Static Detection of Silent Misconfigurations with Deep Interaction Analysis

JIALU ZHANG, Yale University, USA
RUZICA PISKAC, Yale University, USA
ENNAN ZHAI, Alibaba Group, USA
TIANYIN XU, University of Illinois at Urbana-Champaign, USA

The behavior of large systems is guided by their configurations: users set parameters in the configuration file to dictate which corresponding part of the system code is executed. However, it is often the case that, although some parameters are set in the configuration file, they do not influence the system runtime behavior, thus failing to meet the user's intent. Moreover, such misconfigurations rarely lead to an error message or raising an exception. We introduce the notion of silent misconfigurations which are prohibitively hard to identify due to (1) lack of feedback and (2) complex interactions between configurations and code.

This paper presents ConfigX, the first tool for the detection of silent misconfigurations. The main challenge is to understand the complex interactions between configurations and the code that they affected. Our goal is to derive a specification describing non-trivial interactions between the configuration parameters that lead to silent misconfigurations. To this end, ConfigX uses static analysis to determine which parts of the system code are associated with configuration parameters. ConfigX then infers the connections between configuration parameters by analyzing their associated code blocks. We design customized control- and data-flow analysis to derive a specification of configurations. Additionally, we conduct reachability analysis to eliminate spurious rules to reduce false positives. Upon evaluation on five real-world datasets across three widely-used systems, Apache, vsftpd, and PostgreSQL, ConfigX detected more than 2200 silent misconfigurations. We additionally conducted a user study where we ran ConfigX on misconfigurations reported on user forums by real-world users. ConfigX easily detected issues and suggested repairs for those misconfigurations. Our solutions were accepted and confirmed in the interaction with the users, who originally posted the problems.

CCS Concepts: • **Software and its engineering → Software configuration management and version control systems**.

Additional Key Words and Phrases: Silent Misconfiguration, Misconfiguration Detection, Configuration Specification

ACM Reference Format:
Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static Detection of Silent Misconfigurations with Deep Interaction Analysis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 140 (October 2021), 30 pages. https://doi.org/10.1145/3485517

1 INTRODUCTION

Software misconfigurations today have been one of the most common causes of service failures [Xu and Zhou 2015]. For example, almost all the mainstream cloud providers, including Amazon [ama

Authors' addresses: Jialu Zhang, Yale University, New Haven, Connecticut, 06511, USA; Ruzica Piskac, Yale University, New Haven, Connecticut, 06511, USA; Ennan Zhai, Alibaba Group, Bellevue, Washington, 98004, USA; Tianyin Xu, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801, USA.

2475-1421/2021/10-ART140
https://doi.org/10.1145/3485517

2017], Facebook [Spangler 2019], Google [goo 2018], and Microsoft [mic 2014], have experienced significant outages resulting from misconfigurations. In addition, an earlier survey [Yin et al. 2011] on misconfiguration in commercial and open source software shows that misconfigurations are the most common root causes of "high-severity" issues. Misconfigurations are responsible for 31% of server downtime issues, compared to just 15% for software bugs.

One of the key reasons for the increase in problems caused by misconfigurations is an increase in software complexity [Yin et al. 2011]. For example, widely-deployed software systems, *e.g.*, Apache, Hadoop, and MySQL, have more than 300 tunable configuration parameters for users. Such complexity in software configurations presents significant challenges for system management.

Research into misconfiguration detection has been the focus of the system and software engineering research community for a long time [Attariyan et al. 2012; Attariyan and Flinn 2010; Mehta et al. 2020; Nadi et al. 2015; Su et al. 2007; Sun et al. 2020; Wang et al. 2004, 2003; Xu et al. 2016a, 2013; Yin et al. 2011; Yuan et al. 2006, 2011; Zhang et al. 2014; Zhang and Ernst 2013; Zhang et al. 2021]. At the same time, the verification community has overlooked this important problem, as modern verification techniques and tools inherently rely on the existence of a formal specification. Therefore, the first step to take towards the verification of configuration files should be specification formulation—the specification is a set of rules expressing the mutual connections of configuration parameters. Although there have been several attempts to automatically derive the specification for configuration checking in the past years, these tools are limited in the scope of the properties that they can infer. Specifically, there is a class of tool that learns simple specification by using configuration files as training sets [Mehta et al. 2020; Santolucito et al. 2017, 2016]. The other type [Chen et al. 2020; Rabkin and Katz 2011a; Xu et al. 2016a, 2013] infers coarse-grained information by analyzing the source code of system programs; however, they only analyze the parts of the system implementation where a configuration parameter is consumed, deriving this way the properties about the scope of a parameter. *The main obstacle of the existing tools, thus, is that they do not analyze the system program as a whole, missing the complex interactions between multiple configuration parameters and the code affected by them.*

The focus of this paper is to employ program analysis to derive such non-trivial interactions between configuration parameters. Motivated by real-world examples, we show that the rules that we inferred can help us detect a special class of misconfigurations that we call *silent misconfigurations*. We chose this name because these misconfigurations typically occur without any helpful system logs or messages. Those misconfigurations have not been systematically studied before, though they commonly appear in practice and the users repeatedly ask for help on public forums (Section 2.1). Such misconfigurations typically happen when the user explicitly modifies a configuration parameter c, but this modification does not have any effect due to the implicit influence of other "seemingly unrelated" configuration parameters. Such implicit interactions cannot be detected by existing tools because those tools either analyze only the configuration files or analyze only the system source code. To detect such convoluted silent misconfigurations, the configurations and the source code that affected by the configurations have to be analyzed jointly—silent misconfigurations are mainly the result of code affected by one configuration *overwriting* the other. To capture that fact, we need to understand the interactions of the source code and configurations, which existing tools are short of.

We developed a tool, called ConfigX, that uses static analysis to determine which parts of the code are affected by which configurations. ConfigX analyzes the interactions between the configuration-related code blocks with customized control- and data-flow analysis to derive specifications. Additionally, we conduct reachability analysis to eliminate spurious rules to reduce false positives.

An example of a silent misconfiguration. The following example shows a real-world silent misconfiguration collected from Stack Overflow [apa 2012]. The user wanted to allow access to only one country but also to exclude some proxies within that country. The user changes the following lines in the Apache web server configuration file, explicitly stating their intent:

```
<Limit GET POST>
    order deny,allow
    allow from ...country.com
    deny from ...proxy.country.com
</Limit>
```

However, instead of producing the `HTTP 403 Forbidden` message, the specified proxy was allowed to access the web server. This misbehavior constitutes a severe security threat and is the exact opposite of what the user intended. Running our tool ConfigX on the reported configuration file will report that the configuration parameter `deny from` has no effect: the system behavior remains the same independently if this option is present or removed from the configuration file.

In this case, the entry `order` plays a key role: it decides the order in which the functions `deny` and `allow` are executed. If the value of `order` is set to `deny,allow`, any IP address matching the value specified in the entry `deny from` gets a temporary return value `deny`. After that it checks the entry `allow from`, and if it matches, it receives the return value `allow`, that overwrites the previous value. Our tool automatically derived the rule that states exactly that:

```
AdvOrder((Order = deny,allow), Allow, Deny)}
```

Silent misconfigurations are hard to detect. Detecting silent misconfigurations is challenging, not only because they do not have useful logs, but also because they require users to understand the interactions between configurations and the code that they affect. Existing systems cannot detect silent misconfigurations because (1) they rely on accurate error messages to detect potential misconfigurations [Sun et al. 2020; Xu et al. 2016a], (2) they fail to analyze interactions between configuration parameters [Wang et al. 2004; Xu et al. 2016a], or (3) they do not look into source code [Mehta et al. 2020; Santolucito et al. 2017, 2016; Zhang et al. 2014].

Silent misconfigurations are common. As we report in Section 2, we identified more than 100 reports on silent misconfigurations for systems like Apache, vsftpd and PostgreSQL. Based on the reported issues, we noticed that the main problem is that the system behavior does not correspond to intended configurations, and the system silently "went wrong" without error messages or other observable symptoms. In addition, we deduced that silent misconfigurations are often caused by incorrect interactions between configurations.

Our approach: ConfigX. This paper is the first approach towards systematically detecting silent misconfigurations. We developed a tool called ConfigX. ConfigX consists of two components: analyzing and checking. The analyzing component of ConfigX automatically derives the specification of configurations from the system source code. This specification is given as a set of rules. The analyzer consists of three different parts. We first compile the system source code, using the LLVM compiler, into LLVM IR [Lattner and Adve 2004]. We next apply a field-sensitive static analysis to establish a connection between the LLVM IR and the configurations. Our static analysis maps the configurations to variables in LLVM IR. In the second part, we derive interactions between configuration values using customized control- and data-flow analyses. The third part is a reachability analysis, which eliminates the spurious rules derived during the previous parts.

Once there is a specification, ConfigX checks for silent misconfigurations in a given configuration file. Since a configuration file might not set all the configuration parameters, we first augment the configuration file by explicitly assigning the system defaults to the configuration parameters that do not occur in the file. Next, we generate the intermediate representation used in ConfigX. Finally, the ConfigX checker inspects whether the augmented intermediate representation violates any of the interactions derived from the ConfigX analysis.

Main technical challenges in designing ConfigX. The main technical challenge in detecting silent misconfigurations is to analyze the complex interaction of source code blocks and use this rich comprehensive semantics to derive complex interactions between configurations, extending this way significantly the class of misconfiguration that can be automatically analyzed and detected. State-of-the-art tools [Chen et al. 2020; Xu et al. 2016a, 2013] only check configuration values themselves and do not further analyze the source code affected by the configuration values, making them fundamentally limited to detect the semantics-related silent misconfigurations.

