An Automatic Verification Framework for Software Configurations

Abstract

System failures resulting from configuration errors are major reasons for compromised availability and reliability of today's software systems. Although many misconfiguration handling techniques such as checking, troubleshooting, and repair have been proposed, offering automatic verification for configuration files – as often done for regular programs – is still an open problem. This is because software configurations are typically written in poorly structured and untyped "languages", and specifying constraints and rules for configuration verification is non-trivial in practice.

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

ConfigV is capable of detecting various errors that cannot be detected by previous efforts, including entry ordering errors, fine-grained value correlation errors, and missing entry errors. We evaluate ConfigV using a real-world dataset with 261 incorrect MySQL configuration files, and ConfigV is able to correctly detect errors (previous work failed to detect) in 217 files.

1 Introduction

Configuration errors are one of the most important root causes of today's software system failures [28, 29]. In their empirical study Yin *et al.* [29] report that about 31% of system failures were caused by misconfiguration problems and only 20% were caused by bugs in program code. 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 [26,27,30,31].

While many efforts have been proposed to check, troubleshoot, diagnose, and repair configuration errors [8,23, 25], 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 [9,16,19] that can detect errors without executing the code. Two main obstacles why we cannot simply apply the existing automatic tools and techniques to verification of configuration files are: 1) a lack of a specification which would describe properties of configuration files, and 2) 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. Thus, automated verification of configuration files would be highly desirable [24,28,31].

To overcome these obstacles, researchers proposed approaches based on statistical analysis and learning [24, 30,31] that try to infer rules and policies about how configuration files are constructed. Rather than explicitly specifying entries' types or rules, these efforts focused on learning policies from a sample dataset. The learned rules are usually constructed in the following way: for every entry in a configuration file, they check if it deviates from a "typical" value, *i.e.*, a value computed from a large training set consisting of configuration files. If the entry is significantly different from a typical value, they suspect that it could be a potential configuration error.

The downside of these is that their learning approaches are limited to simplistic configuration errors, such as type errors and syntax errors, or they heavily rely on template-based inference [31]. Many sophisticated configuration errors, which are difficult to template in practice, cannot be detected in this way. As an illustration of such an error, consider entry ordering errors, which happens when some entries are inserted in a wrong order. For example, if in a PHP configuration file an entry `extension = mysql.so` appears before `extension = recode.so`, it would lead to a crash error, where the Apache server cannot start due to the segmentation fault error. The correct ordering should be `extension = recode.so` before `extension = mysql.so` [29]. These types of errors cannot be detected by existing learning efforts, since it is hard to build a corresponding template [28].

In this paper, we present ConfigV, the first automatic verification framework for general software configurations. ConfigV is based on a collection of powerful learning algorithms that do not necessarily depend on templates. The learning process takes as input a large sample of configuration files. The process is language-agnostic and works for any kind of configuration file, 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, ConfigV learns an abundant set of rules specifying various properties that hold on the given sample. The files in the sample might contain errors – we are, therefore, using probabilistic learning to derive a set of accurate rules.

For practical purposes we use a real-world dataset [2]. Every file in our dataset contains several errors, but they are typically different errors and only appear in a small percentage of files. Using that insight we were able, with the help of the probabilistic cutoff, to learn an accurate set of rules. The rules, in general, specify which properties variables need to satisfy. One can see this learning process as a way of deriving a specification for configuration files. Once there is a specification, we can do formal verification. With these rules we can efficiently check the correctness of the configuration files of interest and detect potential errors.

The learning process has two phases. In the first phase, ConfigV analyzes the sample dataset and generates a well-structured and probabilistically-typed intermediate representation. In the second phase, ConfigV derives rules and constraints by analyzing the intermediate representations. Every learned rule is annotated with a probability depicting which percentage of the files the rule was seen as correct. We only accept those rules whose correctness probability is above the given threshold.

Finally, while the learned rules establish correlations between two or more variables, using ConfigV, one can also detect whether a single variable has an appropriate value assigned to it. We do that by measuring the values that are typically assigned to that variable and reporting if the current value deviates too much from the recorded values.

