EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection

Jiaqi Zhang[†], Lakshminarayanan Renganarayana[§], Xiaolan Zhang[§], Niyu Ge[§], Vasanth Bala[§], Tianyin Xu[†], Yuanyuan Zhou[†]

[†]University of California San Diego
{jiz013, tixu, yyzhou}@cs.ucsd.edu

[§]IBM Watson Research Center
{lrengan, cxzhang, niyuge, vbala}@us.ibm.com

Abstract

As software systems become more complex and configurable, failures due to misconfigurations are becoming a critical problem. Such failures often have serious functionality, security and financial consequences. Further, diagnosis and remediation for such failures require reasoning across the software stack and its operating environment, making it difficult and costly.

We present a framework and tool called EnCore to automatically detect software misconfigurations. EnCore takes into account two important factors that are unexploited before: the interaction between the configuration settings and the executing environment, as well as the rich correlations between configuration entries. We embrace the emerging trend of viewing systems as data, and exploit this to extract information about the execution environment in which a configuration setting is used. EnCore learns configuration rules from a given set of sample configurations. With training data enriched with the execution context of configurations, EnCore is able to learn a broad set of configuration anomalies that spans the entire system. EnCore is effective in detecting both injected errors and known real-world problems – it finds 37 new misconfigurations in Amazon EC2 public images and 24 new configuration problems in a commercial private cloud. By systematically exploiting environment information and by learning correlation rules across multiple configuration settings, *EnCore* detects $1.6x$ to $3.5x$ more misconfiguration anomalies than previous approaches.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability

Keywords Configuration, Data Mining, Inference

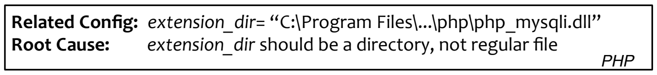
ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2305-5/14/03…$15.00.
http://dx.doi.org/10.1145/2541940.2541983

1 Introduction

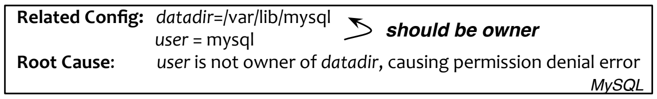
As software systems become more flexible and feature-rich, their configurations have become highly complicated. For example, MySQL and Apache httpd server, individually, has more than 200 configuration entries. As a result, correctly configuring software systems has become a highly complex task that renders manual effort difficult and error-prone [21, 41, 43, 44]. Studies show that configuration errors are both common and highly detrimental to businesses [29, 30, 41, 46]. Not only do misconfigurations lead to lost business due to system unavailability [1, 19, 24], they also demand substantial resources be devoted to troubleshooting [43, 44]. In one particular case of a large commercial corporation, misconfiguration has contributed to as many as 27% of the trouble tickets in the customer support database [46]. The situation is exacerbated by the fact that many organizations impose security and performance policies for best practices. Configuration settings that are otherwise valid from a functional perspective may not conform to these policies, leading to security flaws or performance anomalies. Detecting such sub-optimal configurations is also highly desireable.

A promising approach for taming the configuration problem (or the configuration hell [43]) is to automatically check a set of configuration settings for potential errors before deployment, just as one would normally do with application source code. However, most configuration files used today lack the rich structural and semantic information available in programming languages which enable sophisticated analysis for errors [40, 41]. To overcome the limitation, researchers have attacked the problem of misconfiguration detection by learning, for each configuration entry, the common values used in a large collection of configurations (i.e. the training set), and flagging those values that deviate from the common ones as potential misconfigurations [34, 41, 42].

While proven useful in some scenarios, the potential of these tools [34, 41, 42] is greatly limited due to the simplistic treatment of each configuration setting as a string literal in isolation. Configuration settings bridge applications to their operating environment. Therefore the diagno-


Related Config: *extension_dir*= "C:\Program Files\...\php\php_mysqli.dll"
Root Cause: *extension_dir* should be a directory, not regular file
PHP

(a) Wrong value causes modules not loaded. Environment information needs to be considered to detect the problem.


Related Config: *datadir*=/var/lib/mysql
user = mysql
should be owner
Root Cause: *user* is not owner of *datadir*, causing permission denial error
MySQL

(b) Wrong file owner causes permission problem. The correlation between different configuration entries needs to be examined.

Figure 1: **Real-world system problems caused by misconfigurations.** The root causes of these problem are hidden in (a) environment factors, and (b) the correlation between two configuration entries. Neither of these configuration errors can be detected by analyzing configuration values because (a) the value varies widely in the training set, and (b) the values are common in the training set and no correlation is considered.

sis and remediation of misconfigurations requires reasoning across both sides. Figure 1 shows two real-world misconfigurations, neither of which can be detected with existing approaches. In Figure 1(a), `extension_dir` specifies the location of the extension libraries. While it should be a directory, the user incorrectly supplied a regular file. Existing approaches that detect errors by relying on anomalies in the configuration values cannot detect this error because the value of `extension_dir` often varies across a set of samples and hence cannot be meaningfully considered as an anomaly [41]. However, by analyzing a set of sample values for this setting in the context of the executing system, one can detect that the value should be a directory and not a file.

In the example shown in Figure 1(b), `datadir` points to the directory where MySQL stores the table data, and `user` specifies the system user id with which MySQL operates on the data. MySQL requires that the user `mysql` should own the `datadir`, while in this example this is not the case in this example. Existing methods that compare the values of these configuration settings across a set of samples cannot detect this error because detecting this error requires reasoning along two dimensions: (i) correlation between multiple configuration settings – correlating `user` and `datadir`, and (ii) validation of the configuration settings in the context of the system (environment) they are used – checking in the system if `datadir` is owned by user `mysql`.

In order to detect the misconfigurations that are not explicitly reflected in their text values, we need to broaden the scope of the analysis and look beyond single configuration settings. Based on our observations (described in Section 2) we propose to widen the analysis to include two important factors: (i) multiple configuration settings and (ii) the execution context or environment.

Cloud computing and related technologies such as virtual machine images have made it possible to easily capture and analyze the execution environment of systems. This ability has enabled researchers to view systems as structured data [11]. Our approach embraces this view of *systems as data* and exploits it to extract the environment information relevant to configuration settings in a system.

1.1 Contributions

Our contributions include (i) a general framework for systematic inclusion of the properties of execution environment, (ii) a tool, called *EnCore*, for efficient learning and effective checking of misconfigurations and (iii) a set of findings from experimental evaluation of the limitations of using off-the-shelf data mining techniques for misconfiguration detection.

Our tool *EnCore* provides a generic configuration data analysis framework that can be readily used for learning misconfiguration rules based on correlation among multiple configuration entries. It also provides features to capture domain knowledge and best practices. The framework has a novel type inference scheme that exploits environment information to derive semantic types for each configuration setting. It also includes an annotator which augments each configuration entry with all its environment related properties. The type information significantly speeds up the rule inference process, and reduces the number of false positives.

We demonstrate the effectiveness of *EnCore* in rules learning and misconfiguration detection from three widely used server applications: Apache, MySQL, and PHP. By systematically exploiting environment information and learning correlation rules across multiple configuration settings, *EnCore* detects 1.6 to 3.5 times more misconfigurations than previous approaches.

