# Synthesizing Configuration File Specifications with Association Rule Learning

**Abstract.** System failures resulting from configuration errors are one of the major reasons for compromised reliability of today's software systems. Although many techniques have been proposed for configuration error detection, these approaches mainly can only be applied after an error has occurred. Verifying configuration files is, nevertheless, a challenging problem, because 1) software configurations are typically written in poorly structured and untyped "languages", and 2) specifying rules for configuration verification is challenging in practice. This paper presents VeriConf, a verification framework for general software configurations. Our framework works as follows: in the pre-processing stage, we first automatically derive a specification. Once we have a specification, we check if a given configuration file adheres to that specification. The process of learning specification works through three steps. First, VeriConf parses a training set of configuration files (not necessarily all correct) into a well-structured and probabilistically-typed intermediate representation. Second, based on the association rule learning algorithm VeriConf learns rules from these intermediate representations. These rules establish relationships between the keywords appearing in the files. Finally, VeriConf employs rule graph analysis to refine the resulting rules. VeriConf is capable of detecting various configuration errors, including ordering errors, integer correlation errors, type errors, and missing entry errors. We evaluated VeriConf by verifying public configuration files on Github, and we show that VeriConf can detect known configuration errors in these files.

## 1 Introduction

Configuration errors (also known as misconfigurations) have become one of the major causes of system failures, resulting in security vulnerabilities, application outages, and incorrect program executions [43,45,46]. Recently, there was a problem in accessing Facebook and Instagram [6], and a Facebook spokeswoman reported that this was caused by a change to the site's configuration systems. In a recent software system failures study [47], researchers revealed that about 31% of system failures were caused by configuration errors, which is higher than the percentage of failures resulting from program bugs (20%).

In this paper we propose a framework for automated verification of configuration files: we derive specifications by learning "raw" configuration files and proactively report potential errors in the configuration files of interest, without waiting for them to happen. We developed a tool, called VeriConf, and evaluated it on almost a thousand configuration files from Github. The systems research community has recognized this as an important problem [44]. While many efforts have been proposed to check, troubleshoot, and diagnose configuration errors [26, 40, 42], those tools heavily rely on analyzing the source code of the target systems and need to run these systems multiple times to understand where are the errors—*i.e.*, they are still not on a level of automated verification tools used for regular program verification [27,35,37] that can detect errors without executing the code.

We believe there are two main obstacles to why we cannot simply apply the existing automatic program verification techniques to verification of configuration files: 1) the lack of a specification describing properties of configuration files; 2) the structure of configuration files – they are mainly a sequence of entries assigning some value to system variables (called *keywords*). 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.

VeriConf overcomes the above obstacles by first automatically inferring a specification for configuration files. It is unrealistic to expect the users to write an entire specification for configuration files on their own. This process can easily lead to incomplete or even contradictory specifications. Instead, we learn specification from a large training set of configuration files [11]. The first step is to translate this training set into a more structured typed representation. We then apply the learning process and we learn an abundant set of rules specifying various properties that hold on the given training set. The rules, in general, specify which properties keywords in configuration files need to satisfy. The learning process is language-agnostic and works for any kind of configuration files, but all of the files in the training set need to be of the same kind (such as MySQL or HTTPD configuration files). We see this learning process as a way of deriving a specification for configuration files. It is hard to talk here about a complete specification, but it is still a set of formal rules describing the properties that configuration files need to satisfy.

Having a specification, VeriConf can then 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 the derived specification.

The errors found can cause total system failures, but can also be more insidious, for example slowing down the system only when the server load increases beyond a certain threshold. Since these runtime errors may only be triggered after some time in a deployment environment, the standard debugging techniques [49] of starting a system multiple times with different configuration settings will not help detect these misconfigurations.

To the best of our knowledge, there have been several prior efforts that attempt to verify configuration files [31, 39, 44, 50]. However, state-of-the-art efforts either are impractical to be used in reality, or can only deal with simplistic configuration errors. In general, these efforts fall into two categories.

- On the one hand there are tools that can detect sophisticated configuration errors, *e.g.*, ConfigC [39]. However, these tools heavily rely on datasets containing 100% correct configuration files to extract configuration rules. Existing investigation studies [41,47] have demonstrated determining or obtaining 100% correct configuration files to drive rules is almost impossible in reality. Without 100% correct datasets, these tools do not work.
- On the other hand there are tools, *e.g.*, EnCore [50] and ConfValley [31], that can only detect rather simplistic configuration errors (*e.g.*, value range errors and simple integer correlation errors), but cannot detect more complex configuration errors, such as ordering errors or Nevertheless, these types of errors cannot be detected by above existing efforts.

Our learning algorithm surmounts all of difficulties present in existing tools. VeriConf implements a learning algorithm inspired by *association rule learning* [24]. It analyzes a large training set of configuration files [11]. Those are real-world reported misconfigurations. A file in the training set might contain several different errors and these errors only appear in a small percentage of files. We first translate those file into an intermediary typed language. With with every type we associate a set of very general interfaces. An VeriConf's user does not need to provide them, they are internally associated to the types. The learner then instantiates those interfaces with the keywords appearing in the training set. However, since files might also contain errors we take this into account when learning correct rules. All learned rules are annotated with the probability of correctness. To ensure the learned rules are good enough, VeriConf employs a further analysis to refine the learned rules. In general, VeriConf builds a graph modeling the learned rules, and then analyzes this graph to rank the importance and relevance of the learned rules.

We have implemented a tool, VeriConf, and evaluated it on almost 1000 real- world configuration files from Github. We demonstrate that we are able to detect known errors, based on StackOverflow posts, in this data set. Furthermore, we find compelling evidence that out optimizations are effective, for example using probabilistic types removes 1023 false positives errors reports to reduce the total error reports to 324. The rule graph analysis drastically improves the ranking of importance of the error reports, which allows users to more quickly fix the most critical misconfigurations. Additionally, VeriConf scales linearly, whereas a previous tool [39] showed exponential slow downs on the same benchmark set.