Building such an automatic verification framework for configuration files, nevertheless, requires addressing several challenges. First, we need to assign a type to every variable in a configuration file during the transformation phase. However, the type of a variable cannot always be fully determined from a single value. For example, an entry `temp_dir = 300` assigns to variable `temp_dir` either a directory named "300" or sets the size of that directory to 300 (an integer). Some existing type inference work would report this is an error, because `temp_dir` should be assigned an integer [31]. 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. The entry in the above example might be assigned the following probabilistic type `{temp_dir, 300, [(File, 60%), (Int, 40%)]}`. Using probabilistic types, we can generate a more accurate language model, thus significantly improving our verification capabilities.

Second, when learning rules we use very general templates to infer them. 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, ConfigV introduces no 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 Motivating Examples

In this section we illustrate functionality of ConfigV on several 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 [24, 31], cannot detect these configuration errors. Most of the presented examples were found on StackOverflow, a popular question and answer website for programmers. To better understand problems that users have with configuration files, we explored and analyzed misconfigurations on a large number of user forums and online discussion sites.

Example 1: Ordering error. Ordering errors were reported by Yin *et al.* [29] and our first example illustrates how ordering errors can cause a system to crash. 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
```

This 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. We call the above example of a misconfiguration file an *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 can effectively solve or detect this problem [27, 28, 31].

By invoking ConfigV, the user can detect such a configuration error. In particular, ConfigV reports that recode.so should appear before mysql.so, as shown below:

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

Example 2: Fine-grained value correlation error. Our next real-world misconfiguration example [1] comes from a discussion on StackOverflow. 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 complains 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.

If we run ConfigV on this configuration file, ConfigV would return:

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

We can see that during the learning process not only do we learn simple statements that compare two values, but also we learn more complex correlations. We call these more complex relations fine-grained value correlations, and the errors *fine-grained value correlation errors*. This type of error is more sophisticated than the simple value correlation that some tools can detect [29, 31]. A typical value correlation error states that one entry's value should have a certain correlation with another entry's value. For example, in MySQL, the value of max_connections should be higher than mysql.max_persistent.

Our tool can learn this simple correlation as well, but learning more complex properties requires a different approach to the learning process. We need to track several variables. It is not enough just to compare them, but we need to learn, as in this particular case, which invariant must be preserved. It was pointed out by Xu *et al.* [26] that detecting fine-grained value correlation errors is a much more challenging task than the normal value correlation problem.

To the best of our knowledge, ConfigV is the first tool that is able to check such fine-grained value correlation problems.

Example 3: Missing entry error. Many critical system outages result from the fact that an important entry was missing in the configuration file. We call such a problem a *missing entry error*. In a public misconfiguration dataset [2], many MySQL failure reports were caused by missing entry errors. Below is a real-world missing entry error example [29]: when a user wants to use OpenLDAP to enable her directory access protocol, she needs to use the password policy overlay. This is usually achieved via the following entries in the OpenLDAP configuration file:

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

When using the password policy overlay in OpenLDAP, users must first include the related schema. Leaving out the `include schema/ppolicy.schema` entry, as done by many users [29], causes the failure of LDAP. If the user runs ConfigV on such a misconfiguration file, ConfigV would return:

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

Example 4: Singular value anomalies. Many system performance problems are caused by the anomaly that a value is set either too high or too low. For example, a parameter relating to memory might be too large, exhausting the RAM and causing extreme slowness or even a crash. Consider the following real-world misconfiguration file [6]: a user was experiencing the "My SQL Server has gone away" error. This is difficult to debug, since the error message is not specific. It turns out the following line in her configuration file was problematic:

```
max_allowed_packet = 100M
```

The user eventually resolved the issue by replacing the above line with `max_allowed_packet=2M`. One possible way to reach this conclusion more quickly is to determine that the original value, 100 MB, is statistically deviant and extremely large for the `max_allowed_packet` attribute. ConfigV can detect these outliers. Run on the user's original configuration file, ConfigV would output:

```
WARNING: Violated Upper Hampel Rule for
    max_allowed_packet with value 104857600
```

In the above, 104857600 represents the integer value of 100 megabytes. The idea is to output a warning to the user that her value falls outside of a range considered normal, so that she can make the appropriate adjustment to the value herself. In the above situation, it is prudent to set the value of `max_allowed_packet` lower. We call such a value a *singular value anomaly*.

Other errors. In addition to the given motivating examples ConfigV can also detect and report configuration errors that can be exposed by existing work, such as EnCore [31] and CODE [30] (a full scope of ConfigV's capabilities is given in §9). A recently reported configuration error made the MySQL daemon fail to start [5]. One entry was written as `datadir=/root/appfinder/mysql`, and the type, as well as format, seemed correct. However, the problem is that this directory should not contain the root directory; the correct entry should be `datadir=/appfinder/mysql`. Such an error is called a system environment-related configuration error. Both ConfigV and EnCore can handle misconfigurations concerning system execution. Furthermore, ConfigV is also able to detect type errors and syntax errors in configuration files. An example of a type error is assigning to a variable `general_log` a file path `/var/log/mysql/mysql.log`. ConfigV successfully reports this typing error:

```
TYPE ERROR: Expected a Int with P=1.0 for "
    general_log[mysqld]"
```

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.

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 [26]. 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

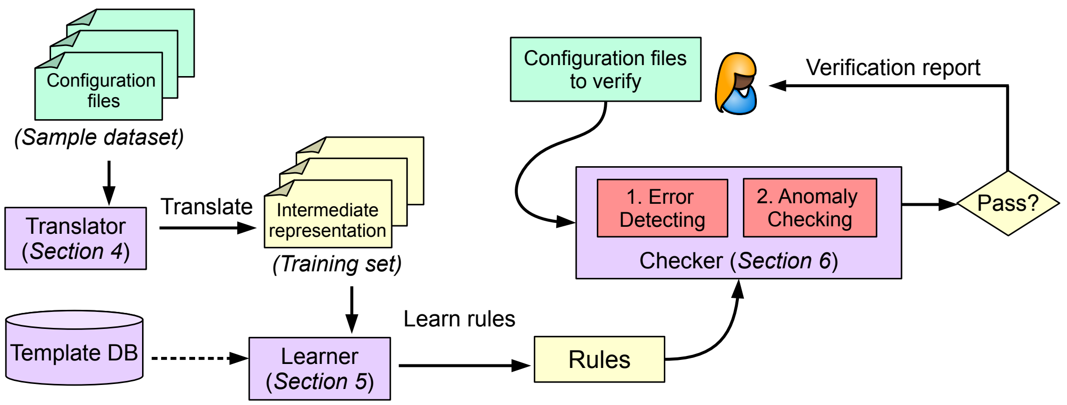

Figure 1: ConfigV's workflow. The green components represent configuration files, including both sample configuration datasets and users' input configuration files to verify. The purple components are the modules of ConfigV. Because template DB is not necessarily used, we use dashed arrow between it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

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 §2) 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 Translator

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 §5). Coupled with rules generated by learner module, we will have a complete language model to verify configuration files of our interest.

Translating or parsing is system dependent. In other words, for MySQL and HDFS, we need to develop different parsers to handle each of them, respectively. ConfigV allows users to provide extra help to the translator for their specific system 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 [15].

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

When a value has a probabilistic type, we generate rules for all its types. This means that by assigning `foo` a probabilistic type (*e.g.*, `(foo, 300,`

Table 1: 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.	$\mathsf{sameSubnet}(addr1, addr2)$

`[(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 $\mathscr{T}$ 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 1 contains the most important types and templates.

In general, the templates works as follows: if there are two entries $\texttt{k1} = \texttt{v1}$ and $\texttt{k2} = \texttt{v2}$, 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 $\texttt{k} = \texttt{v}$ 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 Learner

The goal of the learner module is to derive rules and constraints from the intermediate representation generated by the translator. In general, the learner module has two components. The first component (§5.1) learns rules for checking configuration errors like missing entry errors, ordering errors, and fine-grained value correlation errors. These errors tend to cause total system failures. The second component (§5.2) aims to derive constraints on entries to check for suspicious (or anomalous) values that may violate standard practice. These anomalies can cause partial degradation of the system, such as significant reduction in performance, or even total failure as in Example 4 of §2.

5.1 Derivation of Probabilistic Rules

The first component learns two types of rules: rules that are inferred using templates associated with types and rules which we call *untyped specifications*. An example of an untyped specification is the ordering or missing entry constraint.

A strawman solution. We first present a strawman solution (employed by previous work [22]) that uses a set of correct configuration files as a learning set, from which it is possible to derive rules that must hold with absolute certainty. In practice, however, it is difficult to obtain a set of files that is both guaranteed to be without errors and large enough to learn sufficiently many rules. This usually requires manual verification of the learning set, which is prone to error.

As a result of this restriction, these efforts only consider a rule if it holds over exactly every file in the learning set. This behavior can be formally described as follows:

$$\begin{aligned}
C &:= \text{Correct Learning Set} \\
LR &:= \text{Learned Rules} :: \{\text{Rule}\} \\
RR &:= \text{Reported Rules} :: \{\text{Rule}\} \\
C &= \{\text{Configuration Files in Intermediate Representation}\} \\
LR &= \{r \mid \forall file \in C,\ holds(r, file)\} \\
RR &= \{r \mid r \in LR \wedge \neg\, holds(r, \text{user file})\}
\end{aligned}$$

In the above, the ":: {Rule}" notation indicates that *LR*, the learned rule set, and *RR*, the reported rule set, are sets of rules. A rule is only committed to the *LR* learned rule set if the rule holds on all files in the training set *C*, as denoted by the $\forall$ quantifier. A broken rule is reported to the user in the *RR* rule set if the rule is in *LR*, but does not hold on the input user file.

In general, each rule can be viewed as a mapping from a tuple of entry keys $(k_1, \ldots, k_n)$ in the intermediate representation of a configuration file to a probability function, P_{rel}. This can be represented by a rule $r := (k_1, \ldots, k_n) \rightarrow P_{rel}$. P_{rel} is a probability function specific to the relation to assess. The problem with the strawman solution is that the probability functions are restricted to be boolean, which is to say that the relation can either be unsatisfied or satisfied, either 0 or 1. For instance, for the rule that `max_connections` must be greater than `mysql.max_persistent`, the rule is denoted by (`max_connections`, `mysql.max_persistent`)

Algorithm 1 Probabilistically Learn Rules

Let θ be the list of possible types. Let U be the list of possible untyped specifications.