To the best of our knowledge, our work is the first one to combine environment information and correlations among multiple configuration settings to automatically generate an effective configuration anomaly detector.

2 Design Principle

In this section, we elaborate the findings that motivate and influence the design choices of *EnCore* . Our design principles derive from observations made in two exercises: the study of configuration entries, and our experience in applying off-the-shelf data mining techniques for misconfiguration detection.

2.1 Characteristics of Configuration Parameters

To understand the use and importance of the environment information and correlations between configuration entries, we manually studied the configuration entries in 4 representative server applications – Apache, MySQL, PHP, and sshd. In this section, we describe our key observations on the correlation characteristics, which validate the prevalence of the phenomena, and motivates *EnCore*. Together with the examples in Figure 1, they show the benefits of exploiting the environment and correlation in misconfiguration detection.

Finding 1: Configuration entries are not isolated, but have relation to the execution environment.

Apps	Total Studied	Env-Related	Correlated
Apache	94	29 (31%)	42 (46%)
MySQL	113	19 (17%)	31 (27%)
PHP	53	16 (30%)	20 (38%)
sshd	57	12 (21%)	29 (51%)

Table 1: **Number (percentage) of configuration parameters that are associated with environment and correlations.** "Total" is the number of entries we examined. "Env-Related" is the number of entries related to the environment. "Correlated" is the number of entries correlated with the others. Apache includes the entries of two main modules: core and mpm; PHP includes the core entries; MySQL's entries are randomly sampled.

In our study, many configuration entries connect the application functionalities to its execution environment, and the values of these configuration entries are associated with the properties of the environment. In other words, these configuration values should not be seen as arbitrary strings. Rather, they reflects the system properties, and have rich semantic information. *This is the key characteristic that distinguishes them from other program variables.*

Although neglected by most existing work, the environment information is critical for detecting a range of configuration errors. Taking the example in Figure 1(a), if a misconfiguration detector catches the fact that *extension_dir* should be a directory in the file system, it can easily identify the one in Figure 1(a) as an anomaly since it is not a directory.

Table 1 shows the statistics of configuration entries whose values refer to system environment objects in the studied applications, where a significant portion, more than 20% of the configuration entries, point to environment objects. With the abundant environment information we can extract from systems, there is great potential for exploiting them in misconfiguration detection.

Finding 2: Many configuration entries are correlated.

From Figure 1(b), we have seen that the setting of one configuration entry depends on the setting of the other entries, or the environment objects. Thus, the correlation information is essential for correctly setting these configuration entries and for detecting errors. Though some types of correlations (e.g., the equality and inequality) can be observed in the textual values, many correlations are often indirect or even implicit. For example, in Figure 1(b), the correlation between `datadir` and `user` goes beyond the textual values and requires one to understand the semantics of the two entries.

As shown in Table 1, around one third to half of the configuration entries have correlation with each other. This indicates that the correlation among configuration entries should be an important component in a misconfiguration detection tool. Unfortunately, impaired by treating configuration values as textual strings, the state-of-the-art detection tools are limited and unable to leverage these correlation information.

	Apache	MySQL	PHP
Original	5773	175	1672
Augmented	9853	555	1942
Binominal	12921	859	2374

Table 2: **The number of attributes generated using data mining methods in the studied software.** "Original" is the number of attributes originated from the configuration files. "Augmented" is the number after environment infomation integration. "Binomial" is the number after conversion from nominal data to binomial.

2.2 Challenges in Applying Data Mining

Since our goal is to extract the configuration correlations, either with the executing environment or between different configuration entries, the straightforward idea is to apply data mining methods to learn the association rules, as proposed in [34] and shown to be effective in other works [45, 47]. We started out by trying to use off-the-shelf data mining methods (association rule mining) on our configuration data set. We tried standard association rule mining algorithms including Apriori [33] and FP-Growth [20] provided by two widely used data mining tool sets – Weka [10] and RapidMiner [8]. In this section, we describe our experience and findings of applying these data mining methods on the configuration data and motivate the design choices of *EnCore*. In the discussion, we mainly use the results from FP-Growth as Apriori does not scale to large data sets [23, 35]).

Finding 3: The state-of-the-art mining algorithms are not scalable to handle the scale of configuration settings.

Configuration files (especially after augmented with system information) usually contain a large number of settings, which are turned into *attributes* in data mining algorithms. Table 2 shows the number of configuration settings (in terms of columns) in the studied software. While the number of unique configuration entries are limited, the mining algorithms treat each occurrence of an entry as a different attribute. Further, the addition of environment information as additional attributes (described in Section 4) increases the total number of attributes. At the same time, Apriori and FP-Growth suffer from the boolean discretization problem [38] – before the mining process, the nominal attributes need to be discretized into boolean values, which causes a dramatic growth in the number of attributes. Table 3 shows the execution time and size of the intermediate frequent item set for Apache, MySQL and PHP, where the number of settings range from 100 to 200+. The experiments were carried on a server with 8 processors and 16GB of memory. Note that when the number of settings increase, both the execution time and the size of frequent item set increase exponentially, making it impossible to reason about the results or even finish the experiments.

Feature selection (and reduction) is a common technique used to attack the scalability problems of data mining methods. There are two approaches: (i) scheme-independent selection and (ii) wrapper methods that involve the machine

entries	Apache			MySQL			PHP		
	attrs	time(s)	freq.	attrs	time(s)	freq.	attrs	time(s)	freq.
100	219	0.15	6K	217	0.13	13.9K	150	0.52	6K
150	436	1.6	173K	286	62	3.8M	235	3.8	542K
175	503	170	14M	315	358	10M	279	106	4.9M
200+	554	OOM	-	343+	OOM	-	336+	OOM	-

Table 3: **Time cost (in seconds) and size of frequent item set with different number of attributes.** With more than 200 attributes, some experiments are terminated with Out Of Memory (OOM) exception. The entries are randomly selected. The number of attributes refer to the scale involved in the mining after adding enviornment attributes and discretization.

learning methods themselves to decide the useful attributes. Neither of these two schemes can directly help reduce the number of attributes in our case. The first method requires the users to select attributes based on domain knowledge [12]. Although both Weka and RapidMiner provide interface for such selection, it is not possible for the users to choose from hundreds to thousands of configuration attributes without any clue on what might be a better combination. The second method, data mining techniques developed to wrap themselves in the attribute eliminating process [16, 37], are mostly useful for classification purposes and are not suited for the correlation learning.

In summary, off-the-shelf data mining techniques do not scale well to the large number of attributes in the configuration data enriched with environment information. We attack this scalability challenge in *EnCore* by using a type-based and template-guided approach to learning. As described in Section 5.1, *EnCore* effectively addresses the scalability problem by restricting the types of the involved attributes and limiting the rule types; it also expresses the various correlations by providing different learning templates.

Finding 4: Frequent-item-sets style relations are not expressive enough to describe domain-specific correlations.