In summary, we make the following contributions:

- We propose the automated configuration verification framework, VeriConf, that can learn specifications from a training set of configuration files, and then use the specifications to verify configuration files of interest.
- We describe the logical foundation of using association rule learning to build a probabilistic specification for configuration files.
- We analyze the learned rules to further refine the generated specification and we empirically show the usefulness of this approach.
- We implement a VeriConf prototype and evaluate it by detecting sophisticated configuration errors from real-world dataset.

## 2 Motivating Examples

In this section we illustrate capabilities of VeriConf by using several real-world misconfiguration examples. These examples are sophisticated configuration errors that were reported on StackOverflow [15], a popular forum for programmers and administrators.

**Example 1: Missing Entry Errors.** Many critical system outages result from the fact that an important entry was missing from the configuration file. We call such a problem a *missing entry error*. In a public misconfiguration dataset [11], many misconfiguration issues were reported exactly to be missing entry errors. Below is a real-world missing entry error example [13]: when a user configures her PHP and PostgreSQL, she needs

to use both `pgsql.so` and `curl.so` in the `/ctc/php5/conf.d/curl.ini` configuration file. This is usually achieved by the following entries in the curl configuration file:

```
extension=pgsql.so
extension=curl.so
...
```

However, in this example the user accidentally left out the `extension=pgsql.so` entry, as done by many users [13,47], causing a segmentation fault. If the user would run VeriConf on her file, our tool returns:

```
MISSING ENTRY ERROR: Expected "extension=pgsal.so"
in the same file: "extension=curl.so"
```

**Example 2: Fine-grained Integer Correlation Errors.** Our second misconfiguration example [8] comes from a discussion on StackOverflow. The user has configured her MySQL as follows:

```
max_connections             = 64
thread_cache_size           = 8
thread_concurrency          = 8
key_buffer_size             = 4G
max_heap_table_size         = 128M
join_buffer_size            = 32M
sort_buffer_size            = 32M
```

The user then complains that her MySQL load was very high, causing the website's response speed to be very slow. The accepted answer to the post reveals that the value `key_buffer_size` is used by all the threads cooperatively, while `join_buffer` and `sort_buffer` are created by each thread for private use. By further consulting the MySQL manual, we are instructed that when setting `key_buffer_size` we should consider the memory requirement of other storage engines. In a very indirect manner, we have learned that there is a correlation between `key_buffer_size` and other buffer sizes of the system. VeriConf learns a specific constraint, *i.e.* `key_buffer_size` should not be greater than `sort_buffer_size` * `max_connections`. If we run VeriConf on the above configuration file, VeriConf will give an explicit answer:

```
FINE GRAINED ERROR: Expected
  "max_connections" * "sort_buffer_size" > "key_buffer_size"
```

We call these errors as *fine-grained integer correlations*. VeriConf can also detect simpler integer correlation: one entry's value should have a certain correlation with another entry's value. For instance, in MySQL, the value of `key_buffer` should be larger than `max_allowed_packet`. While several existing tools [39,50] can detect simple integer correlation errors, VeriConf is, to the best of our knowledge, the first system capable of detecting such complex fine-grained integer correlation errors.

**Example 3: Type Errors.** Many system availability problems are caused by assigning incorrect type of values. Consider the following misconfiguration file from github [19]: a user tries to install MySQL and she needs to initiate the path of the log information

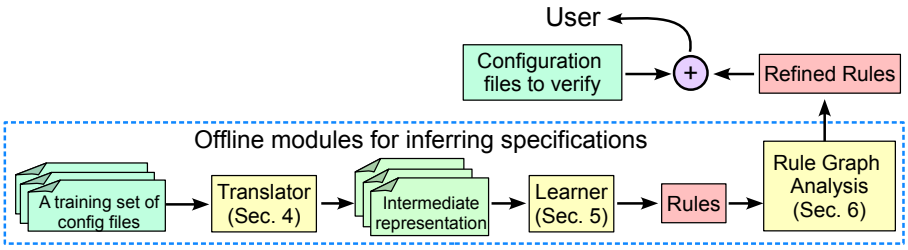


Fig. 1: VeriConf's workflow. The dashed box is the specification learning module. The yellow components are key modules of VeriConf.

generated by MySQL. This user puts the following entry assignment in her MySQL configuration file:

```
slow-query-log = /var/log/mysql/slow.log
```

This misconfiguration will lead to MySQL fails to start [18]. With VeriConf, this user can get the following result:

```
TYPE ERROR: Expected an integer type for "slow-query-log"
```

This was indeed the error, since in MySQL there is another entry named "slow-query-log-file" used to specify the log path.

**Example 4: Ordering Errors.** Ordering errors in software configurations were first reported by Yin *et al.* [47], but not many existing tools can detect them. The following example contains an ordering error in a MySQL configuration file that causes the system to crash [7].

```
innodb_data_file_path          = ibdata1:10M:autoextend
innodb_data_home_dir           = /var/lib/mysql
innodb_flush_log_at_trx_commit = 1
innodb_lock_wait_timeout       = 50
```

By invoking VeriConf the user receives a correct report that `innodb_home_dir` should appear before `innodb_data_file_path`, as shown below:

```
ORDERING ERROR: Expected "innodb_data_home_dir[mysqld]" BEFORE
  "innodb_data_file_path[mysqld]"
```

## 3 The VeriConf Framework Overview

Figure 1 gives an overview of the VeriConf framework. The main part is dedicated to learning and inferring the specification for configuration files. This process is done offline, before the user even starts to use VeriConf. There are three main steps in the process: translation, learning, and rule refinement.

**Translator.** The translator module first parses the input training set of configuration files and transforms them into a typed intermediate representation. Entries in a configuration file follow a key-value pattern, where some environmental variable ("key") is

assigned a value. However, it is not always possible to fully determine the type of the key by inspecting the value at a single entry [43]. We address this problem by introducing *probabilistic types*. Rather than giving a variable a single type, we assign several types over a probability distribution that can later be resolved to type upon which we will learn rules.

**Learning.** The learner converts the intermediate representation from the translator to a set of rules. It employs variation on the *association rule algorithm* [24], to generate this list of rules, which describe properties of a correct configuration file. This is a probabilistic verification approach that learns a specification for a correct file over an unlabeled training set of both correct and incorrect configurations files. The learning algorithm uses various instances of a rule interface to learn different classes of rules, such as ordering or integer relations. These rules are then considered to be required for any configuration files to be correct, and can be used for verification.