```
1:  learnRules(C):
2:  Input: set C contains intermediate representations
3:          of configuration files
4:
5:  $R = \{\}$ // rule set
6:  $\Pi$ // probabilities associated with rule set
7:  for each file $F$ in $C$ do
8:      Initialize $opt$ flag.
9:      for each $z \in \theta \cup U$ do
10:         $Q$ contains candidate entries to consider
11:         $RM$ contains entries in remove set
12:         if $z$ is a type then
13:             $opt = 1$
14:             $Q = filter(F, \tau)$
15:             // Q contains $F$ entries with $\tau$ in type list
16:         else if $z$ is an untyped spec then
17:             $opt = 0$
18:             $Q = F$
19:         end if
20:         for each entry $e$ in $Q$ do
21:             if $g(e)$ is false then
22:                 continue to next entry
23:             else
24:                 $CR = \text{constructRules}(e, z, opt, Q \backslash RM)$
25:                 $R = R \cup CR$
26:                 For each $r \in CR$, update $\Pi$
27:             end if
28:             $RM = RM \cup e$
29:         end for
30:     end for
31: end for
32: return $(\Pi, R)$
```

Algorithm 2 Construct Rules

```
1:  constructRules(e, z, opt, S):
2:  Input: entry $e$ from intermediate representation
3:          type or untyped specification $z$
4:          option $opt$, set to 1 if template
5:                      or 0 if untyped spec
6:          set of entries $S$
7:
8:  $R = \{\}$
9:  Set $j$ as an iteration variable.
10: if opt == 1 then
11:     run the following sequence L over all elements of $templates(z)$
12:     i.e., iterate $j$ over $templates(z)$
13: else if opt == 0 then
14:     $j = z$
15: end if
16: Beginning of L:
17: Let $n$ be the arity of $j$.
18: Let $P$ be the set of permutations of distinct entries from $S$, of size
    $n - 1$.
19: for each $p \in P$ do
20:     Let $p$ be represented by $(e_1, e_2, \ldots, e_{n-1})$.
21:     for $i = 1 : n$ do
22:         Create a $n$-tuple by placing $e$ at the $i$-th position and filling
    the remaining entries with $p$, in order. Let this $n$-tuple be $m =
    (e_1, e_2, \ldots, e_i, \ldots, e_{n-1})$.
23:         $r = \text{constructRule}(m)$
24:         if $r$ is satisfied then $R = R \cup r$
25:         end if
26:     end for
27: end for
28: End of L.
29: return $R$
```

$\rightarrow 1$, where $B_>$ is set to 1. In our approach, we permit the probability functions to take on any value from 0 to 1.

Probabilistic approach. In ConfigV's learner module, the rule learning mechanism is tolerant enough to accept a dataset of possibly incorrect configuration files. Rather than manually correcting each file, we extend the previous formalism to run probabilistic learning on our intermediate representations (generated by the translator).

Our probabilistic approaches for learning missing entry, ordering, and fine-grained value correlation rules stem from existing work with building non-probabilistic versions of these rule-learning algorithms. For each of these rules, we consider all possible permutations of keys that appear in every file which are appropriately typed, and for our learning process, calculate the likelihood that each of these permutations constitute a rule. Rule patterns in the example set that appear frequently are accepted, so they can be used to evaluate new files.

More formally, this approach can be defined as follows:

$$
\begin{aligned}
I &:= \text{Incorrect Learning Set} \\
LP &:= \text{Learned Probabilistic Rules} :: \{\text{P\_Rule}\} \\
RP &:= \text{Reported Probabilistic Rules} :: \{\text{P\_Rule}\} \\
I &= \{\text{Configuration Files in Intermediate Representation}\} \\
(\Pi, LP) &= \text{learnRules}(I) \\
RP &= \{r \mid r \in LP \wedge \Pi(r) > p \wedge \neg holds(r, userfile)\}
\end{aligned}
$$

Note that the formalism relies on Algorithm 1. The algorithm considers both typed templates and untyped specifications. As a similar iteration logic is used for both, an *opt* flag distinguishes between the two situations. constructRule in Algorithm 2 refers to the creation of a rule from a tuple of entries, $m = (e_1, e_2, \ldots, e_i, \ldots, e_{n-1})$. The rule creation is formally represented as $m(k(e_1), k(e_2), \ldots, k(e_i), \ldots, k(e_{n-1}))$. In order to determine if the rule satisfied, the relation associated with $m(v(e_1), v(e_2), \ldots, v(e_i), \ldots, v(e_{n-1}))$ is evaluated for truth. The rule can be template-generated, for instance `max_connections` > `mysql.max_persistent`, or it can be untyped-specification-generated, for instance the ordering that `recode.so` must come before `mysql.so`.