We used the following techniques in ConfigX that were not used or considered previously in existing tools:

- We designed a comprehensive analysis to capture and analyze the interactions between code blocks related to configurations.
- We proposed reachability analysis to systematically prune out the detected spurious rules in our rule learning process. While previous work uses the statistical method [Xu et al. 2013] to reduce false positives, our reachability analysis is sound.
- Since silent misconfigurations are either syntax-related or semantics-related, we included the system defaults in our analysis—this way our analysis is a combination of source code and configurations.

Evaluation. To extensively evaluate our tool, we have run ConfigX on five different datasets across three widely used systems, Apache, vsftpd and PostgreSQL. First, we ran our tool on the dataset collected from Stack Overflow. ConfigX is able to detect silent misconfigurations even when full configurations are not available. In addition, we also ran ConfigX on the publicly available Apache dataset used by previous work [Xu et al. 2015]. We further created three datasets by crawling configuration files for Apache, vsftpd and PostgreSQL from Github repositories. ConfigX detects more than 2200 silent misconfigurations in 457 configuration files in five datasets in less than an hour. The time was mainly spent in the analyzing phase, while the checking phase takes only negligible time.

To additionally evaluate how useful ConfigX is in real world, we conducted a user study on configurations appearing on Github and Stack Overflow. We ran ConfigX on those configurations and reported the detected silent misconfigurations to the users. The user feedback is very positive. Most users immediately confirmed the silent misconfigurations reported by ConfigX, and used our suggestions to correct their configuration files. Altogether, we corrected fourteen real-world silent misconfiguration problems. Our experience interacting with the real-world users confirmed that silent misconfigurations appear often in practice and are hard to detect.

In summary, we make the following contributions:

- We identify a new class of misconfigurations, termed *silent misconfigurations*. We show that silent misconfigurations are common in practice but difficult for existing techniques to handle.
- We developed a tool named ConfigX that automatically derives complex interaction specifications between configurations by analyzing the interactions of code related to configurations; ConfigX uses the specifications to detect silent misconfigurations.
- We empirically evaluated ConfigX on real-world silent misconfiguration issues as well as on five public datasets across Apache, vsftpd and PostgreSQL. ConfigX detects more than 2200 silent misconfigurations in 457 configuration files in less than an hour.

- We conducted a user study to report the detected silent misconfigurations back to real-world users. Most users appreciatively confirmed the validity of the detected issues and quickly fixed them in their codebase with the help of ConfigX.

2 UNDERSTANDING SILENT MISCONFIGURATIONS

We present our study on real-world silent misconfigurations in three mature, widely used open-source server systems, the Apache HTTP server [apa 2021a], vsftpd [vsf 2021], and PostgreSQL [pos 2021]. We describe the common patterns of silent misconfigurations and give some concrete examples. We further discuss the implications that drive the design of ConfigX.

2.1 Methodology

We collected 48 user-reported issues caused by silent misconfigurations from two data sources, Stack Overflow [sta 2021] and Server Fault [ser 2021], two Q&A forums where configuration-related issues are commonly reported. We identify silent misconfigurations by looking for issues in which users reported missing modules/features/behaviors due to misconfigurations. For example, in one post [StackOverflow #6070335 2011], the question was "*How to retain original request URL on mod_proxy redirect?*" and the correct answer was "*If you are using* `mod_proxy`*, disable* `ProxyPreserveHost` *in the Apache configuration.*" Specifically, we selected posts that contain the keywords such as "configuration", "silent", "module", "enable", "disable", etc. in either the question or any of the answers. We do not consider unanswered or unvoted questions, because the root causes may not be determined. We manually analyze each post to understand the silent misconfiguration, including the configuration pattern and the source-code manifestation.

We focus on Apache, vsftpd, and PostgreSQL. The studied software programs are widely used and their configuration design represents the state of the practice. Moreover, these three programs are extensively studied [Attariyan et al. 2012; Xu and Zhou 2015; Yin et al. 2011]. In total, we collected 27, 11, and 10 silent misconfiguration issues for Apache, vsftpd, and PostgreSQL, respectively.

Our data is available at: https://doi.org/10.5281/zenodo.4697690.

2.2 General Findings

In this section, we present our analysis of silent misconfigurations in Apache. Table 1 lists 27 real-world silent misconfigurations reported by Apache users. Silent misconfigurations in vsftpd and PostgreSQL are manifested in the same vein.

Finding 1: *The majority (74.0%) of silent misconfigurations are caused by interactions between multiple configuration parameters and their values.*

Inspecting the source code that uses the configurations, we find that this interaction is due to the complex interaction of source code blocks that are decided by configurations. All these 19 cases have the same root cause: the effect of one configuration was either disabled by other configurations or overwritten by the code affected by other configurations.

However, none of the existing work [Santolucito et al. 2017; Xu et al. 2013; Yin et al. 2011] analyzed the source code affected by configurations; therefore, they cannot detect silent misconfigurations. Specifically, existing work on configuration dependencies [Chen et al. 2020; Xu et al. 2013] only focuses on analyzing the dependencies between configuration values themselves, but does not consider the *interactions* between source code affected by the configurations.

As modern systems like the ones studied have hundreds of tunable configuration parameters, identifying interactions between configurations is an error-prone and tedious. Expecting users to dig into the source code to understand interactions between configurations is unrealistic—it defeats the purpose of having configurations in the first place [Xu et al. 2015, 2013].

Table 1. Apache silent misconfiguration study results. In the root cause column, "interaction", "syntax", and "one-off" refer to interactions among multiple configuration values, syntax error (incorrect configuration values), and one-off cases that do not reveal common patterns, respectively.

Post ID	Feedback	Root Cause	Patterns
519272	No	Syntax	Unmatched Guard
213916225	Useless	Syntax	Unmatched Guard
36448791	No	Syntax	Missing Module
20127138	Incomplete	Syntax	Missing Module
41831003	Incomplete	Syntax	Missing Module
10994650	No	Interaction	Miss Handling Default
32401502	No	Interaction	Miss Handling Default
55942751	No	Interaction	Miss Handling Default
42650086	Useless	Interaction	Miss Handling Default
7748595	No	Interaction	Miss Handling Default
21263746	No	Interaction	Miss Handling Default
13277968	No	Interaction	Miss Handling Default
45306092	No	Interaction	Miss Handling Default
43239190	Useless	Interaction	Miss Handling Default
13712283	No	Interaction	Miss Handling Default
26097825	No	Interaction	Miss Handling Default
21338450	No	Interaction	Implicit Overwrite
7093385	No	Interaction	Implicit Overwrite
16340	No	Interaction	Implicit Overwrite
6070335	No	Interaction	Implicit Overwrite
9943042	No	Interaction	Advanced Ordering
18392741	Useless	Interaction	Advanced Ordering
9507645	No	Interaction	Advanced Ordering
24728814	No	Interaction	Advanced Ordering
4400154	No	Interaction	Advanced Ordering
39143631	Incomplete	One-off	N/A
47795431	Useless	One-off	N/A

This finding shows that we need a systematic analysis that accurately captures the interactions between configurations.

Finding 2: *A significant number (18.5%) of silent misconfigurations are caused by a new kind of guard-related syntax errors in configuration files.*

Guards are conditions held on configurations. Configurations are valid if and only if the guards hold. If the guards are missing or unsatisfied, the configuration will not be passed as a parameter to the system, even if users specify the configuration parameter in the configuration file. However, none of the existing work [Santolucito et al. 2017; Xu et al. 2013; Yin et al. 2011] supported guards, and thus they failed to report these syntax errors.

This finding showed that we need a static analyzer to check the guards on configurations to users, and proactively detect guard-related syntax silent misconfigurations before any problematic configurations are passed to the system.

Finding 3: *Most (70.4%) silent misconfigurations happen without any system messages or logs. None of our observed error messages were useful.*

Particularly, for 19 out of 27 (70.4%) of the cases, there was no feedback, e.g., logs ("No" in the Feedback column of Table 1). For the rest of the eight cases, though some of them indeed

Table 2. The categories of silent misconfigurations. Unmatched guards and missing module are called syntax-related silent misconfigurations; Miss handling default, implicit overwrite, and advanced ordering are semantics-related silent misconfigurations.

Patterns	Descriptions
Unmatched guards	"Guard configuration parameters" should be set together in pairs (defined in §3). In a typical unmatched guard misconfiguration, one of the paired configuration parameters is missing.
Missing module	In some cases, guards require a specific module to be enabled. However, this needed module is not explicitly loaded in the configuration files, making the guard become a silent misconfiguration.
Miss handling default	When a configuration parameter is not explicitly set in the configuration files, the system uses its default value. However, this default value might disable some other configurations that are explicitly set, which will result in a silent misconfiguration.
Implicit overwrite	Two configuration parameters can share non-trivial interactions such that setting them to certain values will cause that one configuration parameter is overwritten by another, making it this way obsolete and resulting in a silent misconfiguration.
Advanced ordering	A silent misconfiguration based on non-trivial interactions between three configuration parameters. Setting one parameter to a certain value might cause implicit overwrites between the other two parameters, resulting in a silent misconfiguration.

return error messages to users, none of them are useful. Our criterion of usefulness is that the log message needs to include the root-cause configuration parameters. For example, one post asked why `mod_rewrite` was not working [StackOverflow #43239190 2017]. The root cause was a silent misconfiguration that violates complex interactions between configuration parameter `rewriteBase` and parameter `rewriteRule`. However, the error message that Apache returned is "*Use `LimitInternalRecursion` to increase the limit if necessary.*" This message is not helpful for understanding and fixing the silent misconfiguration and is in fact actively misleading.

2.3 Patterns and Examples

We classify silent misconfigurations into five patterns, as shown in Table 2. This section presents examples of each pattern along with methods for detecting misconfigurations of that pattern.