In the data mining community, correlations among different objects are usually described using frequent item sets or linear regression models. While they are sufficient in describing co-occurrence (things likely appearing together) and linear relationship of objects, they cannot capture the complex relations between configurations. For example, a “directory” configuration entry could be concatenated with another “file name” entry to form a “file path” entry; the file specified by the “file path” entry can be owned by the user specified by a “user name” entry. Clearly, these complex domain-specific correlations cannot be expressed by frequent item sets. This motivates the need for new techniques to learn the correlations among configuration settings. As described in Section 5.1, *EnCore* addresses this challenge via rule templates that capture the complex correlation patterns (not actual correlations, but patterns of correlations).

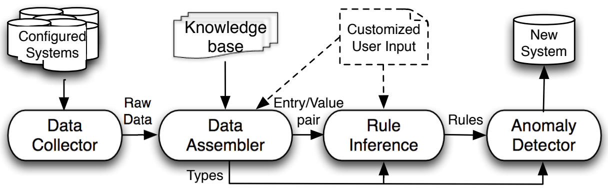


Figure 2: **The architecture of *EnCore*.** Both of the data assembler and the rule inference can be customized by users optionally.

3 System Architecture

Based on the above observations, we propose a framework and tool, called *EnCore*, which incorporates both system environment information and the correlations among multiple configuration entries to effectively and efficiently infer the best-practice rules from the the configured systems. These inferred rules are further applied by a configuration checker to detect configuration anomalies.

As depicted in Figure 2, *EnCore* has four major steps: data collecting, data assembling, rule inferencing, and anomaly detection. This section briefly describes each component.

Data Collector. The data collector gathers the necessary information from the training set (a set of configured systems). Its output is the raw data including all files relevant for analysis, as well as additional environment information in text format. Since we assume that the users have access to the images, privacy is not of concern. But if needed, techniques such as FTN [22] can be used to alleviate the issue.

Data Assembler. The data assembler first parses the collector’s output and converts them (both configuration files and environment data) to uniform key-value pairs. It then infers the type of each configuration entry, which forms the foundation of the *EnCore* analysis framework, as all subsequent analyses incorporate the type information. Each configuration entry is then augmented with the additional environment information collected from the system. The assembler relies on a set of heuristics to infer the predefined types, but it also accepts an optional user input file that specifies heuristics to infer new types. The details are described in Section 4.

After data assembly, the environment data and the original configuration entries are integrated together, and treated equally in the following components. Therefore we use “attribute” to refer to both of them in the following sections.

Rule Generator. *EnCore* learns the best-practice configuration rules deployed in the training set. In order to address the challenges discussed in Section 2.2 and cater the learning process for the characteristics of configuration files, *EnCore* utilizes the rule templates, either predefined or user specified, to guide the learning process. The templates specify the possible relationships (e.g. association rules or file ownership) among configuration entry types, not entry values. Thus, a small set of templates can cover a wide range of concrete rules. Section 7 shows that a total of 79 concrete rules are generated from the 11 predefined templates in 3 applications. The template concept is shown to be also effective in other works [18].

Templates are formalized with a concise grammar to facilitate the creation of new templates by users of *EnCore*. For example, the following template

[A<FilePath>]=>[B<UserName>] *T1*

specifies that 'an entry of type *UserName* is the owner of another entry of type *FilePath*', where the types are inferred in the previous step. Based on the given templates and input key-value pairs, the rule generator iterates over all the possible combinations of attributes and selects concrete rules that best fit the templates. A concrete rule has the placeholders (e.g. A and B in *T1*) filled with concrete attribute names. For example, after learning from the training set that contains images of MySQL, a concrete rule *datadir*→*user* is instantiated from the template *T1*. This rule dictates that the user specified by the *user* entry should be the owner of the directory specified by the *datadir* entry, and helps detect misconfigurations such as the one shown in Figure 1(b). This step also prunes possible false rules with multiple filters. The output of the inferences are the rules that can be utilized by the anomaly detector. More details are in Section 5.

Anomaly Detector. The anomaly detector detects the rule violations in the configurations of the target systems. Its input is the rules, the type information, and the target systems. It outputs warnings whenever it finds an anomaly, such as a violation of a correlation rule, a wrong type, or a suspicious value. The results are ranked based on the type of the violation and the underlying independent statistical model. Since the checking and the learning are cleanly separated, the learned rules can be reused to check different systems. More details are in Section 6.

EnCore can be used either after a new system is configured, or after a failure happens. The user inputs the training set to *EnCore* together with the system to be checked. *EnCore* reports warnings that pinpoint to the potential problematic configuration entries. The usage scenario is similar to that of other misconfiguration detection tools [41, 42], except that the user can also optionally provide *EnCore* with additional types and templates that are specific to the user's environment.

4 Data Assembler

The data assembler takes the raw system files including the target configuration files, as well as the system environment information such as metadata of files in the file system, system configurations, and hardware specifications. It parses these files, infers the types of configuration entries, and augments each configuration entry with the environment information according to its type. The output is a set of well formed key-value pairs. Figure 3 summarizes the process.

4.1 Parsing Configuration Files

The data assembler first converts the configuration files from application-specific format to uniform key-value pairs. We

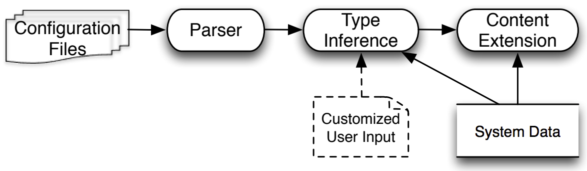


Figure 3: **Data assembling.** In addition to the configuration files, the assembler also takes the system environment data and users' input for type inference and augmenting the configuration entries.

build the parser on top of Augeas [27], a general configuration file parser supporting various software configuration formats. Augeas provides an extensible interface to import other parsers, enabling users to easily import their own configuration parser into *EnCore*. After parsing the configuration files, the assembler stores and organizes all the data in a *.csv* file. Each column represents a structured configuration entry generated by the parser, and each row represents the values of all the entries in a system.

4.2 Inferring Configuration Entry Types

EnCore relies on the type information of each configuration entry for further environment data integration and correlation detection. Similar to previous work on type inferring [31], *EnCore* needs to have the domain knowledge of what the types are and how to determine them.

Without the source code information which is required in [31], the type inference employed in *EnCore* is a novel two-step process that leverages both syntactic patterns of data values as well as the environment information of the system. The first step performs *syntactic matching*, making a crude *guess* at the type of the entry using a predefined syntactic pattern. For example, any string that contains a slash is a potential `FilePath` type. This step is followed by a heavy-weight *semantic verification* that validates the type by checking the corresponding external resources (such as the file system). For example, if an entry value is classified as a `FilePath` type by the first step, the verification searches the full file system meta-data to validate the existence of the path in the file system. The first step prunes away most of the improbable types , making the inference efficient; the second step guarantees the inference accuracy. The combination of the syntactic matching and heavy-weight semantic verification proves to be both effective and accurate (see Section 7.2). Table 4 shows the details of each step and default types *EnCore* infers.