The candidate entry set Q in Algorithm 1 differs for the two cases. For typed templates, due to typing restrictions, we must first filter and examine entries associated with the same type, in order to check that the template is satisfied over appropriately typed argument entries. This manifests itself via $Q = filter(F, \tau)$ in Algorithm 1. For untyped specifications, we do not need to adhere to this typing restriction, so we simply set $Q = F$.

A rule r will be reported as broken if the probability the rule is correct, $\Pi(r)$, is greater than some user defined constant, p. This constant can be adjust to the user's preference. A small p will increase the likelihood of finding an error, but also increase the number of false positives that are reported.

5.2 Learning Suspicious Constraints

With a configuration file that has been verified against catastrophic failures (*e.g.*, missing entry, type, and ordering errors), the user may also want to examine more subtle issues. Anomalous values can cause tricky, but impactful, performance and memory issues that are hard to debug, as discussed in Example 4 of §2. Consequently, anomalous values should be flagged and a warning returned to the user indicating the violation.

We now describe the technique we use to detect anomalous values for numerical attributes. Let A be the set of attributes contained in the configuration files in the sample dataset. Let A_n be the subset of attributes of A which are numerically typed. Then, for each attribute $a \in A_n$, we construct a vector v_a of the values corresponding to attribute a, seen over the entire sample dataset. For each v_a, we compute an interval

$$[\hat{v_a} - 50 * MAD(v_a), \hat{v_a} + 50 * MAD(v_a)],$$

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

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

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

6 Checker

With the learned rules and constraints in hand (generated by the learner module), ConfigV checks whether any entry in a target configuration file violates the learned rules and constraints. For a given configuration file, ConfigV parses it the same way employed in the translator, thus obtaining a structured and typed representation for the target configuration file. Then, the checker uses two submodules (shown in Figure 1) to check the target configuration file based on rules and constraints.

Error detecting. The first sub-module of our checker, named error detecting in Figure 1, is able to detect the following errors: missing entry errors, ordering errors, correlation errors (including fine-grained value correlation errors), type errors, and system environment errors (depending on templates). In particular, the checker simply sees whether the representations parsed from the target configuration files violate our learned rules.

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

7 Discussion and Limitations

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

Legal misconfigurations. While ConfigV can check for diverse configuration errors without human intervention, most of the existing proactive misconfiguration detection techniques, including ConfigV, cannot handle configuration errors resulting from events that occurred during system runtime. Such configuration errors are referred to as *legal misconfigurations* [29]. In particular, many parameter misconfigurations have perfectly legal types and values, but do not deliver the functionality intended by users. For example, after a website's traffic significantly increases, the parameter `key_buffer_size` in MySQL may not be able to handle increasingly more data traffic, thus leading to outage of the whole system. These cases are more difficult to detect by automatic checkers and may require more user training or better configuration design. A potential solution is to combine existing misconfiguration diagnosis tools, *e.g.*, X-ray [7] and ConfAid [8], with ConfigV in order to enhance misconfiguration checking capability.

Misconfiguration across software components. As exposed by Yin *et al.* [29], cross-software configuration correlation problems also account for a considerable number of misconfiguration cases. For example, in a LAMP-based Web server, one entry in the PHP configuration file, `mysql.max_persistent = 400`, may make users encounter a "too many connections" error, because a correlated entry in the underlying MySQL configuration file assigns `max_connection` to 300, which is less than the MySQL connection number in the PHP configuration file (*i.e.*, 400). It is quite difficult to detect such tricky errors through learning approaches, because not only are users or engineers unaware of the hidden interactions [28], but also it is hard to obtain global knowledge of the entire configuration due to the business privacy concerns of each software provider. One possible solution to this problem might be to introduce some cryptographic protocol, *e.g.*, private set intersection [14], to privately extract the overlapping entries, *e.g.*, `mysql.max_connection` in the above MySQL and PHP case, for double-checking.