**Rule Graph Analysis.** Finally, the logically structured representation of learned rules allows for a further *rule graph analysis*. The purpose of this module is to refine the learned rules. To this end, we introduce the concept of a rule graph that can be built from the output of the modified association rule learning algorithm. We analyze the properties of this graph to construct a ranking of rules by their importance, as well as to produce a measure of complexity for any configuration of the target system. While the metrics in used in VeriConf are effective, they are not intended to be exhaustive. The information contained in the structured representation of the learned rules, is a unique benefit of the learning algorithm, that has potential to be leveraged in many new ways.

## 4 Translator

The translator takes as input a training set of configuration files and transforms it into a typed and well-structured intermediate representation. The translator can be seen as a parser used to generate an intermediate representation for the learner module (cf. Sec. 5). Translating or parsing is system dependent since each configuration language (MySQL, Apache, PHP) uses a different grammar. VeriConf allows users to provide extra help to the translator for their specific system configurations.

The translator converts each key-value assignment $k = v$ in the configuration file to a triple $(k, v, \tau)$, where $\tau$ is the type of $v$. There are two major challenges in this step. First is that configuration files' keywords are not necessarily unique and may have some additional context (modules or conditionals). To solve this, we rely on the fact that keywords in a configuration file must be unique within their context, and rename all keywords with their context. The set of unique keys, $\mathcal{K}$, for the sample training set in Figure 2 would then be [“`foo[server]`”,“`bar[client]`”].

**Probabilistic Types.** An additional challenge is that it is not always possible to fully determine the type of key based on a single example value. For this reason, we introduce *probabilistic types*, as contrasted with *basic types*. In VeriConf, the set of basic types contains strings, file paths, integers, sizes, and Booleans. Taking the configuration 1.1, we can assume `foo` is a Boolean type by the grammar of MySQL, but the keyword `bar` could be many types. If we choose the type based on the first example, `bar` will be a

Listing 1.1: file1.cnf

```
[server]
foo = ON
[client]
bar = 1
```

Listing 1.2: file2.cnf

```
[server]
foo = ON
[client]
bar = ON
```

Listing 1.3: file3.cnf

```
[server]
foo = OFF
[client]
bar = OFF
```

Fig. 2: A sample training set of configuration files

integer type. If we choose a type that fits all examples, `bar` will be a string. However the correct classification needed is a Boolean type. Let us assume that there is a critical rule we must learn that the Boolean keywords should have the same values, $eq(\texttt{foo}, \texttt{bar})$. If we take `bar`:: $int$, we do not learn the above rule, nor do we learn this rule with `bar`:: $string$ - only with `bar`:: $bool$ is the rule is valid. To resolve this ambiguity, and choose the best type, the translator assigns a distribution of types to a keyword based on examples from the training set of configuration files (denoted $\mathcal{TR} = \{C\}$).

A probabilistic type is a set of counts over a set $\mathcal{T}$ of basic types. Formally, we define a space of probabilistic types $\tilde{\mathcal{T}}$, where $\tilde{\tau} \in \tilde{\mathcal{T}}$ has the form $\tilde{\tau} = \{(\tau_1, c_1), \ldots, (\tau_n, c_n)\}$, such that $\tau_i \in \mathcal{T}$, $c_i \in \mathbb{Z}$. Figure 3 provides a calculus for probabilistic types over an example subset of basic types. This allows us to give a clear definition of the set of possible (*i.e.*well-typed) equality rules.

Every unique keyword $k \in \mathcal{TR}$ has a probabilistic type, expressed $k : \tilde{\tau}$, as opposed to the basic type notation $k :: \tau$. The count for $(\tau_i, c_i) \in \tilde{\tau}$ should be equal to the number of times a key in $\mathcal{TR}$ has a potential match to type $\tau_i$. The judgment PTYPE counts, over all files $C \in \mathcal{TR}$, the times the value of a key matches a user defined set of acceptable values for each type $\tau_i \in \tilde{\tau}$. We use the notation $\tilde{\tau}[\tau_i = N]$ to create a probabilistic type with the count of $N$ for $\tau_i \in \tilde{\tau}$.

In the BOOL judgment, a keyword with a probabilistic type $k : \tilde{\tau} \in \tilde{\mathcal{T}}$ can be resolved to the basic type $bool$ when the $\tilde{\tau}$ satisfies the resolution predicate $p_{bool}$, *i.e.* the probabilistic type has sufficient evidence. The definition of sufficient evidence must be empirically determined by the user depending on the quality of the training set.

$$\frac{\begin{array}{c} c_{int} = |\{\forall C \in \mathcal{TR}.\, \forall (k, v) \in C.\, v \in \mathbb{Z}\}| \\ c_{bool} = |\{\forall C \in \mathcal{TR}.\, \forall (k, v) \in C.\, v \in \{0, 1, ON, OFF\}\}| \end{array}}{k : \tilde{\tau}[int = c_{int}, bool = c_{bool}]}\ \text{PTYPE}$$

$$\frac{k : \tilde{\tau} \qquad p_{int}(\tilde{\tau})}{k :: int}\ \text{INT} \qquad \frac{k : \tilde{\tau} \qquad p_{bool}(\tilde{\tau})}{k :: bool}\ \text{BOOL} \qquad \frac{k_1 :: \tau \qquad k_2 :: \tau}{eq(k_1, k_2) :: Rule}\ \text{EQ\_RULE}$$

Fig. 3: Type judgments for a probabilistic type system with $\mathcal{T} = \{bool, int\}$ and an equality rule

In order to define resolution predicates, we use the notation $|\tau_i \tilde{\tau}|$ to select $c_i$ from a $\tilde{\tau} \in \tilde{\mathcal{T}}$. As an example, for the sample training set provided in Figure 2, we might

choose to set $p_{bool}(\tilde{\tau}) = |bool\tilde{\tau}| \geq 3 \wedge |int\tilde{\tau}| \leq 1$ and $p_{int}(\tilde{\tau}) = |int\tilde{\tau}| \geq 3$. We then can run inference on that sample training set to derive a probabilistic type `bar`: $[bool = 3, int = 1]$. This is then resolved, as required in our earlier example, to `bar`:: $bool$.