EnCore is able to infer most configuration entry types in the taxonomy of [31]. Certain types such as `TimeInterval` and `Count`, however, are not easily distinguishable, due to the identical syntactic patterns and lack of verification methods. Since our purpose of type inference is to detect type violations and to provide a better foundation for correlation detection instead of precise type analysis, we consider this limitation acceptable – missing this information means possible lower rule detection efficiency as it affects the effec-

Types	Syntactic	Semantic
FilePath	/.+(/.+)*	File System
UserName	[a-zA-Z][a-zA-Z0-9_]*	/etc/passwd
GroupName	[a-zA-Z][a-zA-Z0-9_]*	/etc/group
IPAdress	[\d]{1,3}(.[\d]{1,3}){3}	N/A
PortNumber	[\d]+	/etc/services
FileName	[\w _-]+.[\w _-]+	File System
Number	[0-9]+[.0-9]*	N/A
URL	[a-z]+://.*	N/A
PartialFilePath	/?.+/(.+)*	File System
MIME Types	[\w/-.]+	IANA [3]
Charset	[\w]+	IANA
Language	[a-zA-Z]{2}	ISO 639-1
Size	[\d]+[KMGT]	N/A
Boolean	Values Set	N/A
String	N/A	N/A

Table 4: **Predefined type and inference methods.** Due to space limit, we show here simplified regular expressions used in syntactic matching. More complicated expressions are used for other values, e.g., IPv6 address, which is not shown in the table.

AttrType/Example/Ref.	Augmented Attributes	Type	Value
FilePath datadir=/usr/data File System	datadir.owner	UserName	mysql
	datadir.group	GroupName	mysql
	datadir.type	Enum	dir
	datadir.permission	Permission	664
	datadir.contents	String	dirDes
	datadir.hasDir	Boolean	True
	datadir.hasSymLink	Boolean	True
IPAddress AllowFrom=10.0.1.1 RFC 1918, RFC 4193	AllowFrom.Local	Boolean	True
	AllowFrom.IPv6	Boolean	False
	AllowFrom.AnyAddr	Boolean	False
UserName user=mysql /etc/group	user.isRootGroup	Boolean	False
	user.isAdmin	Boolean	False
	user.isGroup	GroupName	mysql

(a) **Augmented attributes for eligible types by default.** For each configuration entry type, *EnCore* augments it with attributes that reflect system context. Each augmented type is shown with an example entry, the information source in the system (we call it reference) used to compute the augmented values, and the values of the example augmented attributes. Each augmented attribute is assigned with a type.

Env Type	Augmented Attributes
Sys Config	Sys.IPAddress, Sys.HostName, Sys.FSType, Sys.Users
OS Related	OS.DistName, OS.Version, OS.SEStatus
Hw Spec	CPU.Threads, CPU.Freq, MemSize, HDD.AvailSpace

(b) **Augmented attributes for environment info by default.** The attributes are gathered from corresponding files or commands. *EnCore* can be easily customized to consider more data.

Table 5: **Default augmented environment information.**

tiveness of type-based attribute selection (see Section 5), but does not affect the effectiveness. As described in Section 5, we use other ways to accelerate the detection process.

4.3 Environment Information Integration

One of the essential ideas of *EnCore* is to enrich the original data with additional environment information, and take them into consideration in the analysis. *EnCore* selects types inferred in Section 4.2 that carry system semantics, and augments them by attaching new attributes that represent the properties of each type. For example, an entry type of `FilePath` can be augmented with a new attribute that tells whether it is a directory or a regular file. We call these attributes *augmented attributes*. Certain environment data that is independent of the configuration entries are also collected, such as the system hardware specification.

Table 5a shows the types that *EnCore* augments by default. For example, for each entry of type `FilePath`, we attach seven attributes: the owner and group information, whether it's a directory or file, the permission associated with the object, and if it is a directory, the contents of the directory, whether it contains sub directories, and whether it contains symbolic links. This information is retrieved from the file system meta-data collected by the collector. Different types have different attributes, whose values are determined using different sources of information (see Table 5a.)

As shown in the examples in Table 5a, the augmented attributes are directly appended to the original entry names with a dot separator. Table 5b shows the environment information that is independent of the configuration files. These attributes are appended to the existing csv file as additional columns, and are treated equally as other attributes in the rule inference process.

EnCore is designed to be highly customizable. Section 5.3 details how *EnCore* can be customized to infer new types and to include new attributes.

5 Rules Inference

With the extended configuration data as the training set, *EnCore* infers rules using a template-based method. The inferred rules are then written to a file with detailed description of the attributes involved and the relation type, so that they can be used to perform the checking against the target systems.

5.1 Template-Guided Inference Based on Types

To overcome the challenges faced by directly applying data mining techniques (Section 2.2), the rule inference in *EnCore* adopts a template-guided approach. A template aims at capturing the common correlations among the configuration entries and system information, such as the association rules or the file ownership relations in Figure 1(b). Unlike Lint-like checkers [2, 4, 7] which require hand-written rules, *EnCore* only needs a guidance on the type of rules that the users are interested in. It infers concrete rules automatically from the training set.

Figure 4 shows three example templates, their meanings, and the concrete rules inferred from them in different softwares. For instance, the template in Figure 4(a) defines the possible ownership relationship between two entries. Learning from the system images with MySQL, *EnCore* infers the concrete rule that describes the ownership relation between

```
Template: [A<FilePath>] => [B<UserName>]
  Rule:   DataDir => user                                  //MySQL
```

(a) Rule Template: an entry B of type UserName should be the owner of an entry A of type FilePath

```
Template: [A<FilePath>]+[B<FilePath>] => <FilePath>
  Rule:   ServerRoot + LoadModule/arg2 => <FilePath>  //Apache
```

(b) Rule Template: the concatenation of two entries (A of type FilePath and B of type FilePath) forms a file path

```
Template: [A<Size>] < [B<Size>]
  Rule:   upload_max_filesize < post_max_size          //PHP
```

(c) Rule Template: an entry A of type Size should be smaller than an entry B of type Size

Figure 4: **Examples of templates and the concrete rules generated from them.**

entries *DataDir* and *User*, which is used to identify the misconfiguration described in Figure 1(b).

Why use templates? The use of templates brings three benefits to address the challenges faced by direct data mining. 1) It effectively describes the possible types of correlations beyond frequent item set relation; 2) It avoids wasting the computation and resource on patterns that are not likely to exist in the configuration entries; and 3) It makes the learning process "*extensible*": the more templates are used, the larger coverage of possible rules is achieved. Note that the templates can be easily customized, as discussed in Section 5.3.

The template specification uses data type information to restrict the eligible configuration entries. For example, in Figure 4(a), *B* needs to be of type `UserName`, which means when the learning process tries to instantiate the template, it only fills in *B* with the attributes of type `UserName`.

The type information provides an intuitive and effective way of attribute selection, which is critical to solve the scalability problem in the learning phase. In addition, it is a natural base for associating environment information. For example, as described in Section 4, the file permission data are only meaningful to a configuration entry of type `FilePath`.

EnCore provides several predefined templates that make it readily usable. They are shown in Table 6 with their descriptions. Our evaluations are based on these templates.