Pattern 1: Unmatched guards. Guards are conjunctions of conditions held on configurations. Modern software such as Apache has introduced guards to support conditional configurations. Configurations are valid if and only if the guards held on them are satisfied.

However, users sometimes specify guards incorrectly, and this results in silent misconfigurations. In our experience, we have found two patterns of syntax-related silent misconfigurations caused by misuse of guards. The first pattern is the unmatched guard. This is the only pattern that we have identified that does not cause any harm per se, but it is an example of ill-formed code and we issue a warning. For example, any guard that begins with `IfModule ..._module` should be paired with another guard `/IfModule`. In the following configuration snippet, the guard `/IfModule` is incorrectly introduced here without a matched `IfModule ..._module`, making the configuration syntactically ill-formed.

```
</IfModule>
    ServerAdmin david@vizion2000.net
    ServerName dns1.vizion2000.net
```

We show another example that triggers a warning. In the following configuration snippet, there is no configuration under the guard `alias_module` because the code is commented out. Introducing a guard but providing no configuration under the guard does not change any system behavior. Therefore, this guard is empty.

```
<IfModule alias_module>
```

```
        #ScriptAlias /cgi-bin/ ...
    </IfModule>
```

Pattern 2: Missing module. The second pattern of syntax-related silent misconfigurations is the missing module pattern. In the following configuration snippet, the guard requires the `alias_module`; however, the module is not explicitly loaded in the configuration files. To detect the misconfiguration, we will need to parse configuration files and check if the guards are unpaired or any module is missing in its customized syntax analyzer (we describe ConfigX's implementation in Section 4.1).

```
    # missing alias_module (modules/mod_alias.so)
    <IfModule alias_module>
        ScriptAlias /cgi-bin/ "/usr/local/apache2/cgi-bin/"
    </IfModule>
```

Pattern 3: Miss handling configuration defaults. The configuration defaults determine system default behaviors if no configurations are set explicitly. Software often uses the preset default values, unless the user explicitly changed some configuration parameter values. However, those user-customized changes can conflict with the rest of the system defaults, which results in misconfigurations. In the following example from a user forum, the user asks help in redirecting all the URLs that start with www to URLs without www. However, Apache failed to redirect URLs although the following configuration parameters explicitly state that (using some regular expression):

```
    RewriteCond %{HTTP_HOST} ^www\.(.*)$ [NC]
    RewriteRule ^/(.*)$ http://%1/$1 [R]
```

The root cause of this misconfiguration is the interaction between `RewriteCond` and `RewriteRule` parameters and some other parameters that do not even seem to appear in the configuration file, leading to miss handling configuration defaults. To enable the usage of `RewriteRule`, the user must explicitly set the configuration `RewriteEngine` to `On`. The default value of `RewriteEngine` is `Off`, and since it did not appear in the configuration file, Apache was using its default value. This error can be corrected only after the user explicitly adds `RewriteEngine On` into the configuration file.

In fact, having both parameters `RewriteEngine Off` and `RewriteRule` together is also recommended in the Apache user manual [apa 2021b]. This saves users from manually commenting out all the instances of URL redirection like `RewriteCond` and `RewriteRule`. However, users do not have the same level of expertise as Apache developers, and in this example, they fail to understand and handle that complexity.

To detect silent misconfiguations, we analyze the interaction between the code blocks enabled by `RewriteRule` and `RewriteEngine`. ConfigX detected that the code blocks enabled by `RewriteRule` will only be executed if `RewriteEngine` is set to `On`. After learning the interaction, our tool derives the following rule:

```
    Disables(RewriteEngine == On, RewriteRule)
```

Pattern 4: Implicit overwrite. Consider the following configuration snippet (from a real-world issue [apa 2014]):

```
    SSILastModified on
    XBitHack full
```

The user intends to set the Last-Modified header in HTTP responses by introducing two key-value pairs. In this example, `SSILastModified On` is a silent misconfiguration, because independently whether `SSILastModified` is on or not, it does not change program behavior.

By tracing this problem back to the source code, ConfigX automatically detects that `Xbithack Full` implicitly overwrites code blocks enabled by `SSILastModified`. Such silent misconfigurations are subtle and can only be inferred by analyzing the non-trivial interactions between source code blocks that are decided by configurations. To correct this silent misconfiguration, `SSILastModified On` should be removed.

Pattern 5: Advanced ordering. The example shown in Section 1 presents a silent misconfiguration that results from advanced ordering.

2.4 Insights

Based on our observation of silent misconfigurations, we extract the following insights guiding our solution design. First, from our study more than 70% of studied silent misconfigurations are caused by intra-procedural semantic interaction of source code blocks; thus, in order to detect semantics-related silent misconfigurations, analyzing the overlap between different source code blocks is quite necessary. Second, since around 20% of silent misconfigurations are caused by mishandling guards in configuration files only, to detect these syntax-related silent misconfigurations, our syntax analyzer should detect them. Finally, because silent misconfigurations produce no useful system messages or logs, a potential solution to the problem should not rely on any system feedback.

3 DEFINING SILENT MISCONFIGURATIONS

In Section 1 we intuitively defined silent misconfigurations as errors that happen when the user sets a configuration parameter to some value, but this change does not have any effect. As our survey indicates, these types of errors appear often in practice. In this section we provide a formal definition for silent misconfigurations.

We first define an abstraction of a user configuration file. Given some configuration file C, we translate it into a flat intermediate representation, denoted by a list $\hat{C}$. The translation is described in Section 4.1. Having such an intermediate representation makes our approach system-agnostic: one only needs a parser from a configuration file to $\hat{C}$. All our definitions and techniques are defined on $\hat{C}$. Given a list L and an element e_1 and an element e_2, with $\texttt{before}(e_1, e_2, L)$ we denote that $e_1 \in L$ and $e_2 \in L$, and e_1 appears in L before e_2. To simplify the notation, we assume that the list contains no repeated elements.

The list $\hat{C}$ contains two types of elements, either a tuple of the form (g_i, c_i, v_i), or a closure element $\texttt{cl}(g_i)$. In the tuple, c_i is the name of a configuration parameter and v_i is its value. The value v_i is either set by the user, in which case we annotate it as v_i^u, or it is the default value preset by the system, annotated as v_i^d. The "MySQL 8.0 user manual states explicitly that “Without an option file, the server just starts with its default settings.” If a configuration parameter does not have a preset value, we set v_i^d to “Null.” The first argument, g_i, is a guard ensuring the configuration parameter c_i is set to the value v_i. A typical example in a configuration file that will result in a guard is the command `IfModule ...`. In our representation the guard is either `True`, *i.e.*, there is no guard, or it is a conjunction of atoms of the form $c_j = v_j$. An example of a guard in our representation is $\texttt{module1=loaded} \wedge \texttt{engine2=On}$.

The closure element, $\texttt{cl}(g_i)$, closes the scope of a guard. In a well-formed configuration file, for every guard there is an explicit statement or an indentation indicating where the guard ceases to hold. We introduce a predicate `closed` to describe that a guard g_i is closed in the list $\hat{C}$:

$$\texttt{closed}(g_i, \hat{C}) \Leftrightarrow \begin{cases} g_i = \texttt{True} \\ g_i = (c_j = v_j) \wedge \texttt{before}((g_i, c_i, v_i), \texttt{cl}(g_i), \hat{C}) \\ g_i = \bigwedge_{j \geq 1}^{k} g_j \wedge \forall j.1 \leq j \leq k.\texttt{closed}(g_j, \hat{C}) \wedge \texttt{well-formed}(g_i\ , \hat{C}) \end{cases}$$

The guard g_i is closed if it is `True`, or if it is of the form $c_j = v_j$ and its closing element $\texttt{cl}(g_i)$ appears later in the list $\hat{C}$, or if g_i is a conjuction of guards, each guard of the conjuction needs to be

Table 3. The definition of silent misconfigurations. If there exists a tuple (g_i, c_i, v_i) in $\hat{C}$ such that (g_i, c_i, v_i) satisfies one of the preconditions, then setting $c_i = v_i$ is a silent misconfiguration, and its particular name is given in the first column.

Name of an Error	**Preconditions**
Unmatched Guards	$[\![g_i]\!] \Downarrow \mathit{Open}$
Missing Module	$[\![g_i]\!] \Downarrow \mathit{False}$
Miss Handling Default	$[\![g_i]\!] \Downarrow \mathit{True}$, $[\![g_j]\!] \Downarrow \mathit{True}$, (g_i, c_i, v_i) in $\hat{C}$, (g_j, c_j, v_j) in $\hat{C}$, $\exists r \in \mathcal{R}$, s.t., $r = \texttt{Disables}(c_j = v_j^d, c_i)$, v_j^d is a default value for c_j
Implicit Overwrite	$[\![g_i]\!] \Downarrow \mathit{True}$, $[\![g_j]\!] \Downarrow \mathit{True}$, (g_i, c_i, v_i) in $\hat{C}$, (g_j, c_j, v_j) in $\hat{C}$, $\exists r \in \mathcal{R}$, s.t., $r = \texttt{OverWrites}(c_j = v_j, c_i = v_i)$
Advanced Ordering	$[\![g_i]\!] \Downarrow \mathit{True}$, $[\![g_j]\!] \Downarrow \mathit{True}$, $[\![g_k]\!] \Downarrow \mathit{True}$, (g_i, c_i, v_i) in $\hat{C}$, (g_j, c_j, v_j) in $\hat{C}$, (g_k, c_k, v_k) in $\hat{C}$, $\exists r \in \mathcal{R}$, s.t., $r = \texttt{AdvOrder}(c_j = v_j, c_k, c_i)$

closed. Additionally, the guard that is a conjuction has to appear in the list after all its conjucts and before their closing elements. We denote this property with $\texttt{well-formed}(g_i, \hat{C})$, for $g_i = \bigwedge_{j \geq 1}^{k} g_j$:

$$\texttt{well-formed}(g_i\ , \hat{C}) \Leftrightarrow \forall j. 1 \leq j \leq k.\ \texttt{before}((g_j, c_j, v_j), (g_i, c_i, v_i), \hat{C}) \\ \wedge\ \texttt{before}((g_i, c_i, v_i), \texttt{cl}(g_j), \hat{C})$$