Note that a user may pick predicates for probabilistic type resolution that result in overlapping inference rules. For example, if a user instead picked $p_{int}(\tilde{\tau}) = |int\tilde{\tau}| \geq 1$, INT would overlap with BOOL. To resolve the ambiguity in this case, we must add a new rule INTBOOL that introduces a new, internal type $intbool$ to be created when both resolution predicates are true. The antecedent statement of INT must also be updated to $p_{int}(\tilde{\tau}) \wedge \neg p_{bool}(\tilde{\tau})$, and likewise with BOOL. The $intbool$ type is only a placeholder to be used with the additional subtyping relations $intbool <: int$ and $intbool <: bool$ These subtype relations allow the $intbool$ to take the place of either $int$ or $bool$ when determining if two keywords may be compared in the EQ_RULE.

$$\frac{k : \tilde{\tau} \qquad p_{bool}(\tilde{\tau}) \qquad p_{int}(\tilde{\tau})}{k :: intbool}\ \text{INTBOOL} \qquad \frac{k1, k2 \in C \qquad k1 \neq k2}{ord(k1, k2) :: Rule}\ \text{ORDER\_RULE}$$

In the case that there is not enough evidence to resolve a probabilistic type to a basic type, no type-dependent rules may be learned over that keyword. However, we are still able to learn rules such as ORDER, which do not require any resolved type.

## 5 Learner

The goal of the learner is to derive rules from the intermediate representation of the training set generated by the translator. We describe an interface to define the different classes of rules that should be learned. Each instance of the interface corresponds to a different class of configuration errors, as described in Sec. 2.

To learn rules over sets of configuration files, we use a generalization of *association rule learning* [24], a technique that can be summarized as inductive machine learning. Association rule learning is a technique to learn how frequently items of a set appear together. For example, by examining a list of food store receipts, we would learn that when a customer buys bread and peanut butter, the set of purchased items is also likely to include jelly. Since configuration files have complex relations, we extend these association relationships to generalized predicates.

A *rule*, $r$, is then an implication relationship, $r = S \Rightarrow p(S, T)$, between two possibly empty sets of keywords $S, T \subset \mathcal{K}$, which we call the source and target keyword sets. The set $\mathcal{K}$ is the set of unique keys from the training set (denoted $\mathcal{TR}$) and the predicate $p$ is one of the classes of configuration errors. Implicitly we interpret a rule to mean that if the keywords $S \cup T$ appear in a configuration file, the predicate $p$ should hold. The task of the learning algorithm is to transform a training set to a set of rules, weighted with *support* and *confidence*. The set of rules learned from training set $\mathcal{TR}$ constitutes a specification for a configuration file to be considered correct.

The two metrics, support and confidence, are used in association rule learning, as well as other rule based machine learning techniques [30, 33]. We use slightly modified definitions of of these to handle arbitrary predicates as rules. During the learning process, each rule is assigned a support and confidence to measure the amount and quality

of evidence for the rule.

$$\textit{support(r)} = \frac{|\{C \in \mathcal{TR} \mid S_r \cup T_r \subseteq C\}|}{|\mathcal{TR}|}$$

$$\textit{confidence(r)} = \frac{|\{C \in \mathcal{TR} \mid p_r(S_r, T_r) \subseteq C\}|}{support(r) * |\mathcal{TR}|}$$

Support is the frequency that the set of keywords in the proposed rule, $S \cup T$, have been seen in the configuration files $C$ in the training set $\mathcal{TR}$. Confidence is the percentage of times the rule predicate has held true over the given keywords. In the learning process, each class of rule is manually assigned a support and confidence threshold, $t_s$ and $t_c$ respectively, below which a rule will be rejected for lack of evidence. The denote the set rules that are learned and included as part of the final specification are as follows:

$$\begin{aligned} Learn(\mathcal{TR}) = \{r|\ & support(r) > t_s \wedge \\ & \textit{confidence(r)} > t_c \wedge \\ & S,\ T \subset \mathcal{K}\} \end{aligned} \tag{1}$$

### 5.1 Error Classes

Each class of error forms a rule with a predicate $p$, and additional restrictions on the sets $S, T$. Ordering errors have $|S|, |T| = 1$ and use the predicate $order$ to mean the keyword $S$ must come before the keyword $T$ in any configuration file. Missing keyword entry errors also require $|S|, |T| = 1$ and use the predicate $missing$ to mean the keyword $S$ must appear in the same file as the keyword $T$ in any configuration file. The type rule is a set of rules over multiple predicates, which take the form $S \Rightarrow \textit{isType} * (S)$, where $|S| = 1, |T| = 0$ and $*$ matches all the basic types (string, int, etc).

VeriConf also supports two types of integer correlation rules, coarse-grained and fine-grained. Both integer correlation rules are set of rules over the predicates $\{<, =, >\}$. Coarse grain rules require $|S|, |T| = 1$, and the predicates have the typical interpretation. Fine-grained rules use $|S| = 2, |T| = 1$, and interpret the predicates such that for $k_1,\ k_2 \in S,\ k1 * k2$ must have the predicate relation to $T$. The integer correlation rules also use probabilistic types to prune the search space. To avoid learning too many false positives, we restrict this rule to either $size * int = size$, or $int * int = int$. Without probabilistic typing, we would also learn, for example, $size * size = size$, which is an invalid interpretation.