Rule Inference Process. With the type-based templates, *EnCore* learns rules from the training set. Figure 5 summarizes the workflow of the learning process. In *EnCore*, each correlation is associated with a validation method that determines whether the correlation holds or not. The validation methods are identified by the relation operators(such as $<$ and ==) together with the types of the participating attribute placeholders (such as `A` and `B` in Figure 4; we discuss the correlation interface in detail in Section 5.3 when introducing how to customize correlations).

For each template, *EnCore* tries to instantiate the template by replacing the placeholders with eligible attributes that match the data type specified in the template. *EnCore* iterates over every possible instance of the template and checks whether it is valid using the validation method. If the correlation is valid, *EnCore* regards it as a rule candidate, which would further go into the filtering(Section 5.2). Note that this process is highly parallelizable because there is zero state sharing between each instance computation. As a result, we implement *EnCore* as a multi-process program to achieve high performance.

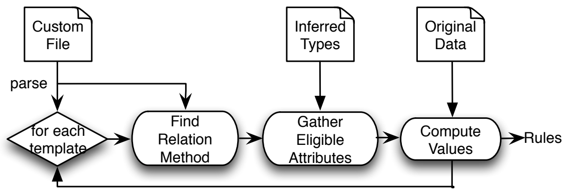


Figure 5: **Workflow of rule inference.**

5.2 Rule Filtering

As accurately as the templates may be defined, like any data-driven method, there are false positives in the resulting rules. *EnCore* use three metrics to filter them. The first two metrics are *support* and *confidence*, which are typical metrics used in association rule detection. *Support* means how many times the frequent item set (in our context, the attributes involved in the rules) occurrs in the data set, and *confidence* means the percentage that the rule is valid.

Configuration entries have a unique characteristic: values of certain attributes are stable in many systems. For example, the warning level in PHP has consistently been set to 10 in our training set. If an entry seldomly changes, the values it carries are not interesting, and the rules involving this entry are likely to be noise. To take this into account, we use a third metric: *entropy* [36]. It measures the diversity of the dataset: its value increases when more diverse values are seen for a given entry. The value of entropy is defined as

$$H = -\sum_{i=1}^{n} p_i \ln p_i, \qquad p_i = N_i/N,$$

where n is the number of different values, p_i is the probability of taking the ith value, N is the times this entry appears in the training set, and N_i is the times it is assigned with ith value. We set a threshold value for each of the three metrics. The threshold for the entropy is denoted by H_t. When there are two values, each having a probablility of of 90% and 10% respectively, it is set to $H_t = 0.325$. For an attribute to be included, it needs to have $H > H_t$, meaning it is more diverse than the minimum threshold. For a rule to be included, all the involved attributes need to be included.

5.3 Customization

EnCore is a fully customizable framework. Users can extend it with different levels of customization using different interfaces. Specifically, users can 1) specify new templates to infer new types of rules; 2) define new types in addition to the default types; and 3) tag new environment information for the environment data integration.

Template	Description
[A<AnyTypeA>] == [B<AnyTypeA>]	An entry should be equal to another entry of same type
[A<AnyTypeA>] = [B<AnyTypeA>]	One instance of an entry should equal to at least one instance of another entry of same type
[A<ExtBoolean>] → [B<Boolean>]	An extended boolean indicates a boolean entry whose extended attribute has boolean value
[A<IPAddress>] < [B<IPAddress>]	An entry of IPAddress is a subnet of another entry
[A<FilePath>]+[B<FileName>]=><FilePath>	Concatenation of a file path entry with a partial file path entry forms a full file path
[A<String>] < [B<String>]	An entry is substring of another entry
[A<UserName>]<[B<GroupName>]	The user name belongs to the group name
[A<FilePath>] ! = [B<UserName>]	The file path is not accessible by the user specified in the entry
[A<FilePath>] => [B<UserName>]	The entry of UserName is the owner of the file path specified in the entry A
[A<Number>] < [B<Number>]	The number in one entry is less then that of the other entry
[A<Size>] < [B<Size>]	The size in one entry is smaller then that of the other entry

Table 6: **Predefined templates with description.** The templates are created based on the common correlations of configuration settings. The operators carry different meanings for different types, and can be overridden by users' customization.

5.3.1 The Customization Interface

A user customizes *EnCore* using a customization file. It has seven sections, each specifying a customizable aspect of the system. Figure 6 shows its format and some sample input. Note that *EnCore* provides predefined types, operators, and templates so that *the customization file is optional.*

As illustrated in Figure 6, each section's name is specified and is prefixed with the "$$" symbol. New type information is specified in the top 3 sections including its name declaration, how to infer it, and how to verify it. To define a new data type, users need to implement the syntactic matching and an optional semantic verification method. The method takes the values to be inferred as input, and returns a boolean value to indicate whether the value is of this type. When an attribute can be inferred for multiple types, customized types have priority over predefined ones, and priority of the customized types is determined by the order they appear in the customization file. The other sections define additional augmented attributes besides predefined ones in Table 5. The augmented attributes are first declared, followed by the methods to compute their values.

The type operator section defines the operators used in the templates, including both aggregation and comparison (see Section 5.3.2). The template section specifies the types of rules to be attempted in the learning.

5.3.2 Customizing Rule Templates

Specifying new rule templates allows users to customize *EnCore*'s rule inference. As discussed in Section 3, a template is a general relation specification, composed of two major parts: the *slots* (the capitalized letter and the type in square brackets in Figure 4) and the *relations* (the plus and arrow symbols in Figure 4). A slot is to be filled in the learning process. It has a name and a data type. The name will be filled with the concrete entry name after a rule is found, while the data type is specified by the user. The data type is used by the rule inference to select eligible attributes.

There are two types of relations. The first defines how different attributes can be aggregated, e.g. string concatenation, arithmetic addition, or any other aggregation methods specified by the users. The second type is for comparison, which defines how values are compared, e.g. "greater than" or "equal to."

```
$$TypeDeclaration
     <NewType>
$$TypeInference
     <NewType> (value): { return True }
$$TypeValidation
     <NewType> (value):{ return True }
$$TypeAugmentDeclaration
     <NewType>.Group
$$TypeAugment
     <NewType>.Group (value):{ return '' }
$$TypeOperator
     <NewType>:<NewType> Operator '<' (v1,v2): { return True }
$$Template
     [A<Types.NewType>] < [B<Types.NewType>] -- 90%
```

Figure 6: **Format of the customization file and a sample input.**

The templates aim at capturing the general correlations among configuration entries and are not specific to any particular software or system. We envision that with more usage of *EnCore*, more templates would be created and shared among users, increasing coverage of rule inference.

5.3.3 Accessible Environment Information

When users are specifying their methods for customization, they may need to access system information. *EnCore* provides users access to all the environment information collected in the data collecting phase, in the form of global variables. They are accessible in any place in the customization file. Table 7 shows the data structures accessible to the users, as well as the sources from which the system gets the information. They are organized in both array lists and maps. For example, with FS.FileList, the user can iterate all the file names (including their paths) in the system. With FS.FileMetaMap, users can retrieve the metadata of the file using the file path as the key.