The other issue with guards is that they might not be available. For example, a guard might state that a module M needs to be loaded for configuration parameters to be set to some values, but if the module M was never loaded then this part of the configuration file is irrelevant. In the terms of the $\hat{C}$ list, this means that whenever a new non-trivial guard is introduced, there must be an element in the list that ensures that this guard is enabled. To capture this, we introduce a predicate

$$\texttt{avail}(g_i, \hat{C}) \Leftrightarrow \begin{cases} g_i = \texttt{True} \\ g_i = (c_j = v_j) \wedge \texttt{before}((g_j, c_j, v_j), (g_i, c_i, v_i), \hat{C}) \\ g_i = \bigwedge_{j \geq 1}^{k} g_j \wedge \forall j. \texttt{avail}(g_j, \hat{C}) \end{cases}$$

We can now define the evaluation of a guard g_j in the list $\hat{C}$. Informally, the guard evaluates to true if it is available and closed. Formally, a guard g can evaluate to one of three values: True, False, or Open. We denote the evaluation of a guard g in the list $\hat{C}$ with $[\![g]\!] \Downarrow_{\hat{C}} \cdot$. When it is obvious from the context, we omit $\hat{C}$:

$$[\![g]\!] \Downarrow_{\hat{C}} \text{True} \Leftrightarrow \texttt{closed}(g, \hat{C}) \wedge \texttt{avail}(g, \hat{C})$$
$$[\![g]\!] \Downarrow_{\hat{C}} \text{False} \Leftrightarrow \neg\texttt{avail}(g, \hat{C})$$
$$[\![g]\!] \Downarrow_{\hat{C}} \text{Open} \Leftrightarrow \neg\texttt{closed}(g, \hat{C}) \wedge \texttt{avail}(g, \hat{C})$$

To be able to define silent misconfigurations, we also need to consider a set of rules, or a specification, and see which rules are violated. We denote the specification by $\mathcal{R}$.

In our particular case, this specification is a set of rules that we learned by using the static analysis on the configuration-related system source code. We learn three types of rules:

(1) Miss handling default: A general form of the rule that we learn is $\texttt{Disables}(c_j = v_j, c_i)$. The meaning of this rule is that when the parameter c_j is set to the value v_j this disables the parameter c_i, making it inaccessible, independently of the value that the user assigns to c_i. Although a simplified form of this rule can be inferred by some existing work [Xu et al. 2013], none of the existing work handles the case of default values. When $v_j = v_j^d$, the user needs to explicitly set a customized value to c_j, otherwise c_i remains disabled.

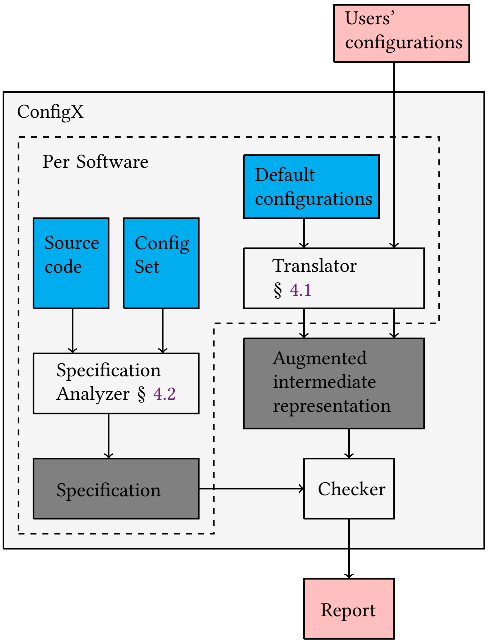

Fig. 1. ConfigX overview. The pink blocks are the inputs that ConfigX receives from users and the outputs that ConfigX provides for them. The blue blocks are the software-specific knowledge base. The gray blocks are intermediate results generated in ConfigX.

(2) Implicit overwrite: The rule $\texttt{OverWrites}(c_j = v_j, c_i = v_i)$ is the result of implicit connections between configuration parameters c_j and c_i, derived by a static analysis on the source code. The rule states that when the parameter c_j is set to the value v_j and the parameter c_i is set to the value v_i, the latter has no effect.

(3) Advanced ordering: The rule $\texttt{AdvOrder}(c_j = v_j, c_k, c_i)$ is another result of implicit connections between configuration parameters. The rule states that when the parameter c_j is set to the value v_j the parameter c_k gets a higher ordering than the parameter c_i. In particular, that means that if both parameters c_k and c_i are set to the same value, then the parameter c_k has a higher precedence and disables the parameter c_i, thus causing a silent misconfiguration. If the parameters are set to two different values then there is no discrepancy and the program behaves as expected.

Finally, given a configuration file $\hat{C}$ and a specification $\mathcal{R}$, we can formally define silent misconfigurations as follows: if there exists a tuple $(g_i, c_i, v_i) \in \hat{C}$ such that (g_i, c_i, v_i) satisfies one of the five preconditions from Table 3, then setting the value v_i to the parameter c_i is a silent misconfiguration.

4 CONFIGX DESIGN

Driven by the insights from Section 2.4, we design and build ConfigX, a system that can detect misconfigurations. Figure 1 gives an overview of ConfigX's architecture. It consists of three main modules. (1) The translator, described in Section 4.1, takes the user's configurations as input, augments them with the system defaults and generates the intermediate representation (IR) used in ConfigX. (2) The analyzer, described in Section 4.2, uses a customized program analysis to derive a specification describing connections between configuration's parameters. (3) The checker, detects possible silent misconfigurations by checking whether the given configuration file adheres to the derived specification.

4.1 Translator

The translator parses the user's configuration into an intermediate representation for post semantics-related silent misconfigurations checking. It also detects syntax-related silent misconfigurations.

Given a configuration file C, before we even start to parse it into an intermediate representation, we first augment that file with missing default values. This pre-processing works in two steps. First, we collect all of the system preset defaults from a user manual. Second, in the configuration file C we identify configuration parameters that are not customized by the user. We then augment the configuration file by explicitly adding the system defaults for those parameters.

After preprocessing the file, we invoke a parser that converts the configuration file C into a list $\hat{C}$, containing either tuples of the form (g_i, c_i, v_i) or closure statements of the form $\texttt{cl}(g_i)$.

Finally, after the configuration file has been translated, we immediately check for some silent misconfigurations, namely unmatched guards and missing modules.

Example 4.1. Consider the following generic configuration file:

```
module1 = loaded
module2 = loaded
[guard1: module0 = loaded]
    config1 = value1
[close guard1]
[guard2: module2 = loaded]
    config2 = value2
    [guard3: module1 = loaded]
        config3 = value3
[close guard2]
config4 = value4 (default)
```

We parse this file into the following list:

```
[(True, module1, loaded_u),
(True, module2, loaded_u),
(g1: module0 = loaded, config1, value1_u),
cl(g1),
(g2: module2 = loaded, config2, value2_u),
(g2 && g3: module1 = loaded, config3, value3_u),
cl(g2),
(True, config4, value4_d)]
```

Note that the values of configuration parameters have an annotation indicating if they are user defined values (the annotation `u`) or default values (the annotation `d`). We also introduce `g1` and `g2` as shorthands for the guards, while in the internal representation we just use guards as conjunctions.

Already in this first phase we detect silent misconfigurations of unmatched guards and missing modules. For example, $\texttt{avail}(g_1, \hat{C})$ evaluates to false, since `module0 = loaded` is missing in the first three lines, when we invoke it as a guard. This means that $[\![g_1]\!] \Downarrow_{\hat{C}}$ False and the preconditions for the missing modules silent configuration are fulfilled, so we raise the alarm.

4.2 Specification Analyzer

A specification analyzer derives the specification of configurations using a customized static analysis. The analysis takes the system source code and a set containing configuration default values as

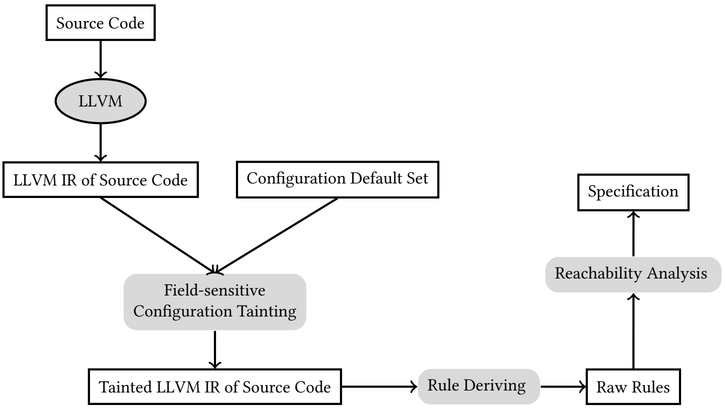
Source Code
LLVM
LLVM IR of Source Code
Configuration Default Set
Field-sensitive Configuration Tainting
Tainted LLVM IR of Source Code
Rule Deriving
Raw Rules
Reachability Analysis
Specification

Fig. 2. Workflow of Specification Analyzer in ConfigX

input and outputs the learned interactions between configuration parameters. An overview of our approach is given in Figure 2 and the pseudocode of our approach is in Algorithm 1. The specification analyzer consists of three different parts. The first part is a field-sensitive configuration tainting, where we use the LLVM compiler [Lattner and Adve 2004] and map the configuration parameters in the user's configuration files to registers in LLVM IR. In the second part, we derive the rules that show interactions between configuration parameters. The third part is a reachability analysis, which eliminates spurious rules systematically.

