versions in the commit history to be correct – otherwise, TravisCI should not work correctly in the past. **Ennan:** [Need one or two sentences to describe TravisCI misconfiguration problem is important.]

Driven by the above motivating example, we propose an extension to ConfigV that learns a given training set with pre-knowledge that this training set contains some configuration files guaranteed to be correct. Such an extension employs a new algorithm, named version space learning, to significantly decrease the false positive rate while maintaining the detection capability of ConfigV.

6.1 Version Space For Verification

Version space learning builds a logical constraint model for binary classification, which we use to test a configuration file for membership in the set of all correct files [**?**]. Traditional version space learning builds a model that tests membership using a series of disjunctions from a set of predefined hypotheses. In the extended ConfigV, rather than describing a correct file by allowable traits, we describe the required traits. This way, our system can not only flag misconfigurations, but also give the points of failure for non-membership. We now show in detail how this process works.

Our definition of a configuration file, C, is a file that can be transformed to an ordered list of $(keyword, value)$ pairs, called $Line$s. ConfigV builds a model for a single file, $M(C)$, that is the set of all possible relations between the lines in the file. A relation is described by user provided templates. A template is a function taking some number of lines and determining a relation on the keywords of the lines ($Line^n \rightarrow Rel_{k1,..,kn}$). For example, we may derive the relation `extension mysql.so` comes before `extension recode.so` by using the ordering template. ConfigC features the following default templates.

Template	Input	Relation
ordering	(Line,Line)	Before | After | None
integer relation	(Line,Line)	$<$ | $==$ | $>$
type	(Line,Line)	String | Int | Filepath | IP
missing entry	(Line,Line)	Required | Not

It is assumed if a file is correct, all relations in that file are in the set of necessary relations for any other file to be correct ($Correct(C) \implies \forall r \in M(C), r \in Nec$). In this way, the initial model is built by creating the strongest conditions for a correct file, called the *specific boundary* in version space learning. This model is then iteratively relaxed as more examples are seen, a process called *candidate elimination*. Since we maintain the strongest condition for correctness, this

approach will identify many correct files as incorrect, which we call a high false positive rate. This is a problem when a user is then asked to manually review many files for errors, when the files are in fact correct.

6.2 Learning From Temporal Properties

We now present a new algorithm to decrease the false positive rate by learning on both correct and incorrect examples, as well as temporal structure of these examples. This extension will build a logical formula representing the entire history of examples, and use an SMT solver to find a classification model.

Since a configuration file (*e.g.*, TravisCI configuration file) is dependent on the code it is trying to build, we must consider a more general sense of configuration file. We will call this a program summary P_t, which is a representation of the repository which contains the information relevant to the learning process. The subscript on P_t is a timestamp tag based on the ordered git commit history. In the case of TravisCI, this include the `.travis.yml` file, as well as code features that may effect build status, such as programming language and a list of imported libraries. The summary must be *sufficiently detailed*, that is it must contain every piece of information that might lead to a build error.

From this summary we can then build a model $M(P_t)$, as in ConfigV, which is the full set of possible relations derivable from the program summary. Again, the user must provide templates for the learning process.

We will denote the build status returned from TravisCI when run on P_t with $S(P_t)$. Any files with a $Pass$ build status are correct, and any files with an Err build status are incorrect. For brevity, we denote sequences of build statuses with the following notation:

$$S(P_t) = Pass \wedge S(P_{t+1}) = Err \implies S(P_{t,t+1}) = PE$$

We now consider both incorrect and correct and so must introduce the *general boundary*. The general boundary is the dual of the specific boundary, and is the most relaxed requirement for a positive classification. We denoted specific boundary as the set of necessary relations Nec, and now denote the general boundary as the set of breaking relations Br. With this notation, we can formally express the requirement that the program summary is sufficiently detailed.

