Towards Proactive Misconfiguration Checking

Abstract. Software failures resulting from configuration errors have been commonplaces as modern software systems become increasingly larger and more complex. Although existing efforts aim to overcome misconfiguration problems by either building post-failure error diagnosis tools or hardening systems based on misconfiguration reports, these retroactive solutions, nevertheless, require human (*e.g.*, user) intervention, thus leading to impractical administration cost. Complementing existing post-failure forensics, this paper proposes, ConfigC, a *proactive* misconfiguration checker. With ConfigC in hand, the users of software systems can submit their configuration files to ConfigC for misconfiguration detection *before* deploying them in the real software settings. At the heart of ConfigC lies a machine learning mechanism, which learns rules and constraints from pluggable correct sampling configuration files, and then detects errors violating the learned constraints. ConfigC mainly offers three significant benefits. First, there is no any extra burden added to users who only need to simply use ConfigC as a "seamless" pre-checker for their configurations. Second, because many misconfiguration errors have been eliminated by ConfigC, the workloads of post-failure forensics in runtime are significantly reduced, thus making these tools truly practical. Finally, since there have been many correct but not specific example configuration files in practice, increasingly more samples can be freely added to ConfigC, thus giving ConfigC evolutionary capability.

1 Introduction

Configuration error is one of the most important root-causes of modern software system failures [5, 6]. In practice, misconfiguration problems may result in security vulnerabilities, application crashes, severe disruptions in software functionality, and incorrect program executions [5, 7, 8]. Although several efforts have been proposed to automate configuration error diagnosis after failures occur [1–4], they rely on manual ways to understand and detect the failure symptoms. Such solutions typically incur high administration cost (*e.g.*, requiring users to write pattern-matching scripts) and are unreliable (*e.g.*, some errors may show no user-visible symptoms). These shortcomings often lead to prolonged delays between the occurrence and the detection of errors, thus causing unrecoverable damage to system states.

We expect to explore a fundamentally different means to tackle the misconfiguration problem. Rather than using retroactive approaches to deal with configuration errors after the software failures occur, we would like to *proactively* prevent misconfiguration problems *before* users first deploy the configuration files to their system settings. In other words, we aim to offer users a configuration error filter in order to significantly minimize the probability of misconfigurations in the system runtime.

Based on the above intuition, this paper presents ConfigC, a proactive misconfiguration checker which takes users' configuration files as input and then reports potential configuration errors to the users before they deploy their systems according to these configurations. At the heart of ConfigC lies a machine learning mechanism, which learns rules and constraints from pluggable correct sampling configuration files (*i.e.*, for the training purposes), and then detects errors violating the learned constraints.

ConfigC mainly offers users three significant benefits. First, there is no any extra burden added to users who only need to simply use ConfigC as a "seamless" pre-checker for their configurations. Second, because many misconfiguration errors have been eliminated by ConfigC, the workloads of post-failure forensics in runtime are significantly reduced, thus making these tools truly practical. Finally, since there have been many correct but not specific example configuration files in practice, increasingly more samples can be freely added to ConfigC, thus giving ConfigC evolutionary capability.

2 Motivating Examples

Fig. 1 presents misconfiguration examples in real-world that we aim to address. These examples are extracted from an existing experience study on configuration errors [6].

3 ConfigC Design

XXX: [ConfigC Design:

- Present an architecture of ConfigC with a fig. Briefly describe how it works (step by step).
- Detail learner part
- Merge
- Check
- Limitations

]

A rule R is added to the set of all rules, if ∃ learning file f s.t. R(f) is non vacuously true That rule R is then removed from the set of all rules, if ∃ learning file f st R(f) is false.

This is accomplished in two passes. First we collect all possible rules for every file. Then we merge the all the rules to create our final set. This conviently gives rise to an embarresingly parallel situation, which Haskell allows us to easily take advantage of by using the parallel mapping library parmap.

```
  potentialRules = parmap findAllRules learningSet.
  finalRules = foldl1 mergeRules potenialRules
\begin{lstlisting}


Each rule is of the Attribute typeclass, which means a rule must support the fol
\begin{lstlisting}
class Attribute r where
```