### 5.2 Checker

With the rules generated by the learner module, VeriConf checks whether any entry in a target configuration file violates the learned rules and constraints. VeriConf parses a verification target configuration file with the translator from Sec. 4 to obtain a set of key-value pairs, $\mathcal{V}$, for that file. Then, the checker applies the learner from Eq. 1, $\textit{Learn}(\mathcal{V})$ to build the set of relations observed in the file, with the thresholds $t_s, t_c = 100\%$. The checker will then report the following set of errors:

$$\begin{aligned} \textit{Errors}(\mathcal{V}) = \{r \mid S_r \cup T_r \in \mathcal{V} \wedge \\ r \in \textit{Learn}(\mathcal{TR}) \wedge \\ r \notin \textit{Learn}(\mathcal{V}) \end{aligned}$$

For any relation from the verification target that violates a known rule, the checker will report the predicate and keyword sets associated with that rule as an error. Since this is a probabilistic approach, in our tool VeriConf, we provide the user with the support and confidence values as well to help the user determine if the rule must be satisfied in their particular system. For instance, the `key_buffer` misconfiguration from Sec. 2 will only be noticeable if the system experiences a heavy traffic load, so the user may choose to ignore this error if they are confident this will not be an issue.

## 6 Rule Graph Analysis

The learner outputs a set of rules learned from the training set as described in Sec. 5. Recall that a rule is an implication relationship of the form $r = S \Rightarrow p(S, T)$. This data is necessary to perform the core verification task, but can also be used for further analysis. By interpreting the rules as a graph (which we call the *rule graph*), we can use tools from graph theory to extract information about the configuration space that can improve the quality of the learned model. We inspect properties of this rule graph to sort reported errors by those most likely to be valid. To demonstrate the additional value of the rule graph, we also use it to estimate the complexity of a configuration file.

Accessibility of the rule graph is a useful property of the association rule learning technique applied by VeriConf. While it is possible to analyze the models from other machine learning techniques, such as neural networks [34] and conditional random fields [38], these analyses require a deep knowledge of the applied techniques. In contrast the rule graph is a relatively simple, yet information rich, representation of the learned model. The following section provides a precise definition of the rule graph and demonstrates useful metrics we derive for the purposes of ranking reported errors and complexity analysis.

### 6.1 Rule Ordering

We define the *rule graph* as a directed hypergraph $H = (V, E)$, with vertices $V = \{keywords\}$ and labeled, weighted edges $E = \{(V_s, V_t, l, w)\}$. The set of edges is constructed from the learned rules, using the source and target keyword sets as sources and targets respectively, the predicates as labels, and the confidence as weights:

$$\begin{aligned} \forall r \in & Learn(\mathcal{TR}).\ \exists e\ \in E. \\ & V_s = S_r \wedge V_t = T_r \wedge l = p_r \wedge w = \textit{confidence(r)} \end{aligned}$$

We will also denote $E_{V_1,V_2} \subset E$ as the *slice set* of $E$ over $V_1, V_2$. We can think of $E_{V_1,V_2}$ as being the subset of edges in $E$ such that each source set $V_s$ shares a vertex with $V_1$ and each target set $V_t$ shares a vertex with $V_2$. Formally:

$$E_{V_1,V_2} = \{(V_s, V_t, l, w) \in E \mid \exists v_1 \in V_1 \wedge v_1 \in V_s \wedge \exists v_2 \in V_2 \wedge v_2 \in V_t\}$$

We denote a standalone vertex $v$ in our subscripts as notational convenience for the singleton set containing that vertex $v$.

The size of an edge set is the sum of all weights in that set, so:

$$|E| = \sum_{(S,T,l,w) \in E} w$$

The use of the support and confidence thresholds $t_s$ and $t_c$ in the learner ensure that all weights in the rule graph are positive.

We define a measure of degree $\mathcal{D}(v)$ for each vertex $v$ as the sum of in-degree and out-degree. Explicitly, for a vertex $x \in V$:

$$\mathcal{D}(x) = \sum_{v \in V} |E_{x,v}| + \sum_{v \in V} |E_{v,x}|$$

We may now use this measure to rank our errors. The more rules of high confidence are extracted for a keyword by the learner, the higher the $\mathcal{D}(v)$ of the corresponding vertex in the rule graph. In our final analysis, we use this classification to order the reported *errors* by estimated importance.

Keywords (specifically their corresponding vertices) of low $\mathcal{D}(v)$ may be rarer configuration parameters where rules learned are more likely to be governed by technical necessity, rather than industry convention. As such, errors reported involving low-degree keywords are more likely to be errors of high significance and should be presented with high importance to users of VeriConf.

Specifically, for an error reported by VeriConf on a rule $r$ involving keywords $K$, we rank the errors by:

$$\textit{RANK(r)} = \frac{\sum_{k \in K} \mathcal{D}(k)}{|K|}$$

The results from ranking errors in this way are presented later in the paper.

### 6.2 Complexity Measure

We may also use the rule graph to advance our general knowledge of the configuration space, outside the strict confines of a verification system. As an example, we present a heuristic for configuration file complexity based on the topology of the rule graph. This measure of complexity could be used by software organizations to manage configuration files in much the same way as Kolomogrov complexity [32] is used to manage code - identifying potentially brittle configurations for targeted refactoring.

For a configuration file with a set of keywords $K$ and a rule graph $H = (V, E)$, we define our complexity measure:

$$\mathcal{C}(K, H) = \sum_{k \in K} \begin{cases} 1 * (1 - \frac{|E_{k,K}|}{|E_{k,V}|}) & \text{if } |E_{k,V}| > 0 \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

The complexity measure, $\mathcal{C}$, can be thought of as an extension of the naïve line-counting measure of complexity. When a keyword in the configuration file is present in the rule graph, we may consider the set $E_{k,K}$ to be all learned rules involving keyword $k$ that are *relevant* to the configuration file being examined. The set $E_{k,V}$ denotes *all* learned rules involving $k$. Given these sets, we may think of $\frac{|E_{k,K}|}{|E_{k,V}|}$ as representing the amount that $k$ is constrained in the current configuration file relative to how much it could be constrained in the global configuration space. The more constrained a configuration keyword in a particular configuration file, the *less* it should contribute to the complexity (hence $1 * (1 - \frac{|E_{k,K}|}{|E_{k,V}|})$). If a keyword is not constrained at all in the current configuration file or is not present in the rule graph, we revert to the standard counting metric of complexity.

While an in-depth evaluation of the complexity metric presented here is out of scope for this paper, we use this measure to demonstrate the flexibility of the rule graph, and potential for further applications.