Network configuration verification. ConfigV mainly focuses on software configurations, *e.g.*, MySQL and Apache, so that our approach is limited to support network configuration verification. This is because network configurations have quite different representations, format, and rules from software configurations, since network configurations are typically written in more domain-specific policy languages. In fact, many network verification tools, *e.g.*, NoD [18] and Dobrescu *et al.* [11], have been proposed to check whether network configurations meet their specifications.

More complex configuration structure. Currently, ConfigV mainly targets key-value configuration files, but in practice many systems, *e.g.*, OpenStack [3], employ very complex configuration format and structure, which ConfigV cannot handle. Verifying such structurally complex configurations typically requires ConfigV to learn a much more sophisticated language model, which is challenging in practice. Recent efforts [20, 21] may present possible solutions to this limitation. These techniques can learn a call-graph from a training program set, and check a new program based on properties extracted from this generated call-graph. If we examine a structurally complex configuration file as a program with these tools, we may be able to use similar techniques to verify whether the configuration file violates a property of interest.

8 Implementation

ConfigV is primarily implemented in Haskell, and the suspicious values detection module was written in R. For clarity, we only present the key components of the Haskell implementation here.

The translator is developed as a parser for the grammar described in §4. The learner and checker module fully implemented the design in §5 and §6, respectively.

To verify errors, our prototype follows three functions, `learn`, `merge`, and `check`, to learn the rules.

The `learn` function takes the intermediate representation of a single configuration file (as generated by the translator), and generates the set of rules that can be derived from that file. Specifically, these rules take the form of a *P_Rule* to track the counts of relation events. The `merge` function will unify two sets of rules, from two separate configuration files, into a single consistent set of rules. For the probabilistic rules, this tends to be simply a union between the two sets. When two equivalent rules are merged, the relation event counts should be summed to update the probability distributions.

The `check` function will start all the rules we have learned, and filter until we only have the rules that have been broken by the file of interest. We first eliminate the rules that do not pass the probability check defined in §5.1. Second, we only consider the rules which are relevant to the file – we do not need to check rules about the entries that do not appear in the file. Last, we take the errors as any rule that does not hold over the given file.

The core learning algorithm will learn over each file individually, then merge the results of learning together. Since we learn each set of rules on each file in isolation from the others, we have a pleasingly parallel situation. The `learn` function can then be called in parallel over the entire learning set. As a functional language, there is no programming overhead to implement this in Haskell – we simply import the parallel mapping library [4] to replace all uses of `map` with `parmap`. The merge stage could also be parallelized by using a divide and conquer approach, but this is not a priority since the learning stage only needs to be run once per learning set, and can then be cached (in our case, as a .json file) for fast reading when doing verification.

The source code for our implementation is available at *(URL omitted for blind review)*.

9 Evaluations

We conduct three experiments to evaluate our ConfigV prototype. We answer the following questions.

- Whether ConfigV can successfully detect real-world configuration errors (§9.1)?
- Whether ConfigV can correctly detect anomalous values (§9.2)?
- How long do training and verification last (§9.3)?

9.1 ConfigV's Effectiveness

We now apply ConfigV to check against real-world misconfiguration problems. We use a real misconfiguration

Table 2: Sampled incorrect configuration files for misconfiguration detection evaluation

ID	Problem Description	Error Type	ConfigV Report
1	MySQL cannot create test file	Missing Entry	Expected innodb_force_recovery=1 in the same file as: innodb_strict_mode=1