The programming complexity of each new type inference or template depends on its semantics. Given the easily accessible data provided by *EnCore*, the programs are usually short and straightforward. The predefined type inference methods have 7 to 12 lines of Python code, and the LOC of the predefined template methods range from 4 to 20.

Category	Data Structure	Sources
Files Data	FS.FileList	File System
	FS.FileMetaMap	
Account Info	Acct.UserList	/etc/passwd /etc/group
	Acct.GroupList	
	Acct.UserGroupMap	
Service	Service.Ports	/etc/services
	Service.PortServMap	
Env Variables*	Env.VarValueMap	env
Security	Sec.SELinux	/selinux/*
Hardware**	HW.Cores	/proc/*
	HW.Memory	
	HW.DiskSize	

Table 7: **Data structures for accessing environment information.** *Only available when collecting data from running instances; **The hardware specifications are not available for newly instantiated virtual machine images such as those from Amazon EC2.

6 Anomaly Detector

With the learned rules, *EnCore* looks for potential anomalies in the target systems. It goes through the same data assembling process for the new system as in the learning phase, including parsing, type inferencing, and environment information integration. Then, it checks the following aspects of the target configuration and produces a ranked list of errors.

1. Entry Name Violation. Previous studies show that misspelling is an important source of misconfigurations [31]. If the new system includes entries not seen before, it is likely a misspelled one.

2. Correlation Violation. *EnCore* checks if the target system follows the correlation rules learned from the training set. Since each rule is an expression, the detector of *EnCore* evaluates the expression with the values from the target configuration and reports warnings when violation is found. The rule is ignored if the involved entries are absent in the target configuration file.

3. Data Type Violation. For each entry to be checked, the checker reads its type information inferred from the training set, and gets the corresponding syntactic matching function and semantic verification function. The two functions are used to match and verify the target configuration value. A type violation is reported if the verification or matching fails.

4. Suspicious Values. The detector compares the values of configuration entries from the new system with the values in the training set. It reports a warning if the new value is different from all the previous ones. When multiple entries have unseen values, we adopt the *Inverse Change Frequency* method [42] that gives higher ranks to entries with less diverse values in the training set. Note that the statistical method used here is orthogonal to how rules are learned in *EnCore*: other methods (e.g., those in PeerPressure) can also be adopted here. We chose this simple design and found it satisfy our need. In fact, *with more environment information integrated, more aspects of the configuration entries can be checked against suspicious values.* For example, the detection of the error in Figure 1(a) is directly attributed to the extended attribute of *extension_dir.type* – all the values in the training set have type *directory*, but the value in the target system has type *regular file*. This warning is ranked much higher than other possible suspicious values since its value set in the training set has a cardinality of only 1.

App	Total	Baseline	Baseline+Env	EnCore
Apache	15	4	9	14
MySQL	15	5	14	15
PHP	15	9	12	15

Table 8: **The number of injected misconfigurations detected by *EnCore* in the injection experiment.** "Baseline" is the non-correlation and environment-unawared approach, adopted in most existing work. "Baseline+Env" is non-correlation but environment-aware, i.e., only using the type-based environment infomation integrated by *EnCore*.

7 Evaluation

We evaluate the effectiveness of *EnCore* with the rules learned from public images crawled from Amazon EC2. The experiments are carried out with 3 softwares: Apache Web server, MySQL, and PHP. For each software, we have different number of training data according to the availability of the software on EC2 images: 127 images for Apache, 187 images for MySQL, and 123 images for PHP.

7.1 Effectiveness of Misconfiguration Detection

We perform three experiments to evaluate *EnCore*'s capability of misconfiguration detection. In the first experiment, we inject random errors into correctly configured systems and use *EnCore* to detect the injected errors. In the second experiment, we apply *EnCore* to check against real-world misconfiguration problems. In the last experiment, to see whether *EnCore* is able to detect new configuration errors, we directly apply the anomaly detector to the public images from Amazon EC2 and the virtual machine images from a commercial company's private cloud.

7.1.1 Injected Misconfigurations

For each software, we randomly pick an image that is not in the training set and inject 15 errors with ConfErr [25] into the configuration file. We compare the detection results with the following two approaches: 1) The baseline that does not consider environment information nor correlation information. It resembles the state-of-the-art detection approaches based on value comparison such as PeerPressure [41]. 2) Baseline+Env that enhances the baseline approach with the type-based environment information integration. It takes into account both the configuration values and the environment information, without correlation information.

Table 8 shows the detection results of the three approaches. Compared with *Baseline* and *Baseline+Env*, *EnCore* achieves a significantly higher coverage of errors. The result also shows that the environment information integra-

ID	Software	Problem Description	Info	Rank
1	Apache	Website not granted desired protection because DocumentRoot does not have related Directory	Corr	1(5)
2	PHP	Does not connect to database due to extension_dir pointing to a file instead of the directory	Env	1(1)
3	MySQL	File creation error due to datadir's wrong owner	Env + Corr	1(1)
4	MySQL	Data writing error due to undesired protection from AppArmor	Env	1(2)
5	PHP	Modules not loaded because extension_dir is set to a wrong location	Env	1(1)
6	Apache	Website unavailability because directory contains symbolic links when FollowSymLinks is off	Env + Corr	1(3)
7	Apache	Website visitors are unable to upload files due to the wrong permission set to Apache user.	Env + Corr	1(1)
8	MySQL	Out of memory error due to too large table size allowed in configuration	Env + Corr	-
9	MySQL	Logging is not performed even with relevant entry set correctly due to wrong permission	Env + Corr	1(1)
10	PHP	Failure when uploading large file due to the wrong setting of file size limit	Corr	2(2)

Table 9: **Detection of real world misconfigurations.** The "Info" column describes the information needed to detect the misconfigurations: "Corr" refers to correlation, and "Env" refers to environment infomation. The "Rank" column shows the rank of the real misconfiguration in the warning report, with the total number of warnings shown in "()". "-" means the misconfiguration is missed by *EnCore*. Note that the majority of them require both environment and correlation information.

tion and type detection can greatly enhance existing value comparison based approaches in terms of misconfiguration detection. For example, *Baseline* does not detect wrong file paths, as they usually vary substantially across the training set, but they are captured by *Baseline+Env* and *EnCore*.

Notice that the error injection of ConfErr is within the scope of configuration files and does not touch other system locations. For example, it does not change permissions or ownerships of files specified in the configurations. Thus, the results in this section only reflect *EnCore*'s capability in detecting misconfigurations residing in configuration files.

7.1.2 Real-World Misconfigurations

To evaluate *EnCore*'s ability of detecting real-world misconfigurations, we use the real-world cases described in [46] that are collected from ServerFault (an online user forum focusing on system administration). We randomly sampled 15 problems that are easily reproducible, and reproduced the misconfiguration-induced failures on a new testing image. Table 9 describes these cases, the information required, and the detection results. Among the 15 problems, 8 of them require the environment information and/or the correlation to be detected. The remaining 7 problems depend on either pure configuration entry value checking or unseen configuration name detection and can be detected by both *EnCore* and existing works. In addition, we show two failure cases caused by misconfigurations detected by *EnCore* in EC2 image (details in Section 7.1.3), which are verified to have already caused real world problems.