## 7 Implementation and Evaluation

We have implemented a tool, VeriConf, and evaluated it based on real-world configuration files taken from Github. VeriConf is written in Haskell and is available open source at *url redacted for anonymity*. Thanks to the Haskell's powerful type system, the implementation can easily be extended with with new rule classes or applied to different configuration languages with minimal change to the rest of the code base. A user only needs to provide the functions for the rule interface (a typeclass in Haskell) to 1) learn relations from a single file 2) merge two sets of rules and 3) check a file given some set of rules.

### 7.1 Evaluation

To evaluate our VeriConf prototype, we require a separate training set and test set. For our training set, $\mathcal{TR}$, we use a preexisting set of 256 industrial MySQL configuration files collected in previous configuration analysis work [11]. This is an unlabeled training set, though most of the files have some errors. For our test set, we collected 1000 MySQL configuration files from Github, and filtered the incorrectly formatted files out for a final total of 973 files. In our evaluation we focus on MySQL for comparability of results, but VeriConf can handle any configuration language (that can be parsed to the intermediate representation from Sec. 4).

We report the number of rules learned from the training set and number of errors detected in the test set in Table 1. One interesting note is that without probabilistic types we learned 327 fine grained rules and detected 1367 errors. By introducing probabilistic types, we remove 114 rules and 1023 false positives. We can be guaranteed these are

all false positives since there cannot be a rule of type $size * size = size$ because of the semantic interpretation of the $size$ units.

We also provide the support and confidence thresholds, $t_s, t_c$, used in this evaluation. These number can be adjusted by the user as a slider to control the level of assurance that their file is correct. Since these settings depend on both the user preference and training set quality, we simply choose values for which VeriConf reports reasonably sized output.

We record the histogram of errors across the test set in Figure 4. This is intuitively an expected result from randomly sampling Github - most repositories will have few errors, with an increasingly small number of repositories having many errors.

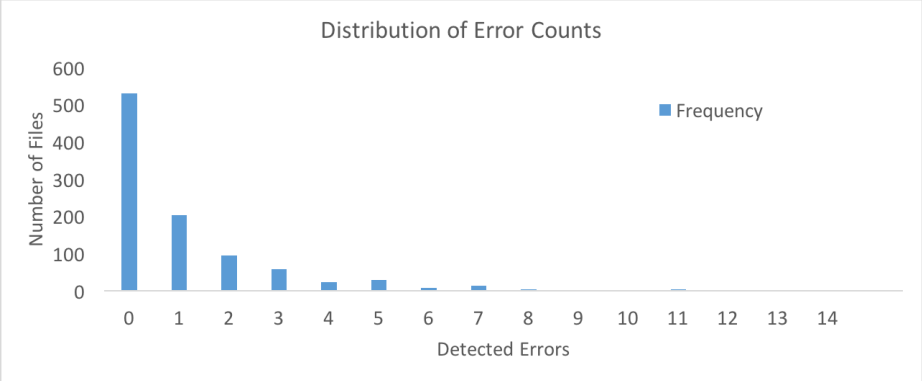


Fig. 4: Histogram of errors - 14 errors were detected in 1 file

Table 1: Results of VeriConf

| Class of Error | # Rules Learned | # Errors Detected | Support | Confidence |
|---|---|---|---|---|
| Order | 13 | 62 | 6 % | 94 % |
| Missing | 53 | 55 | 2 % | 71% |
| Type | 92 | 389 | 12 % | 70% |
| Fine-Grain | 213 | 324 | 24 % | 91% |
| Coarse-Grain | 97 | 237 | 10 % | 96% |

The errors reported may have varying impacts on the system, ranging from failing to start, runtime crash, or performance degradation. However, since VeriConf is a probabilistic system, it is also possible that some errors are false positives, a violation of the rule has no effect on the system. Note that in contrast to program verification, we do not have an oracle for determining if a reported error is a true error or a false positive. While we can run a program to determine the effect a specification has on the

Table 2: Sampled misconfiguration files for error detection evaluation.

| Errors | URLs | None | RG | PT | RG∧ PT |
|---|---|---|---|---|---|
| ORDERING ERROR: Expected "innodb_data_home_dir" BEFORE "innodb_data_file_path" | [12] | 12/12 | 3/12 | 5/5 | 3/5 |
| | [2] | 11/11 | 2/11 | 3/3 | 3/3 |
| | [9] | 9/9 | 3/9 | 4/4 | 3/4 |
| MISSING ERROR: Expected "key_buffer" WITH [isam-chk] | [22] | 6/10 | 2/10 | 2/4 | 2/4 |
| | [1] | 2/3 | 3/3 | 2/3 | 3/3 |
| | [10] | 2/3 | 3/3 | 2/2 | 3/3 |
| TYPE ERROR: Expected an integer for slow_query_log | [16] | 32/34 | 1/34 | 5/7 | 1/7 |
| | [4] | 9/20 | 2/20 | 10/11 | 2/11 |
| | [5] | 9/19 | 2/19 | 10/11 | 2/11 |
| FINE GRAINED ERROR: Expected "max_connections" * "sort_buffer_size" > "key_buffer_size" | [17] | 30/34 | 18/34 | 6/7 | 3/7 |
| | [21] | 23/25 | 9/25 | 8/9 | 3/9 |
| | [20] | 20/23 | 14/23 | 6/7 | 5/7 |
| INTEGER CORRELATION ERROR: Expected "max_allowed_packet" < "innodb_buffer_pool_size" | [23] | 29/32 | 8/32 | 11/14 | 4/14 |
| | [3] | 22/23 | 2/23 | 9/10 | 2/10 |
| | [14] | 10/12 | 4/12 | 4/4 | 2/4 |

success of compiling/running the program, no such test exists for configuration files. Because configurations are dependent on the rest of the system (*i.e.*, the available hardware, the network speed, and the usage patterns), we cannot simulate the all conditions to determine if a reported error will cause system failure. As evidenced by Example 2, some misconfigurations will only cause greater than expected performance degradation, and only under particular traffic loads. In light of this, the definition of a true error is necessarily imprecise.