$$\forall S(P_t) = Err, \exists r \in M(P_t), r \in Br \tag{1}$$

From the above we know that if a build is erroring, then there must exist at least one error. We can also know that if a build is passing, then there must not exist any errors. That is, the model of a passing commit must not contain any rules which are breaking. Note we are not, however, guaranteed that any rules from a passing commit are necessary.

$$S(P_t) = Err \implies \exists r \in M(P_t), r \in Br \tag{2}$$

$$S(P_t) = Pass \implies \forall r \in M(P_t), r \notin Br \tag{3}$$

Additionally, when we commit a break (PE), we can localize the error to one of the relations that changed. Either we removed something that was necessary, or added something that was breaking. Note that this is an inclusive disjunction, since a erroring commit can break multiple things at once. Expressed formally, where $\setminus$ is the set difference, that is:

$$\begin{aligned} S(P_{t,t+1}) = PE \implies \\ \exists r \in (M(P_t) \setminus M(P_{t+1})), r \in Nec \\ \vee \exists r \in (M(P_{t+1}) \setminus M(P_t)), r \in Br \end{aligned} \tag{4}$$

We then can combine Eq 2, 3, and 4 with conjunctions and ask a SMT solver for a model satisfying the formula. The resulting model will be the sets Nec, and Br, which can be used to check new configuration files. Since we used a similar model to ConfigV, we will still be able to provide justifications for the classification results.

7 Related Work

Language support has been considered a promising way to tackle configuration problems [20]. Nevertheless, a practical language-based misconfiguration detection approach still remains an open problem.

Configuration languages. There have been several language support efforts proposed for preventing configuration errors introduced by fundamental deficiencies in either untyped or low-level languages. For example, in the network configuration management area, it is easy for administrators to produce configuration errors in their routing configuration files. PRESTO [8] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [11] adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [7] constructs a language to describe domain knowledge about devices and services for 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.

Huang *et al.* proposed a specification for configuration validation [9]. We will discuss this work in the misconfiguration detection part.

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

Huang *et al.* [9] proposed a language, ConfValley, to validate whether given configuration files meet administrators' specifications. Different from ConfigV,

ConfValley does not have inherent misconfiguration checking capability, since it only offers a language representation and requires administrators to manually write specifications, 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 [22, 23]. EnCore [23] is the work closest 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 interest. In this way, EnCore filters out irrelevant information and reduces false positives; moreover, the templates are able to express system environment information that other machine learning techniques cannot handle. Compared to EnCore, ConfigV has the following advantages. Firstly, ConfigV does not rely on 100% correctness in the files of the given configuration set. Secondly, ConfigV not only covers many more types of misconfigurations, but also introduces probabilistic types. Finally, ConfigV is a language framework, which can 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, 4]. For example, ConfAid [5] and X-ray [4] use dynamic information flow tracking to find possible configuration errors that may result in failures or performance problems. AutoBash [14] tracks causality and automatically fixes misconfigurations. Unlike ConfigV, most misconfiguration diagnosis efforts aim at finding errors after system failures occur, which leads to prolonged recovery time.

Misconfiguration tolerance. There have been several efforts proposed to test whether systems are tolerant to misconfigurations [19]. SPEX [19] takes a white-box testing approach to automatically extract configuration parameter constraints from source code and generates misconfigurations to test whether systems can tolerate 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 system outages.

Bibliography

1 Facebook, Tinder, Instagram suffer widespread issues. `http://mashable.com/2015/01/27/facebook-tinder-instagram-issues/`.

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

3 Misconfiguration dataset. `https://github.com/tianyin/configuration_datasets`.

4 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.

5 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.

6 François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with why3. *STTT*, 17(6):709–727, 2015.

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 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.

9 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.

10 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, pages 348–370, 2010.

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

12 Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper - complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, pages 124–139, 2014.

13 Mark Santolucito, Ennan Zhai, and Ruzica Piskac. Probabilistic automated language learning for configuration files. In *28th Computer Aided Verification (CAV)*, July 2016.

14 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.

15 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.

16 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.

17 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.

18 Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

19 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.

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

21 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.

22 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.

23 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.