Algorithm 1 Specification Analysis

```
Input: $sC$ : System C Source Code
Input: $configSet$ : System Configuration Set
Output: $rules$ : A set of detected rules, each rule means a interaction between configurations
procedure SPECANALYZER($sC, configSet$)
    $llvmsC \leftarrow$ clang($sC$)
    for $c_i$ in $configSet$ do
        $mark\_var\_C \leftarrow$ map($c_i, llvmsC$)
        $taint\_regs\_llvm \leftarrow$ taint($mark\_var\_C, debug$)
        $taintedSet$.append($c_i, taint\_regs\_llvm$)
    for $func$ in $llvmsC$ do
        for $reg_i, reg_j$ in $taint\_regs\_llvm$ do                    ▷ configuraton $c_i$ is tainted to $reg_i$
            $att\_block\_reg_i, asso\_block\_reg_i \leftarrow$ analyze($func[reg_i/v_i]$)
            $att\_block\_reg_j, asso\_block\_reg_j \leftarrow$ analyze($func[reg_j/v_j]$)
            if ($asso\_block\_reg_j$ overwrites $asso\_block\_reg_i$) $\wedge$ reachabilityAnalysis($func, reg_i, reg_j$) then
                $spec\_set$.append($reg_i, v_i, reg_j, v_j$)
            if ($att\_block\_reg_i \subseteq asso\_block\_reg_j$) $\wedge\ v_j = v_j^d$ then        ▷ $v_j^d$ is the default value of $c_j$
                $spec\_set$.append($reg_i, v_i, reg_j, v_j$)
    return $spec\_set$
procedure REACHABILITYANALYSIS($func, reg_i, reg_j$)
    if (reach($reg_i$) $\wedge$ reach($reg_j$)) = unSAT then
        return False
    return True
```

Field-sensitive configuration tainting. The first step in deriving the specification of configurations from the software source code automatically is to determine which parts of the software source code are controlled by each configuration parameter. As shown in Figure 2, we first invoke the LLVM compiler and generate a configuration tainted LLVM IR. To achieve this, our field-sensitive forward configuration tainting performs two operations: (1) configuration mapping and (2) variable tainting.

Configuration mapping automatically maps the configuration parameters specified by the user to variables/functions in C program source code. We mark variables/functions configuration-related if they directly take the string representation of configuration parameters as inputs (Line 4 in Algorithm 1). For example, in Listing 1, configuration parameter `RewriteEngine` is mapped to the function `cmd_rewriteengine` in the Apache source code.

```
static const command_rec command_table[] = {
    AP_INIT_FLAG("RewriteEngine", cmd_rewriteengine, ...),
    ...
}
static const char *cmd_rewriteengine(cmd_parms *cmd, ..., int flag) {
    if (cmd->path == NULL) {
        sconf->state = (flag ? ENGINE_ENABLED : ENGINE_DISABLED);
    }
    ...
}
```

Listing 1. An example of field-sensitive configuration tainting in ConfigX. In Apache, the configuration parameter `RewriteEngine` is first mapped to the function `cmd_rewriteengine` and then mapped to the system variable `sconf->state`. The configuration mapping processes for system vsftpd and PostgreSQL are manifested in the same vein.

After mapping all configuration parameters to the initial functions in the software source code, ConfigX then performs variable tainting – a forward dataflow analysis to find which registers in LLVM are initialized by configurations. In Listing 3, the value of `RewriteEngine` taints the variable `sconf->state`. ConfigX maps the variables in C source code to tainted registers in LLVM by leveraging LLVM debug information (Line 5 in Algorithm 1). By the end, we generate a tainted LLVM IR.

Rule deriving. Rule deriving module reads tainted LLVM IR (generated above), and derives interaction rules between configuration parameters that may lead to silent misconfigurations. The main challenge is how to automatically infer complex correlations between configuration variables. ConfigX addresses this challenge by deeply analyzing the interaction of associated code blocks controlled by configurations.

ConfigX first needs to identify where the configuration parameters are loaded in the LLVM IR. We traverse the generated LLVM IR and then identify which basic block contains the first load of a configuration-related register, *i.e.*, attaching basic block B_i to reg_i. For example, in Figure 3, we attach basic blocks B_1, B_3, and B_5 to tainted registers reg_1, reg_2 and reg_3, respectively. Formally, let $\llbracket P[reg_i] \rrbracket$ denote the attached basic block that contains the first load of reg_i in this LLVM function. In Figure 3, we have $\llbracket P[reg_1] \rrbracket = \{B_1\}$. Similarly, we have $\llbracket P[reg_2] \rrbracket = \{B_3\}$ and $\llbracket P[reg_3] \rrbracket = \{B_5\}$.

After attaching basic blocks to the corresponding registers, ConfigX infers which basic blocks are controlled by the value of configuration parameters. This design intuitively pinpoints the code functionality enabled by setting a given configuration parameter. If a basic block is exclusively executed by setting the specific value to a configuration parameter and this block is not decided by the system OS call or other configuration, this basic block is associated with that configuration parameter (Line 9-10 in Algorithm 1). Let $\llbracket F[reg_i = v_i] \rrbracket$ denote the list of associated blocks exclusively executed when reg_i is set to v_i.

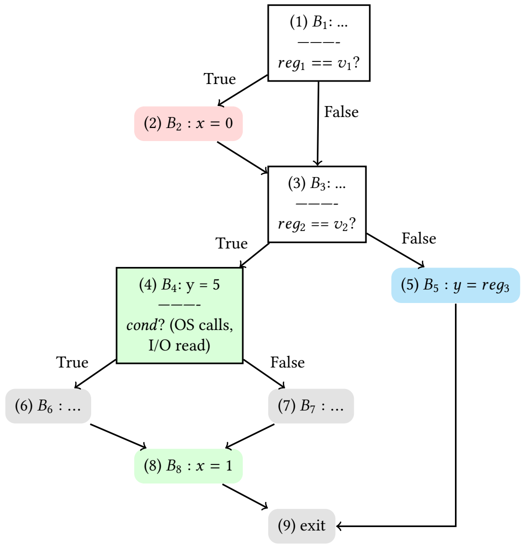

Fig. 3. An example of how ConfigX derives interactions between configurations c_1, c_2 and c_3. Each node in this figure represents an LLVM basic block. x and y are two system inner variables that are not initialized by configurations. Here c_1, c_2, and c_3 taint reg_1, reg_2 and reg_3 respectively in LLVM IR. The red node represents the LLVM basic block associated with the configuration parameter $c_1 = v_1$. The green nodes represent the basic blocks associated with the configuration parameter $c_2 = v_2$. The blue node represents the basic block associated with the configuration parameter $c_2 \neq v_2$. ConfigX derives two rules from this example. First, a miss handling default rule: $\texttt{Disables}(c_2 = v_2, c_3)$, if $v_2 = v_2^d$ (v_2^d is the default value of c_2). Second, an implicit overwrite rule: $\texttt{OverWrites}(c_2 = v_2, c_1 = v_1)$.

We use two examples to illustrate how we infer which basic blocks are controlled by the value of configuration parameters. As shown in Figure 3, since basic block B_2 is executed by setting configuration reg_1 to v_1, we have $\llbracket F[reg_1 = v_1] \rrbracket = \{B_2\} : \{x = 0\}$. Note that $\{B_3\}$ is not associated with $\llbracket F[reg_1 = v_1] \rrbracket$ because setting $reg_1 \neq v_1$ will also execute basic block $\{B_3\}$, which violates the exclusivity. For another example also in Figure 3, since the exit condition in basic block B_4, $cond$, is controlled by the system environment such as file reading, permission checking by the OS and so on, we cannot decide which block controls B_6 and B_7 by looking at the value of the configuration c_2 alone. Therefore, to be conservative, we have $\llbracket F[reg_2 = v_2] \rrbracket = \{B_4, B_8\} : \{y = 5, x = 1\}$. Similarly, we have $\llbracket F[reg_2 \neq v_2] \rrbracket = \{B_5\} : \{y = reg_3\}$.

How does ConfigX handle miss handling default? ConfigX detects a miss handling default misconfiguration (Line 13-14 in Algorithm 1) between two configurations if the attached basic blocks of c_i are the subset of the associated blocks of $reg_j \neq v_j^d$. For example, in Figure 3, since $\llbracket P[reg_3] \rrbracket \subseteq \llbracket F[reg_2 \neq v_2^d] \rrbracket$, ConfigX detects a miss handling default rule: $\texttt{Disables}(c_2 = v_2, c_3)$.

How does ConfigX handle implicit overwrite? ConfigX detects an implicit overwrite misconfiguration between c_i and c_j (Line 11-12 in Algorithm 1) when the variables and functions in the associated blocks exclusively executed by setting $c_i = v_i$ are overwritten by the setting $c_j = v_j$. For example, in Figure 3, since $\llbracket F[reg_2 = v_2] \rrbracket : \{y = 5, x = 1\}$ overwrites $\llbracket F[reg_1 = v_1] \rrbracket : \{x = 0\}$, ConfigX detects a implicit overwrite rule: $\texttt{OverWrites}(c_2 = v_2, c_1 = v_1)$.

Reachability analysis. The goal of reachability analysis is to check the feasibility of derived rules in the above rule derivation process. ConfigX detects a spurious rule if the configuration-related registers that are involved in this rule could not be reached together under any conditions.

For every rule derived, ConfigX computes the reachability condition of every configuration-related register in this rule. Then, ConfigX checks the conjunction of all the reachability conditions by invoking the SMT solver z3 [de Moura and Bjørner 2008]. If this formula is unsatisfiable, the derived rule is spurious. The pseudocode for reachability analysis is shown in Lines 16-19 in Algorithm 1.