Although we cannot identify false positives, we can identify true positives by examining online forums, like StackOverflow, for record that particular configuration settings have caused problems on real-world systems. Furthermore, any error for which we can find evidence online is likely to be more problematic than errors that do not have an online record, using the reasoning that this error has caused enough problems for people to seek help online. In this case, we would like VeriConf to sort the errors by their importance or potential severity. To achieve this sorting we use the rule graph analysis metric described in Sec. 6.1.

To estimate the impact of this metric, we track the rank of known true positives with, and without, the augmented rule ordering in Table 2. For this table, we picked the known true positive rules, listed in the Errors column, and pick configuration files in the test set that have these errors. We picked 3 files for each true positive by choosing the files the most total errors to most clearly observe the effects of our optimizations. We test the following conditions; just rule graph analysis (RG) to sort the errors, just probabilistic types to filter the rules (PT), and both optimizations at the same time (RG $\wedge$ PT). For each entry we list X/Y, where X is the rank of the known true positive, and Y is the total number of errors found on that file.

We also evaluate the speed of VeriConf. Generally, once a set of rules has been learned, it is not necessary to rerun the learner. However, we have only used VeriConf to build rules for MySQL, but any configuration language can be analyzed with VeriConf

given a training set, which requires rerunning the learner. Additionally, in an industrial setting, the available training set may be much larger than ours, so is important that the learning process scales. We see in Table 3 that VeriConf scales roughly linearly.

We compare VeriConf to prior work in configuration verification, ConfigC [39]. ConfigC scales exponentially because the learning algorithm assumes a completely correct training set, and learns every derivable relation. With VeriConf, we instead only process rules that meet the required support and confidence, reducing the cost of resolving to a consistent set of rules. The times reported in Table 3 were run on four cores of a Kaby Lake Intel Core i7-7500U CPU @ 2.70GHz on 16GB RAM and Fedora 25.

Table 3: Time for training over various training set sizes

| # of Files for Training | ConfigC (sec) | VeriConf (sec) |
|---|---|---|
| 0 | 0.051 | 0.051 |
| 50 | 1.815 | 1.638 |
| 100 | 13.331 | 4.119 |
| 150 | 95.547 | 10.232 |
| 200 | 192.882 | 12.271 |
| 256 | 766.904 | 15.627 |

## 8 Related Work

Configuration verification has been considered a promising way to tackle misconfiguration problems [46]. Nevertheless, a practical and automatic configuration verification approach still remains an open problem.

**Language-support misconfiguration checking.** 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, administrators often produce configuration errors in their routing configuration files. PRESTO [29] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [36] adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [28] constructs a language to describe domain knowledge about devices and services for convenient network reasoning and management. Compared with the above efforts, VeriConf mainly focuses on software systems, *e.g.*, MySQL and Apache, and our main purpose is to automate configuration verification rather than proposing new languages to convenient configuration-file writing. The closest effort to VeriConf is ConfigC [39], which aims to learn configuration-checking rules from a given training set. Compared with VeriConf, ConfigC has the following disadvantages. First, ConfigC requires the configuration files in the training set must be correct, which is quite impractical, because it is very difficult to determine a correct configuration set in reality. Second, ConfigC can only cover less

types of misconfigurations than VeriConf. Finally, the training time of ConfigC is much longer than VeriConf.

**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.* [31], for example, proposed a language, ConfValley, to validate whether given configuration files meet administrators' specifications. Different from VeriConf, 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, VeriConf does not need users to manually write anything.

Several machine learning-based misconfiguration detection efforts also have been proposed [48, 50]. EnCore [50] is one of representative work in this area. EnCore 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 with EnCore, VeriConf has the following advantages. First, VeriConf does not rely on any template. Second, EnCore cannot detect missing entry errors, type errors, ordering errors and fine-grained integer correlation errors, but VeriConf can detect all of them. Finally, VeriConf is a very automatic system, but EnCore needs significant human interventions, *e.g.*, system parameters and templates.

**Misconfiguration diagnosis.** Many misconfiguration diagnosis approaches have been proposed [25, 26]. For example, ConfAid [26] and X-ray [25] use dynamic information flow tracking to find possible configuration errors that may result in failures or performance problems. AutoBash [40] tracks causality and automatically fixes misconfigurations. Unlike VeriConf, most misconfiguration diagnosis efforts aim at finding errors after system failures occur, which leads to prolonged recovery time.

## 9 Conclusion

In this paper, we introduce VeriConf, a highly modular framework that allows automatic verification of configuration files. The main problem for verification of configuration files is their lack of specification, so they are not a traditional target area for formal methods. Inspired by the association rule learning algorithm, VeriConf learns a set of rules which describe properties and relations between keywords appearing in configuration files. These rules corresponding to the specification. Our evaluation, based on a real-world examples, shows that VeriConf is able to correctly detect and report configuration errors including ordering, missing entry, integer correlation and type errors.