Table 9 shows that the only misconfiguration missed by *EnCore* is Problem #8. The misconfiguration is that the value of *max_heap_table_size* equals the system memory size, but the system cannot allocate all the space to MySQL. The reason is the lack of hardware information in the training set. The hardware information is intentionally skipped when we crawl the EC2 images, since the images can be instantiated with different hardware configuration at instantiation time. If the data were crawled on a running instance instead of a dormant image, *EnCore* can incorporate such hardware information.

EnCore generates other warnings in addition to the real configuration errors. In the case of Apache, the false warnings are generated mostly from configuration entry name checking. Apache allows nested configuration entries at arbitrary levels. If the combination of the section and entries has not been seen before, a warning will be reported. In Problem #10, the error was ranked No.2 due to another true misconfiguration in the file, which violates a rule with higher confidence.

7.1.3 New Detected Misconfigurations

We collect 120 new images from EC2 which are excluded from the training set and then apply *EnCore* directly on the new images with the rules learned from the training images. It is quite surprising that *EnCore* finds a total of 37 misconfiguration in 25 images. This was not expected because the public images are mostly used as system templates for producing other images and are considered to be correct.

We also apply *EnCore* to 300 images from a commercial private cloud of an IT company using the rules learned from EC2 training images. *EnCore* finds 24 misconfigurations in 22 images. Table 10 summarizes the categories of these misconfigurations. All of them are manually verified to be configuration *errors* that can either cause security holes inside the systems or lead to unexpected behaviors. It is noticeable that the fraction of problematic images in the commercial private cloud is lower than that of EC2. This is expected because they have been deployed in real usage for a long time and should have most problems discovered already.

We also find that many of these new misconfigurations cannot be detected by the software itself because the related rules are not enforced in the source code. This is true even when the undesirable behaviors or security problems caused by the same misconfiguration have already been encountered by the other users [5, 6]. In one case, the log file in MySQL should not be accessible to other users, because it may con-

Source	FilePath	Permission	ValueCompare	Total
EC2	3	10	24	37
PrivateCloud	10	3	11	24

Table 10: **Categories of new detected misconfigurations.** "FilePath" means that file path configuration is missing or set wrongly. "Permission" means the permission configuration is wrong. "ValueCompare" shows the misoncifugraions that violate value comparison rules.

tain sensitive data [5]. Despite the significance of the security vulnerability, MySQL does not report any warning or error, and as a result such misconfiguration remains undetected. Another case is in PHP configuration. The size of the uploaded file is limited by two entries: *post_max_size* and *upload_max_filesize* with the former one having higher priority. Thus, for the latter one to take effect, it needs to be smaller than the former one; otherwise the upload of a file larger than *post_max_size* fails even if the file size is smaller than *upload_max_filesize*.

It is interesting to notice that none of these misconfigurations can be detected without the use of environment and correlation information. For example, the permission violation is detected by the rule found from the 8th template in Table 6, where file path permission is correlated with user names. In this case, it checks the accessibility of the Linux system user "nobody." In summary, the experiments validate the effectiveness and neccessity of these two important factors when dealing with misconfiguration issues.

7.2 Type Inference Accuracy

Table 11 shows the results of type inference (described in Section 4.2). For the entries that do not match any hint or do not pass verification, we assign them with the type String or Number (trivial). It may include the configurations that specify a regular expression or a numerical number.

To verify the results, we manually check the semantics of all the entries and compare them with the inference output. A major source for the false inference is the use of regular expression and wildcards in the configuration files to specify the file names or paths. Also, sometimes the specified file names or paths may not necessarily exist in the system. For example, in Apache users could specify the index file to serve a directory. But the file may actually not exist, and the software handles its absence automatically. Another example is the log files of MySQL that will be created when the software is running. For PHP, the wrong inferences mostly come from the integer values mistakenly determined as Boolean, where all the training images use the values of 0 or 1. Note however, that these happen because we are using the data set from template images of Amazon EC2, which represent pristine, uninitialized states of systems. We believe that the results will be significantly improved if we crawled running production systems that are properly initialized, and have more customized configuration values.

Apps	Entries	NonTrivial	FalseTypes	Undetected
Apache	371	207	14	20
MySQL	131	86	3	11
PHP	249	164	13	8

Table 11: **Data type detection results.** The "Entries" column is the total number of configuration entries. The "NonTrivial" column refers to the types with semantic meanings that are not regarded as "string" or "number." "FalseTypes" / "Undetected" show the number of entries with wrongly detected or undetected types.

Apps	Detected Rules	False Positives
Apache	42	9
MySQL	29	4
PHP	31	10

Table 12: **Detected correlation rules with the filters.**

7.3 Correlation Rule Inference

Table 12 shows the total number of rules inferred from the predefined templates (shown in Table 6). We use the confidence of 90%, support number of 10% of the total number of images in the training set, and the entropy (H_t) of 0.325 (Section 5.2). The threshold values are selected based on the nature of our training set – EC2 images are often used as general template images for the users to customize them for their own needs. Therefore, many of the images' configurations are set as default. In this case, the entropy is usually small. For the same reason, the occurrence of certain optional configuration entries (especially those not appearing in the sample configuration files) is small. As a result, we use a small support number for the purpose of filtering.

A significant source of false rules is undetected types, which causes the comparison of unrelated entries. For example, in Apache, both *MinSpareServers* and *Timeout* are inferred as Numeric type. They do not have correlation, but they are compared in *EnCore*. Since the value of *MinSpareServers* is usually numerically smaller than *Timeout*, it is reported as a rule that "*MinSpareServers* is smaller than *Timeout*".

As described in Section 5.2, besides the standard filters of confidence and support number, we use the entropy threshold. To evaluate its effectiveness in our data set, Table 13 shows the number of false rules filtered in each software.

In our experiments, entropy is mostly effective against the false positives in the correlations related to numeric rules, as well as binomial association rules. The reason is that some of these values are directly derived from the sample configuration files and not changed by the image developers. For example, the *HostnameLookups* directive in Apache is always set to *Off*, and the *min_server_severity* directive in PHP is always set to *10*. The high number of originally reported rules in PHP is mostly caused by the comparison of numeric entries. Since many of these settings are not changed much, the filter would effectively rule them out.

Apps	Original	FP Reduced	FN Introduced
Apache	113	71	7
MySQL	52	23	1
PHP	567	536	1

Table 13: **Effectiveness of the entropy filter.** "Original" shows the number of rules after applying the confidence and support filters. FP (False Positive) Reduced is the number of false rules filtered by entropy filter. FN (False Negative) Introduced is the number of true rules that are filtered.

For the same reason, entropy also filters some true correlations. For example, while *net_buffer_length* should be smaller than *max_allowed_packet* in MySQL, the rule is filtered because all the values of *net_buffer_length* is set as *8K*. Although some may argue that exposing more correlations is more important than suppressing false reports when detecting misconfigurations, given the false positives and false negatives data shown in Table 13, we consider the trade-off worthwhile and beneficial to the end users.