Computing the exact reachability condition requires exploring all possible traces from the start of the program, which does not scale to a large program such as Apache or PostgreSQL. To address this challenge, ConfigX computes the approximate reachability condition by limiting the computation to a single function. We preserve soundness; every spurious rule filtered out in ConfigX is guaranteed to be a false positive. Compared to the existing tools that also derive rules via source code analysis [Nadi et al. 2015; Xu et al. 2013], reachability analysis is first introduced in ConfigX to systematically reduce false positives rather than adopting statistical methods.

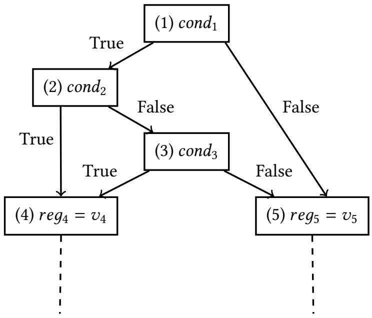

Fig. 4. An Example of False Positive Pruning via Reachability Analysis in ConfigX

For example, suppose ConfigX has derived a raw rule that involves two configuration-related registers reg_4 and reg_5. In Figure 4, the reachability condition of reg_4, $Reach(reg_4)$ is $(cond_1 = True) \wedge ((cond_2 = True) \vee (cond_3 = True))$. Similarly, we have $Reach(reg_5) = (cond_1 = False) \vee ((cond_2 = False) \wedge (cond_3 = False))$. Since $(Reach(reg_4) \wedge Reach(reg_5))$ is evaluated unsatisfiable in z3, ConfigX detects this as a spurious rule. Reachability analysis plays an essential role in our rule deriving process, and we report the effectiveness and efficiency of reachability analysis in Section 5.

5 EVALUATION

We aim to evaluate ConfigX by answering the following questions: (1) How effective and efficient is ConfigX in deriving specification of configurations? (2) How many silent misconfigurations have been detected by ConfigX in Apache, vsftpd and PostgreSQL in total? (3) How efficient is ConfigX?

Why are state-of-the-art tools not helpful? Before answering the above questions, we compare ConfigX with the state of the art. In general, the representative misconfiguration prevention systems cannot detect silent misconfigurations. Table 4 shows a concrete comparison between ConfigX and four state-of-the-art efforts (PCheck [Xu et al. 2016a], Spex [Xu et al. 2013], cDep [Chen et al. 2020], EnCore [Zhang et al. 2014], and ConfigV [Santolucito et al. 2017]).

First, PCheck [Xu et al. 2016a] relies on accurate error messages to detect potential single-parameter misconfigurations; thus, PCheck is not able to detect silent misconfigurations. Second, Spex [Xu et al. 2013] and cDep [Chen et al. 2020] analyze and capture dependencies between configurations. However, the detection of semantics-related silent misconfigurations requires an understanding of the complex interactions between configurations and source code. Unlike

Table 4. All Evaluated Systems and Their Comparisons

	Syntax-related Misconfiguration		Semantics-related Misconfiguration		
Type / Tool	**Unmatched Guards**	**Missing Module**	**Miss handling Default**	**Implicit Overwrite**	**Advanced Ordering**
PCheck [Xu et al. 2016a]	✓	✓	X	X	X
Spex [Xu et al. 2013]	X	X	X	X	X
cDep [Chen et al. 2020]	X	X	X	X	X
Encore [Zhang et al. 2014]	X	X	X	X	X
ConfigV [Santolucito et al. 2017]	✓	✓	X	X	X

ConfigX, Spex and cDep are limited to the analysis of the configuration values themselves and do not further analyze the system source code affected by the configuration values. Consequently, they cannot deal with the three patterns of semantics-related silent misconfigurations that we have identified. Third, EnCore [Zhang et al. 2014] and ConfigV [Santolucito et al. 2017] take fundamentally different approaches and rely on different assumptions. They collect massive configuration files for training rather than analyzing the source code, so that they cannot detect semantics-related misconfigurations because they fail to analyze complex semantics interactions between variables in the source code.

5.1 Implementation and Experimental Setup

We have built a ConfigX prototype using a mix of Python, LLVM and open-source software libraries. Our translator uses Python libraries to parse user configuration files into our intermediation representation. Our reachability analysis invokes the SMT solver z3[de Moura and Bjørner 2008] for false positive elimination. All the experiments in this section are conducted on a MacBook Pro equipped with Haswell Quad Core i7-4870HQ 2.5 GHz CPU, 16GB memory, and PCIe-based 512 GB SSD harddrive.

To evaluate the misconfiguration detection capability of ConfigX, we have collected five real configuration datasets (one Stack Overflow dataset, one benchmark Apache dataset, and three Github datasets for Apache, vsftpd, and PostgreSQL).

Stack overflow dataset setup. To check whether ConfigX can detect silent misconfigurations even when user's configuration files are incomplete, we build the first dataset (Stack Overflow dataset) from questions posted by Apache users on Stack Overflow. Specifically, we extract the configuration code snippets posted by the users who asked the questions and put them into our dataset. These posts typically contain $O(10)$ lines of configurations and a typical Apache configuration file consists of $O(100)$ lines of configuration. Having only incomplete configurations adds a significant challenge to misconfiguration checking tasks. This dataset is meant to be representative of the usage of ConfigX by users who want to check suspicious configuration snippets.

Existing benchmark dataset setup. To check whether ConfigX could detect silent misconfigurations in existing real-world public datasets, we run ConfigX on an Apache configuration dataset publicly available at [Xu 2017].

Github dataset setup. We further collect user's complete configurations across three widely used systems, Apache, vsftpd, PostgreSQL from the online code hosting platform GitHub, with roughly 100 configuration files from each system. These configuration files are contained in user's repositories and used by users in their system deployment. This dataset is representative of the usage of ConfigX by users who run ConfigX to verify the correctness of their configurations before they run the system.

Semantics: Miss Handling Defaults (VSFTPD)

```
if(tunable_data_connection_timeout  > 0)
   {vsf_sysutil_set_alarm(tunable_data_connectio
n_timeout);}
else if (tunable_idle_session_timeout  > 0)
   {vsf_sysutil_clear_alarm();}
```

Misconfiguration: idle_session_timeout = 600 (silent)
data_connection_timeout = 300 (System default, and augmented by ConfigX)

Description: The default value of "data_connection_timeout" is 300. Then "idle_session_timeout" has no effect unless user sets "data_connection_timeout" to be less than or equal to 0.

Impact: The function "vsf_sysutil_clear _alarm()" is not executed, so the alarm for the function "sysutil" will not be freed as the user intended.

Semantics: Implicit Overwrite (VSFTPD)

```
unsigned int caps = 0;
if (tunable_chown_uploads  )
  {caps |= kCapabilityCAP_CHOWN;}
if (tunable_connect_from_port_20  )
  {caps |= kCapabilityCAP_NET_BIND_SERVICE;}
vsf_secutil_change_credentials(...,caps,...);
```

Misconfiguration: chown_uploads = YES (silent)
connect_from_port_20 = YES

Description: Enabling "connect_from_port_20" will implicitly overwrite the system internal variable "caps" that is previously initialized by "chown_uploads".

Impact: Program enters a problematic state different than user's intention when "chown_uploads" is turned on.

Semantics: Miss Handling Defaults (PostgreSQL)

```
if (wal_level != configFile-> wal_level...)
   Xlrec.track_commit_timestamp  =
track_commit_timestamp;
```

Misconfiguration: wal_level = 'replica'
track_commit_timestamp = 'off' (silent)

Description: The preset default value of "wal_level" (left to !=) is already "replica". To enable the usage of "track_commit_time--stamp", the "wal_level" defined in the user's configuration must be set to some value other than the default value "replica".

Impact: The usage of "track_commit_time stamp" was disabled, contrary to what the user intended.

Semantics: Implicit Overwrite (PostgreSQL)

```
if (... && configFile. wal_level  == MINIMAL)
    error state
if (... && EnableHotStandby  )
    if (configFile. wal_level  < REPLICA):
        error state
    ...
```

Misconfiguration: wal_level = 'minimal'
hot_standby = 'off' (silent)

Description: If "wal_level" is set to "minimal" (which is also less than "replica" in PostgreSQL), then the program will enter an error state anyway. Any value that user sets for "hot_standby" does not matter here; it will not change any program behavior.

Impact: PostgreSQL entered into an error state.

Semantics: Advanced Ordering (Apache)

```
int ret = OK; // default
if (a->order[method]==ALLOW_THEN_DENY) {
    ret = HTTP_FORBIDDEN;
    if (find_allowdeny(r,a-> allows,method))
        ret = OK;
    if (find_allowdeny(r,a-> denys,method))
        ret = HTTP_FORBIDDEN;}
else if (a->order[method]== DENY_THEN_ALLOW)
{   if (find_allowdeny(r,a-> denys,method))
        ret = HTTP_FORBIDDEN;
    if (find_allowdeny(r,a-> allows,method))
        ret = OK;}
else {   if (find_allowdeny(r,a-> allows,method)
     && !find_allowdeny(r,a-> denys,method))
         ret = OK;
    else
         ret = HTTP_FORBIDDEN;}
```

Misconfiguration: Order Deny,Allow (silent)
Allow from all
Deny from 192.168.30.1 (silent)

Description: Setting configuration "Order" to "Deny,Allow" makes Apache first filter and deny the IP address matched with the value defined in the "Deny", then grant the access back to any entry matched with the "Allow". Apache first denies the access to 192.168.30.1, but it then grants access back to IP address 192.168.30.1 because of the setting "Allow from all".

Impact: User wants to block the address 192.168.30.1. However, this three-line configuration snippet is equivalent to a single line of "Allow from all". In this case, any IP address is allowed. This contradicts the user's intention and creates a serious security issue.