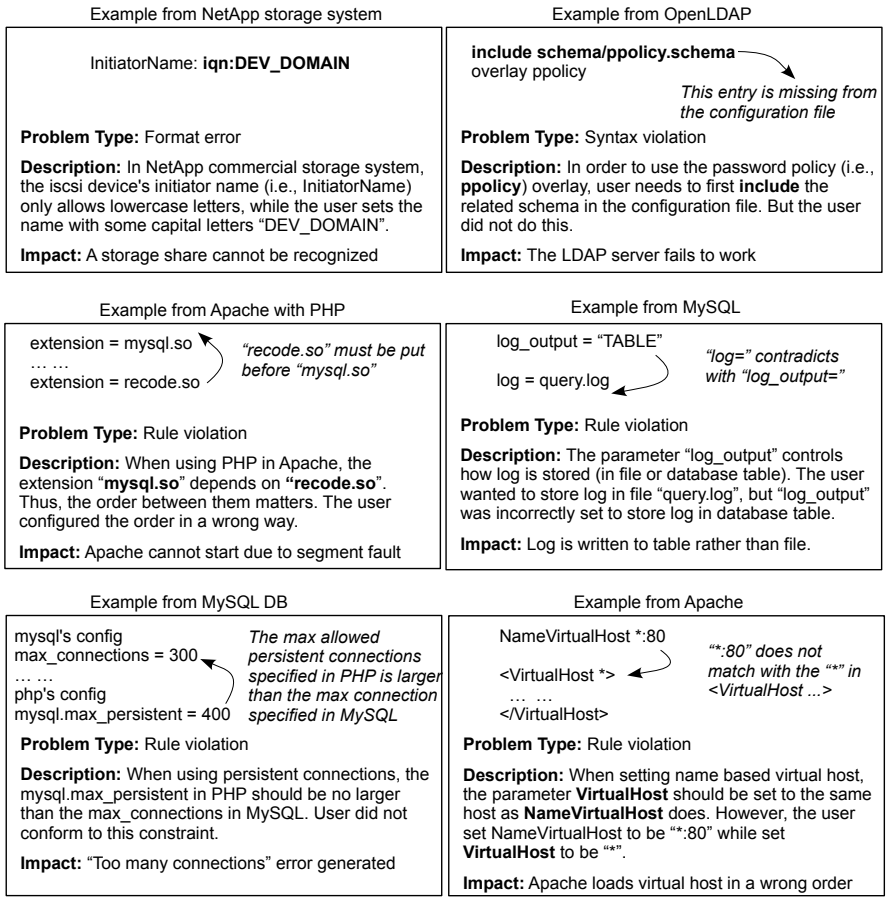

Fig. 1: Motivating examples.

```
learn :: ConfigFile Common -> [r]
merge :: [r] -> [r] -> [r]
check :: [r] -> ConfigFile Common -> Error
```

3.1 learn

For a single given file, we take very line ordering to be a rule.

3.2 merge

Then when merging these sets of rules, we take the intersection of the rules infered on the individual files. Maybe we could also do something like only taking rules that show up multiple times.

3.3 check

To check a file by using a rule set, we simply take all the rules that are releveant to the user's file. Rules that are relavent are the ones where both parts of the ordering are present. We learn the rule set for the user file, and every rule in the learned set must be present in the user file.

4 Implementation and Evaluations

XXX: [evaluation here ...]

5 Related Work and Conclusions

XXX: [conclude here ...]

References

1. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Oct 2010)
2. Su, Y., Attariyan, M., Flinn, J.: AutoBash: Improving configuration management with operating systems. In: 21st ACM Symposium on Operating Systems Principles (SOSP) (Oct 2007)
3. Wang, H.J., Platt, J.C., Chen, Y., Zhang, R., Wang, Y.: Automatic misconfiguration troubleshooting with PeerPressure. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
4. Whitaker, A., Cox, R.S., Gribble, S.D.: Configuration debugging as search: Finding the needle in the haystack. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
5. Xu, T., Zhou, Y.: Systems approaches to tackling configuration errors: A survey. ACM Comput. Surv. 47(4), 70 (2015)
6. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: 23rd ACM Symposium on Operating Systems Principles (SOSP) (Oct 2011)
7. Yuan, D., Xie, Y., Panigrahy, R., Yang, J., Verbowski, C., Kumar, A.: Context-based online configuration-error detection. In: USENIX Annual Technical Conference (USENIX ATC) (Jun 2011)
8. Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., Zhou, Y.: Encore: Exploiting system environment and correlation information for misconfiguration detection. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (Mar 2014)