8 Related Work

Misconfiguration Detection. There are generally two types of approaches for misconfigurations detection and troubleshooting: black-box approaches [13, 22, 39, 41, 42, 47] and white-box approaches [14, 15, 31, 32]. The former learns configuration rules from large-scale configuration settings, while the latter infer rules by analyzing source code. Compared with white-box approaches, black-box approaches require no source code or heavy-weight program analysis and can go across the software boundary (which is hard for white-box approaches). Since *EnCore* is a black-box approach, we mainly discuss the related black-box detection approaches in this section.

Strider [42], PeerPressure [41], and [34] detect misconfigurations by comparing configuration values of one system with those of peer systems. Strider uses manually labeled working/failing configuration cases, as well as changing frequency to filter the suspicious entries. PeerPressure enhances Strider with a statistical model and eliminates the need of manual labeling. Ramachandran et al. [34] use both frequency and Google results as ranking heuristics to discover the string dependencies between configuration entries across different components. None of them exploit the environment information, nor the various correlations between configuration entries. The evaluation shows that by examining the whole system software stack and its internal correlations, *EnCore* goes beyond the existing approaches in terms of the capability of misconfiguration detection.

Other works address different aspects of the black-box approaches. FTN [22] focuses on the integrity and privacy issues of the peer contents. AutoBash [39] learns from the peers' problem solution. CODE [47] targets on the configuration access pattern anomaly. *EnCore* complements these work with both environment and correlation information as the new information source.

Configuration Languages. In order to make the configurations more expressive, some software use programming languages in configuration files, including Emacs Lisp, Google's GCL (General Configuration Language) used internally for configuring software, and the Click language that configures Click routers [26]. The results are mixed – while use of programming languages provides stronger expressive power and allows configurations to be analyzed like programs, the added complexity makes it more difficult for end users [9, 28]. The use of programming language does not preclude the need for *EnCore*. Though the type checking of the language may mitigate the burden of type inference of *EnCore*, the need for integrated environment information and configuration rule learning is still desired.

Configuration Testing. Configuration testing tools, such as SPEX [44], ConfErr [25], and KLEE [17], can be used to generate realistic, high-coverage test cases to defend the systems against misconfigurations. Those tools can benefit from *EnCore* since it provides new error injection opportunities such as erroneous environment settings and violations of correlation rules.

9 Conclusion and Future Work

Configuration settings serve as a bridge between applications and their executing environment. Therefore detecting misconfiguration requires reasoning on both sides. We have presented *EnCore*, a misconfiguration detection tool that enriches the plain configuration values with the environment information and unveils their implicit rules and correlations. Based on the configuration rules learned from the enriched data, *EnCore* is able to detect misconfigurations that are beyond the capability of existing tools.

Our future work includes several appealing applications of *EnCore*. First, the idea of integrating environment information can be naturally extended to deal with cross-component misconfigurations: the configuration of other components can be seen as one kind of environment factors. Also, the information integrated by *EnCore* (including both the assembled values and inferred rules) can be used to enhance configuration testing and assist the process of auto-configuration.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank Mona Attariyan for providing us with misconfiguration examples from her research work. This research is supported by NSF CNS-1017784, NSF CNS-1321006, and IBM Graduate Fellowship.

References

[1] Misconfiguration brings down entire .se domain in Sweden. http://www.circleid.com.

[2] Google Code Style Guide. http://google-styleguide.googlecode.com.

[3] Internet Assigned Numbers Authority. http://www.iana.org.

[4] Lint, a C program Verifier. http://www.unix.com/man-page/FreeBSD/1/lint.

[5] Mysql log security. http://www.securityfocus.com /advisories/3803.

[6] PHP configuration error. http://stackoverflow.com/ questions/7754133.

[7] PyLint. http://www.logilab.org/project/pylint/.

[8] RapidMiner. http://www.rapid-i.com.

[9] At what point does a config file become a programming language? http://stackoverflow.com/questions/648246/at-what-point-does-a-config-file-become-a-programming-language.

[10] Weka. http://www.cs.waikato.ac.nz/ml/weka.

[11] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang. Virtual machine images as structured data: the Mirage image library. In *HotCloud*, 2011.

[12] S. Anand, D. Bell, and J. Hughes. The Role of Domain Knowledge in Data Mining. In *Proceedings of 4th International Conference on Information and Knowledge Management (CIKM'95)*, December 1995.

[13] M. Attariyan and J. Flinn. Using Causality to Diagnose Configuration Bugs. In *Proceedings of 2008 USENIX Annual Technical Conference*, June 2008.

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

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

[16] P. S. Bradley and O. L. Mangasarian. Feature Selection via Concave Minimiation and Support Vector Machiens. In *Proceedings of the 5th International Conference on Machine Learning (ICML'98)*, July 1998.

[17] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceeedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

[18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[19] J. Gray. Dependability in the Internet Era, 2001. Keynote presentation at the 2nd HDCC Workshop.

[20] J. Han, J. Pei, and Y. Yin. Mining Frequent Pattern without Candidate Generation. In *Proceedings of the 2000 ACM International conference on Management of Data (SIGMOD'00)*, May 2000.

[21] M. Hong, Z. Lu, and Y. Fuqing. A Component-based software configuration management model and its supporting system. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, May 2002.

[22] Q. Huang, H. J. Wang, and N. Borisov. Privacy-Preserving Friends Troubleshooting Network. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[23] R. J. and B. Jr. Efficiently Mining Long Patterns from Database. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, June 1998.

[24] R. Johnson. More details on today's outage. http://www.facebook.com/notes /facebook-engineering/more-details-on-todays-outage/431441338919.

[25] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, June 2008.

[26] E. Kohler, B. Chen, M. F. Kaashoek, R. Morris, and M. Poletto. Programming language techniques for modular router configurations. Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, August 2000.

[27] D. Lutterkort. Augeas - a Configuration API. In *2008 Linux Symposium*, 2008.

[28] J. Mason. Against The Use Of Programming Languages in Configuration Files. http://taint.org/2011/02/18/001527a.html.

[29] K. Nagaraja, F. Oliveria, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, December 2004.

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

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

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

[33] A. Rakesh and S. Ramakrishnan. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, September 1994.

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

[35] B. S., M. R., U. J., and T. S. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, May 1997.

[36] C. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, July 1984.

[37] K. Smets and J. Vreeken. SLIM: Directly Mining Descriptive Patterns. In *Proceedings of 2012 SIAM International Conference on Data Mining (SDM'12)*, April 2012.

[38] R. Srikant and R. Agrawl. Mining Quantative Association Rules in Large Relational Tables. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, June 1996.

[39] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the 21st ACM Symposium on Operating Sytems Principles (SOSP'07)*, October 2007.

[40] S. Traugott and J. Huddleston. Bootstrapping an Infrastructure. In *Proceedings of the 13th Systems Administration Conference (LISA'99)*, November 1999.

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

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

[43] M. Welsh. What I wish systems researchers would work on. http://http://matt-welsh.blogspot.com/2013/05.

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

[45] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the 2009 Symposium on Operating Systems Principles*, 2009.

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

[47] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowsky, and A. Kumar. Context-based Online Configuration-Error Detection. In *Proceedings of 2011 USENIX Anuual Technical Conference*, June 2011.