Fig. 5. Typical rules and real-world silent misconfigurations detected by ConfigX. Configurations that marked with (silent) indicate the location of the silent misconfigurations.

Table 5. Rules Learned by ConfigX

Software Type	Apache	vsftpd	PostgreSQL
Miss Handling Default	11	98	1
Implicit Overwrite	6	26	8
Advanced Ordering	1	0	0

Table 6. Breakdown of Silent Misconfigurations Detected by ConfigX

Type of Error	Apache Stack Overflow	Apache Existing Benchmark	Apache Github	vsftpd Github	Postgres Github
Unmatched Guards	1	7	1	0	0
Missing Module	0	142	5	0	0
Miss Handling Default	9	18	58	900	0
Implicit Overwrite	0	0	0	105	22
Advanced Ordering	2	653	310	0	0

5.2 Evaluation Results: Detecting Real Misconfigurations

We present the breakdown of all the rules of configurations derived by ConfigX in Table 5. We had learned 151 rules, and we manually inspected them and confirmed their correctness. We apply these rules on our five datasets: one Stack Overflow dataset, one benchmark Apache dataset, and three Github datasets for Apache, vsftpd and PostgreSQL.

Overall, ConfigX detected 2233 silent misconfigurations in 457 configuration files across the five datasets. The 2233 detected errors violated some of the learned rules in Table 5. Our user study (Section 6) further confirms the usefulness and the validity of the detected misconfigurations by ConfigX. Table 6 details these misconfigurations. Figure 5 shows several representative examples of silent misconfigurations extracted from the results. The rest of Section 5.2 analyzes the examples shown in Figure 5 and extracts some insights.

Miss handling default. There are two main reasons for the existence of miss handling default misconfigurations. First, the software developers may on purpose by default turn off the functionality of the configuration parameter for the sake of security. Configuration `RewriteEngine` described in Section 2.3 is an example. However, this design is not ideal and confuses users. To prevent this type of silent misconfigurations, ConfigX can send users the message "To enable the functionality of `RewriteRule`, `RewriteEngine` has to be set to `On`."

Second, miss handling default misconfigurations could be triggered due to a potential system design deficiency. For the vsftpd example in Figure 5, the default settings make both if-statements evaluate to true. Instead of setting parameter `Idle_session_timeout` itself, the user has to set parameter `Data_connect_timeout` to enable the functionality of `Idle_session_timeout`. This is not intuitive and even misleading to users. Instead, a useful message "To enable the functionality of parameter `Idle_session_timeout`, parameter `Data_connect_timeout` has to be set less than or equal to `zero`." is sent by ConfigX.

Implicit overwrite. Implicit overwrite silent misconfigurations are often introduced by software updates, and a new configuration is introduced to replace the functionality of the old, legacy one. However, this legacy configuration may still be valid for the new version of software. For example, in Figure 5 configuration `wal_level` is a newer version of configuration `hot_standby`. However, `wal_level` overwrites `hot_standby` with extra functionalities. Hence, tightly coupled functionalities

Table 7. Effectiveness and efficiency of reachability analysis in Apache. The evaluation details for vsftpd and PostgreSQL are manifested in the same vein.

Module in Apache	**False Positives Eliminated**	**Running time (s) Reachability = Off**	**Running time (s) Reachability = On**
mod_include	0	3.14	48.03
mod_rewrite	17	22.03	282.87
mod_access_compat	16	0.60	2.14
mod_proxy_express	0	0.10	0.78
mod_filter	8	0.59	1.68
mod_authz_core	0	1.85	5.18

enabled by two configuration settings confuse the user to set both, resulting in implicit overwrite misconfiguration.

In addition, when implicit overwrite misconfiguration happens, the user manual is often not helpful. For example, in the example presented Section 2.3, setting `XbitHack` to `Full` overwrites any setting of `SSILastModified`. However, the Apache web server's user manual documents that "The SSILastModified directive takes precedence over XBitHack." This information does not help configuration users and actually misleads them. A similar issue arises in the PostgreSQL example in Figure 5 shares a similar issue.

Advanced ordering. For advanced ordering misconfigurations, the root cause is that the interaction among three configuration parameters is too complex for users to handle. This advanced ordering issue causes real damage to the system. It is listed as the number one cause of "HTTP forbidden 403 error" by an Apache expert in this technical report.[1] ConfigX has detected more than 900 cases in real-world examples. Because the number of cases is so large, we believe that system developers should not expect end users to handle configuration interactions of this level of complexity.

Summary. ConfigX is the first tool which not only analyzes the interactions between configurations but also detects actual misconfigurations in a user's file. As the first context-aware configuration specification analysis framework, ConfigX builds a bridge from end users to system developers. By using ConfigX as a checker to validate users' configurations before they deploy problematic configurations, we demonstrate ConfigX effectively detects thousands of real silent misconfigurations in Table 6. The source of silent misconfigurations in Figure 5 is available at https://doi.org/10.5281/zenodo.4697619.

5.3 False Positives

In misconfiguration detection, having a low false-positive rate is another important factor. To achieve a low false positive rate, ConfigX prunes out false positives early during the specification deriving process with reachability analysis (Section 4.2). We report the effectiveness and efficiency of our reachability analysis in Table 7. This table shows ConfigX effectively eliminates 34 false positives in total during the specification deriving process. Table 7 also shows that running a reachability analysis adds reasonable time overhead to the whole system.

Note ConfigX relies on LLVM debug info to map the configuration variables to registers in LLVM IR. If LLVM loses accuracy in the configuration-register mapping process, then it is possible for our tool to generate a false positive. We manually inspected the rules generated by ConfigX and we did not find any such case in practice.

[1] https://www.petefreitag.com/item/793.cfm

Table 8. ConfigX's Runtime Performance

Specification Deriving Time (s)	Apache	vsftpd	PostgreSQL
Reachability = Off	37.76	2.01	90.09
Reachability = On	389.12	577.78	3103.55

5.4 Efficiency

Apache, vsftpd and PostgreSQL are all widely used software. They all contain hundreds of thousands of lines of code in their original implementation. Table 8 shows ConfigX efficiently derives specification for configurations. The time is mainly spent in the offline specification analyzing phase, while the checking phase, the user-facing running time takes only negligible time.

6 EXPERIENCE

To understand the user perceptions on the silent misconfigurations detected by ConfigX, we further conduct a user study using practice described in [Santolucito et al. 2017]. We randomly selected 160 silent misconfigurations detected by ConfigX (Section 5.2). We then generate a report for each of the selected silent misconfiguration. The report includes the location of the silent misconfiguration in the configuration file, the consequence of the silent misconfiguration, and the corresponding code snippets that explain the silent misconfiguration. We also attached a patch for fixing the silent misconfiguration.

Results. So far, we have received responses of 19 of the reported silent misconfigurations and interacted with the owners of the configuration files. This yield rate is slightly higher than in prior studies [Santolucito et al. 2017]. We carefully reviewed each responses. Overall, the user study confirmed the usefulness of ConfigX: in 16 out of 19 cases, the users confirmed the silent misconfigurations, and, more importantly, they used our reports to fix them. In the remaining three cases, the users expressed appreciation, and explained why our solution does not help. Those reasons are: 1) They abandoned the project five years ago, and the repository is immediately archived after checking our report. 2) The main developer passed away. 3) They used a project-specific external plugin to overwrite the configurations before passing them to the system (which is out of the scope of ConfigX). None of the received response was negative.

We experienced the following two interaction patterns:

Pattern 1: Silent misconfiguration is confirmed and/or fixed by the users themselves. From the responses we received, most (12/16) users actively investigated our reported silent misconfigurations, and later fixed the issue with the help of our tool by themselves. They replied with the messages: "Thanks! Fixed.", "Excellent job! The source code is beautiful!" and "You're right. Thanks for pointing this out." We find the users' supportive attitude encouraging and believe that it confirms the importance of our tool.

Pattern 2: Silent misconfiguration is confirmed/fixed with extra steps. Due to the convoluted complexity of the detected silent misconfiguration or the fact that the misconfiguration has existed for a long time, users sometimes needed to take an extra step to confirm or to fix the misconfiguration. This happened in four out of 16 cases.

This extra step usually would go through the following steps: 1) The issue assignee asked other developers to double check and confirm the reported issue. 2) Other developers joined the discussion of the issue. As an illustration, one user filed a pull request to ask the other three repository developers to examine the patched version suggested by ConfigX. After thorough examination and discussion, the detected misconfiguration was confirmed and the pull request was approved. In a different case, a lead developer joined the discussion and confirmed that the reported issue is indeed a misconfiguration. The developer also confirmed that the issue is already fixed in the latest codebase as suggested in our report.

Note that in all of the 16 confirmed cases, the owner took quick actions to address the misconfiguration that we detected. In all of the cases, the whole misconfiguration find-fix-verify cycle was all closed in less than two days. This is in sharp contrast to the fact that sometimes silent misconfigurations existed in configuration files for years without users finding a way to fix them. In our study, the average lifetime of silent misconfiguration is more than two years, and the longest-lived one lingered in its repository for more than five years. This validates that silent misconfigurations are notoriously difficult for users to find and fix, while ConfigX can solve the problem in a short span of time.

Summary. Our user study confirms the usefulness and the validity of ConfigX's misconfiguration detection process. What is especially encouraging is that users took an active role in responding to the misconfigurations that we detected and demonstrated a strong willingness. The source of our user study is available at https://doi.org/10.5281/zenodo.5173050.

7 DISCUSSION

7.1 Users' Attitude Towards Detected Silent Misconfigurations

In the user study, most users (in 15 out of 16 cases), took an active role in troubleshooting the misconfigurations that we detected. Interestingly, in one case, the user first confirmed that the detected misconfiguration is a true positive, but he further explained that this silent misconfiguration was left in the configuration file on purpose as a record (flag) for the other contributors working on the same codebase. Additionally, this silent misconfiguration was used only for future development purpose.