2	How can my wait_timeout/interactive_timeout be ignored	Missing Entry	Expected set_time=1 in the same file as wait_timeout
3	MySQL on EC2 instance becomes very slow after the first query	Correlation	Expected key_buffer_size >= thread_size * sort_buffer_size
4	MySQL cannot successfully make replication	Type Error	Expected master-host[mysqld] should be a valid IP address
5	MySQL running on production servers has gone away error in memory	Type Error	Expected set-variable[myisamchk] should be a memory size like 20M
6	Cannot start MySQL 5.5 as normal user Fedora 15	Ordering	Expected socket[mysqld]=/var/lib/ mysql/mysql.sock BEFORE user[mysqld]=mysql
7	Fail to login MySQL	Type Error	Expected old_passwords[mysqld] should be a valid value
8	MySQL running at CentOS cannot know enable named pipe	Ordering	Expected localhost before bind-address[mysqld]
9	Troubles installing MySQL5 via Darwin Ports	Ordering	Expected port[mysqld]=3306 BEFORE socket[mysqld]=/tmp/mysql.sock
10	Fail to install Percona xtraDB cluster on ubuntu 13.04	Missing Entry	Expected wsrep_node_address[mysqld] in the same file as: wsrep_sst_method [mysqld]
11	MySQL access denied for user rootlocalhost	Ordering	Expected port[mysqld]=3306 BEFORE socket[mysqld]= /tmp/mysql.sock
12	MySQL Partition Problem	Correlation	Expected innodb_flush_max _commit[mysqld] >= innodb _support[mysqld]
13	mysql_upgrade script problems on MySQL 5.0.24	Type Error	Expected log-slow-queries[mysqld] should be a path type
14	MySQL: bug report!!	Ordering	Expected innodb_file_io_threads[mysqld] BEFORE innodb_log_files_in_group[mysqld]
15	MySQL max connections changed without notice	Correlation	Expected innodb_file_per_table [mysqld] <= max_connections[mysqld]
16	Enabling log in MySQL 5 6 prevents server from starting	Type Error	Expected sql_mode[mysqld] should be a string
17	MySQL needs help for rapidly growing table and decreasing speed	Correlation	Expected innodb_buffer_pool_size [mysqld] > sort_buffer_size[mysqld]
18	MySQL filled all RAM on the system	Correlation	Expected join_buffer_szie * max_connections <= key_buffer_size
19	MySQL hit a max limit	Correlation	Expected join_buffer_size * max_connections <= key_buffer_size
20	Cannot create a new thread	Correlation	Expected join_buffer_size * max_connections <= key_buffer_size

Table 3: Sampled configuration files for anomaly detection

ID	Problem Description	URL	ConfigV Report
1	MySQL Server has gone away error in Wamp	goo.gl/axnezi	WARNING: Violated Upper Hampel Rule for `max_allowed_packet` with value 104857600
2	MySQL has abnormally high load for CPU	goo.gl/JrRLrR	WARNING: Violated Upper Hampel Rule for `sort_buffer_size` with value 1048576000 WARNING: Violated Upper Hampel Rule for `read_rnd_buffer_size` with value 283115520
3	User is having performance issues with MySQL	goo.gl/34jTB5	WARNING: Violated Upper Hampel Rule for `query_cache_limit` with value 134217728

dataset [2] that was collected from several online forums, *e.g.*, Stack Overflow and MySQL forums. Each configuration file in this dataset was posted by programmers or system administrators, when they confronted misconfiguration issues in practice. This dataset contains 261 incorrect MySQL configuration files.

We apply ConfigV to learn and check the configuration files in this dataset. Then, we observe how many errors ConfigV can detect. We next manually check whether these detected errors are indeed configuration errors. For all the 261 configuration files, we found ConfigV is able to report all the errors correctly. For 217 of the 261 configuration files, we found ConfigV's error detection results are correct and accurate, *i.e.*, we correctly report the real misconfiguration problems with few false positives. For the rest of the 44 cases, although ConfigV can detect the errors, ConfigV outputs relatively high false positives, *e.g.*, higher than 50 false positives, which is not very helpful to users in practice. The reason is our training set contains too many incorrect configuration files, which produces a lot of noise in our learning process, thus resulting in so many false positives.

In order to clearly evaluate the effectiveness of ConfigV, we extract 20 incorrect configuration files from the dataset. ConfigV is able to correctly report these configuration errors, and these errors cannot be handled by any existing efforts. Table 2 details this information, including the problem descriptions of these files, error types, and the outputs produced by ConfigV.

As shown in Table 2, we found ConfigV is capable of detecting many configuration errors previous efforts, *e.g.*, EnCore [31], cannot deal with. To the best of our knowledge, no existing effort is able to detect fine-grained value correlation errors in MySQL configuration files. For example, the third case in Table 2 shows that ConfigV can detect the following fine-grained correlation: `key_buffer_size` should be larger than `thread_size` * `sort_buffer_size`.