## References


1. Aymargeddon, https://raw.githubusercontent.com/bennibaermann/Aymargeddon/b85d23c0690b1c6a48a045ea45f4c8b19b036fa5/var/my.cnf
2. container, https://www.dropbox.com/s/5alc0zs0qp5i529/ybh8r3n2avj7sqd1rcmx0orzry23bopl.cnf?dl=0
3. containerization, https://raw.githubusercontent.com/billycyzhang/containerization/78c6e8fefbafb89de8c28296e83a2f6fefe03879/enterprise-images/mariadb/my.cnf
4. evansims, https://raw.githubusercontent.com/evansims/scripts/715e4f4519bbff8bab5ab26a15256d79796c923a/config/mysql/my-2gb.cnf
5. evansims-script, https://raw.githubusercontent.com/evansims/scripts/715e4f4519bbff8bab5ab26a15256d79796c923a/config/mysql/my-1gb.cnf
6. Facebook, Tinder, Instagram suffer widespread issues, http://mashable.com/2015/01/27/facebook-tinder-instagram-issues/
7. Fatal Error: Cannot allocate memory for the buffer pool., http://dba.stackexchange.com/questions/25165/intermittent-mysql-crashes-with-error-fatal-error-cannot-allocate-memory-for-t
8. Fine-grained value correlation error, http://serverfault.com/questions/628414/my-cnf-configuration-in-mysql-5-6-x
9. isucon2-summer-ruby, https://raw.githubusercontent.com/co-me/isucon2-summer-ruby/1f633384f485fb7282bbbf42f2bf5d18410f7307/config/database/my.cnf
10. mini-2011, https://raw.githubusercontent.com/funtoo/experimental-mini-2011/083598863a7c9659f188d31e15b39e3af0f56cab/dev-db/mysql/files/my.cnf
11. Misconfiguration dataset, https://github.com/tianyin/configuration_datasets
12. mysetup, https://raw.githubusercontent.com/kazeburo/mysetup/99ba8656f54b1b36f4a7c93941e113adc2f05f70/mysql/my55.cnf
13. PHP CLI Segmentation Fault with pgsql, http://linux.m2osw.com/php_cli_segmentation_fault_with_pgsql
14. puppet, https://raw.githubusercontent.com/a2o/puppet-modules-a2o-essential/9e48057cc1320de52548ff019352299bc4bd5069/modules/a2o_essential_linux_mysql/files/my.cnf
15. Stack Overflow, http://stackoverflow.com/
16. Stats-analysis, https://raw.githubusercontent.com/NCIP/stats-analysis/ec7a1a15b0a5a7518a061aedd2d601ea7cc2dfca/cacoresdk%203.2.1/conf/download/my.cnf
17. Stats-analysis, https://raw.githubusercontent.com/NCIP/stats-analysis/ec7a1a15b0a5a7518a061aedd2d601ea7cc2dfca/cacoresdk%203.2.1/conf/download/my.cnf
18. The issue for slow query log, http://forum.directadmin.com/showthread.php?t=47547
19. Type Error Example, https://github.com/thekad/puppet-module-mysql/blob/master/templates/my.cnf.erb

20. vit-analysis, https://www.dropbox.com/s/09joln8kacu9ceq/ekqjat6m1j5nv9ihjhua9q89sid77cso.cnf?dl=00
21. vitroot, https://raw.githubusercontent.com/vitroot/configs/90441204dbae37521912eaaeedd3574db07b8ae4/my.cnf
22. vitroot2, https://www.dropbox.com/s/qcfmsx12i4pjjtd/missing.cnf?dl=0
23. vps, https://raw.githubusercontent.com/rarescosma/vps/7d0b898bb30eecac65158f704b43bb4d1ca06dbe/_config/mysql/my.cnf
24. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: Acm sigmod record. vol. 22, pp. 207–216. ACM (1993)
25. Attariyan, M., Chow, M., Flinn, J.: X-ray: Automating root-cause diagnosis of performance anomalies in production software. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Oct 2012)
26. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Oct 2010)
27. Bobot, F., Filliâtre, J., Marché, C., Paskevich, A.: Let's verify this with why3. STTT 17(6), 709–727 (2015)
28. Chen, X., Mao, Y., Mao, Z.M., van der Merwe, J.E.: Declarative configuration management for complex and dynamic networks. In: ACM CoNEXT (CoNEXT) (Nov 2010)
29. Enck, W., McDaniel, P.D., Sen, S., Sebos, P., Spoerel, S., Greenberg, A.G., Rao, S.G., Aiello, W.: Configuration management at massive scale: System design and experience. In: USENIX Annual Technical Conference (USENIX ATC) (Jun 2007)
30. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Mining and Knowledge Discovery 15(1), 55–86 (2007)
31. Huang, P., Bolosky, W.J., Singh, A., Zhou, Y.: Confvalley: A systematic configuration validation framework for cloud services. In: 10th European Conference on Computer Systems (EuroSys) (Apr 2015)
32. Kolmogorov, A.N.: Three approaches to the definition of the concept quantity of information. Problemy peredachi informatsii 1(1), 3–11 (1965)
33. Langley, P., Simon, H.A.: Applications of machine learning and rule induction. Communications of the ACM 38(11), 54–64 (1995)
34. Lei, T., Barzilay, R., Jaakkola, T.: Rationalizing neural predictions. arXiv preprint arXiv:1606.04155 (2016)
35. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16. pp. 348–370 (2010)
36. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: Extensible routing with declarative queries. In: ACM SIGCOMM (SIGCOMM) (Aug 2005)
37. Piskac, R., Wies, T., Zufferey, D.: Grasshopper - complete heap verification with mixed specifications. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014. pp. 124–139 (2014)
38. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from big code. In: ACM SIGPLAN Notices. vol. 50, pp. 111–124. ACM (2015)
39. Santolucito, M., Zhai, E., Piskac, R.: Probabilistic automated language learning for configuration files. In: 28th Computer Aided Verification (CAV) (Jul 2016)
40. Su, Y., Attariyan, M., Flinn, J.: AutoBash: Improving configuration management with operating systems. In: 21st ACM Symposium on Operating Systems Principles (SOSP) (Oct 2007)

41. Wang, H.J., Platt, J.C., Chen, Y., Zhang, R., Wang, Y.: Automatic misconfiguration troubleshooting with PeerPressure. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
42. Whitaker, A., Cox, R.S., Gribble, S.D.: Configuration debugging as search: Finding the needle in the haystack. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
43. Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R.: Hey, 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) (Aug 2015)
44. Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., Pasupathy, S.: Early detection of configuration errors to reduce failure damage. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Nov 2016)
45. Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., Pasupathy, S.: Do not blame users for misconfigurations. In: 24th ACM Symposium on Operating Systems Principles (SOSP) (Nov 2013)
46. Xu, T., Zhou, Y.: Systems approaches to tackling configuration errors: A survey. ACM Comput. Surv. 47(4), 70 (2015)
47. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: 23rd ACM Symposium on Operating Systems Principles (SOSP) (Oct 2011)
48. Yuan, D., Xie, Y., Panigrahy, R., Yang, J., Verbowski, C., Kumar, A.: Context-based online configuration-error detection. In: USENIX Annual Technical Conference (USENIX ATC) (Jun 2011)
49. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2005)
50. Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., Zhou, Y.: Encore: Exploiting system environment and correlation information for misconfiguration detection. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (Mar 2014)