9.2 Detecting Anomalous Values

The purpose of checking anomalous values in configuration files is to avoid potential performance or throughput problems. Checking anomalous values is mainly the responsibility of the second sub-module checker of ConfigV (see §6).

We run ConfigV to check 30 real-world benchmarks from the Stack Overflow website, and found ConfigV is able to report anomalous values in these configurations. We also manually check whether these anomalous values are correct by comparing our results with answers on StackOverflow.

Table 3 shows three examples. We present the problem description, the link, and the output results of ConfigV. We found these anomalous values are very hard for administrators or users to detect, because they look correct. However, as shown in the links posted in Table 3, such anomalous values typically lead to critical performance problems in practice.

Table 4: Training time for different datasets

The # of Files for Training	Training Time (sec.)
10	2.997
50	46.4208
100	206.7798
150	431.7772
200	856.6383

9.3 Training & Verification Performance

We run our experiments measuring the run-time of training and verification of ConfigV. Our experiments were run on a machine equipped with 64-bit Intel Core i7-4790 CPU @ 3.60GHz and 8 GB of RAM.

Training performance. We first evaluate the training time of ConfigV. Namely, we examine how long does ConfigV need to use to generate rules. We vary the number of configuration files in our training dataset between 10 and 200, and record the corresponding time. Table 4 presents our evaluational results. We found the training time of ConfigV is acceptable in practice, because we obtained good enough rules on 150 files in the training dataset. For 150 files, ConfigV only requires about 7 minutes of runtime. Furthermore, even if we train ConfigV with a 200-file dataset, we need less than 15 minutes, but such a training dataset can generate about 200,000 rules.

Verification performance. We also measure the verification time of ConfigV, and found that the verification time is very fast. With 232,613 rules, ConfigV needs less than one second to verify a MySQL configuration file. ConfigV only needs 3 minutes to finish the verification of the real-world dataset [2] we used to evaluate the effectiveness of ConfigV, which contains 261 misconfiguration files.

10 Related Work

Language support has been considered a promising way to tackle configuration problems [28]. 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 [12] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [17] adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [10] 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 [13]. 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.* [13] 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 [30, 31]. EnCore [31] 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 [7, 8]. For example, ConfAid [8] and X-ray [7] use dynamic information flow tracking to find possible configuration errors that may result in failures or performance problems. AutoBash [23] 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 [27]. SPEX [27] 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.

11 Conclusion

In this paper, we introduce ConfigV, a highly modular framework that allows automatic verification of configuration files. ConfigV employs a translator to parse a sample configuration dataset into a well-structured and typed intermediate representation, and then uses a learner module to derive rules, building a language model this way. ConfigV then uses this generated language model to verify the correctness of a given configuration file. The verification is done by checking whether any of learned rules is violated. We evaluate ConfigV using a real-world dataset [2] which contains 261 incorrect MySQL configuration files. Our experimental result shows that ConfigV is able to correctly detect errors in 217 files in that dataset. We believe that ConfigV represent a practical path toward fully automated language-based configuration verification.

References

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

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

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

[4] parallel-3.2.1.0: Parallel programming library. https://hackage.haskell.org/package/parallel-3.2.1.0/docs/Control-Parallel-Strategies.html.

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

[6] Singular value error. http://stackoverflow.com/questions/1980004/2006-mysql-server-has-gone-away-error-in-wamp.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.