This case showed that it is possible, although very unlikely, that users sometimes intentionally leave a silent misconfiguration in their codebase for a special purpose. Nevertheless, we proactively detect and report silent misconfigurations to users, and at a minimum they can decide how to deal with the issue based on their specific needs.

7.2 Known Sources of Unsoundness or Incompleteness

Overall, we did not find any false positives in the empirical evaluations. Our definition of silent misconfigurations heavily depends on the learned rules – this fact can be a possible source of unsoundness or incompleteness of ConfigX. There could be more system errors, which manifest behavior similar to silent misconfigurations. However, if we did not previously learn the rules flagging such behavior, ConfigX does not detect these misconfigurations because they are not even considered silent misconfigurations. To be recognized as a silent misconfiguration, we would need to learn a rule classifying problematic behavior and extend the definition. We believe that extending the definition of silent misconfigurations and learning new rules should be an ongoing process.

As the definition of silent misconfigurations is based on the learned set of rules, another source of unsoundness and incompleteness could be learning incorrect rules. If ConfigX learns an incorrect rule, but the user has a correct configuration, reporting this correct configuration as a misconfiguration is an example of unsoundness. If ConfigX learns an incorrect rule, and the user indeed has a misconfiguration, we might not be able to detect that misconfiguration. We illustrate this case in Example 7.1.

Therefore, the sources of unsoundness and incompleteness in ConfigX are incorrect rules learned in Algorithm 1. There are two stages in Algorithm 1 that could lead to incorrect rules: the initial mapping of configuration parameters to system variables, and during the static analysis. False positives might appear in the static analysis stage as the result of an overapproximation by LLVM, but then the reachability analysis refines it to delete spurious rules (as detailed in Sec. 4.2). We, therefore, believe that the main source of learning incorrect rules is when ConfigX cannot map correctly configuration parameters to system variables.

Example 7.1. This example demonstrates how learning an incorrect rule can cause an incompleteness of ConfigX. Consider the following configuration snippet, from the configuration for OpenLDAP, given in its user manual[2]:

```
1:    olcLogLevel 129
```

This configuration parameter is used for specifying the levels of debugging statements. According to the user manual, the configuration `olcLogLevel 129` is equivalent to:

```
1:    olcLogLevel 128 1
```

This is because the configuration parameter `olcLogLevel` takes as input an integer and considers it as a sum of powers of 2, in this particular case 129 = 128 + 1. Each summand is called a "level", and levels define where debugging statements are logged. If `olcLogLevel 129` appears in the configuration file, this will switch on two levels for debugging statements, 128 and 1. More specifically, `olcLogLevel 128` enables access control list processing, while `olcLogLevel 1` makes tracing function calls available. If the user does not want any debugging, the value should be set to `0`.

When `olcLogLevel 129` appears in the configuration file, ConfigX maps the value `129` to the variable `olcLogLevel`, while in fact `olcLogLevel` should be mapped to the values 128 and 1. As the result of this incorrect mapping, we learn an incorrect rule of the form: $\texttt{OverWrites(olcLogLevel} = 129, c_1 = v_1 \vee c_2 = v_2)$, when in fact we should learn the following two rules:

$$\texttt{OverWrites(olcLogLevel} = 128, c_1 = v_1)$$
$$\texttt{OverWrites(olcLogLevel} = 1, c_2 = v_2)$$

If the user, who wants to use ConfigX, explicitely specifies in the configuration file that `olcLogLevel = 128`, and $c_1 = v_1$, this is the case for the implicit overwrite silent misconfiguration (assuming that the corresponding guards are correct). As such, a silent misconfiguration should trigger ConfigX to raise a warning. However, our tool does not detect this silent misconfiguration, since it does not have a proper understanding of the semantics of the assignment `olcLogLevel = 129`.

To correctly parse the configuration `olcLogLevel 129`, ConfigX's static analyzer would first need to understand the system-specific logic in configuration parsing, which it does not support.

To address the exemplified issue, one would either need to codify application-specific knowledge, or explore advanced parsing techniques (e.g., using natural language processing techniques to infer such semantics from user manuals).

7.3 Limitations and Discussions

False positives may occur if an error happens during the configuration mapping process. For instance, if a system uses non-standard pointer arithmetic logic to load the configuration parameters, it is possible that ConfigX maps the configuration parameters to incorrect variables. This initial problem will propagate further and we might later learn incorrect rules between configurations. We have found that this happens with OpenLDAP.

Since ConfigX employs static analysis to detect misconfiguration, we treat the system source code and configuration statically. ConfigX is not able to detect misconfigurations that are generated after system initialization phase or are dependent on external system inputs.

[2] https://www.openldap.org/doc/admin24/slapdconf2.html

Internal validity. ConfigX maps each configuration parameter to program variables; thus, if the misconfiguration is caused by the incorrect order between two configuration parameters [Santolucito et al. 2017; Yin et al. 2011] in the user's configuration file, ConfigX cannot detect them. To our knowledge, this is a limitation of all program-analysis based configuration techniques [Wang et al. 2004; Xu et al. 2016a, 2013].

External validity. ConfigX assumes the availability of the source code and does not work on binary code. ConfigX as a tool currently only supports C programs. On the other hand, the idea is generically applicable to software programs written in other programming languages.

7.4 Larger Implications and Future Directions

We envision ConfigX in its current version as a silent misconfiguration checker that can be run before users import configurations into the system. Encouraged by our detected misconfigurations and confirmations from our users, there are two promising future directions to pursue.

Detecting misconfigurations that are dependent on users' runtime inputs. Some systems, such as MySQL, allow users to dynamically change the value of configurations during the system runtime. The control flows and the execution traces of the system will then be constantly changed and affected by the user's external inputs. Some typical misconfigurations that are dependent on the user's runtime inputs are importing the configurations in an incorrect order, missing the entry configuration parameters and so on. To detect these kinds of misconfigurations, instead of having the user's configuration files as the input, an image that encapsulates the whole list of user's configuring processes is a must. We plan to extend our analysis to support detecting these kinds of misconfigurations in the future.

Designing a high-level language to tackle silent misconfigurations at the source. Designing a configuration language to avoid silent misconfigurations is another exciting research direction. Having a high-level declarative language could help users remove redundancies in their configurations and reduce the size of their configuration files. We have already seen such languages designed successfully for modular router configurations [Morris et al. 1999] and cloud systems [Huang et al. 2015].

In fact, the results of ConfigX could be regarded as the foundations of a configuration language design. The syntax-related silent misconfigurations are directly detected by a language-level type checker. Also, a collection of our learned rules could be viewed and used as semantic constraints on the configurations used in the system.

However, designing such a high-level declarative language to avoid silent misconfigurations is challenging because:

1. The redundancies in a user's software configuration may not be as noticeable as they are in router configurations or in cloud systems. We have yet to determine the reduction percentage in terms of lines of software configuration.
2. It is difficult to properly translate user's high-level intentions to low-level configurations. For example, users sometimes intentionally left a silent misconfiguration in their codebase for a special developmental purpose.
3. When users need to change configurations due to undesired system behaviors, since the system outputs error messages of misconfigurations at a low level, it is difficult for users to change the high-level configuration descriptions to meet the requirements.
4. It adds extra complexity for users to learn a new language at first, start to write language-level configurations from scratch, and then maintain them. There is no straightforward way to reuse legacy configurations. All existing assistance sources (official user manuals, publicly available online discussion forums) target low-level configurations. Moreover, the

previous study shows that the abuse of language support might impair the usability of configuration [Mason 2011].

8 RELATED WORK

Silent misconfigurations have long been known as a severe and challenging problem, as reported in prior studies [Tang et al. 2021; Tian et al. 2019; Xu et al. 2016b; Xu and Zhou 2015; Ye et al. 2020; Yin et al. 2011; Zhai et al. 2020; Zhang and Ernst 2013; Zhang et al. 2021]. First, the silence leads to unexpected behavior which can hardly be observed from the system's perspective, because the runtime execution does not reveal anomalies. As a result, misconfiguration detection approaches (such as Ctest [Cheng et al. 2021; Sun et al. 2020; Xu and Legunsen 2020] and PCheck [Xu et al. 2016a]) that rely on runtime failure behavior and/or performance anomalies can hardly deal with silent misconfigurations. For the similar reason, silent misconfigurations are notoriously difficult to diagnose, as most diagnosis tools need to identify a crashing point or degraded performance metric as the starting points [Attariyan et al. 2012; Attariyan and Flinn 2010; Rabkin and Katz 2011b; Su et al. 2007; Wang et al. 2004, 2003; Yuan et al. 2006].

The challenges of detecting silent misconfigurations are rooted in reasoning about interactions among the configurations and the corresponding code affected by the configurations. Data-driven approaches, such as Encore [Zhang et al. 2014], ConfigC [Santolucito et al. 2016], ConfigV [Santolucito et al. 2017], PracExtractor [Xiang et al. 2020], and ConfSeer [Potharaju et al. 2015] can potentially infer coarse-grained dependencies among configuration parameters using data mining [Mehta et al. 2020; Santolucito et al. 2017, 2016; Zhang et al. 2014] or NLP techniques [Potharaju et al. 2015; Xiang et al. 2020]. However, none of them are able to analyze the interactions at the level of source code; therefore, they cannot capture deep interactions in the way that ConfigX can. As discussed in Section 2, silent misconfigurations are often rooted in configuration-related code that overwrites the effect of other configurations.

A few efforts analyzed the dependencies between different configuration parameters, including Spex [Xu et al. 2013] and cDep [Chen et al. 2020]. Unlike ConfigX, they are limited to the configuration values themselves and do not further analyze the code affected by the configuration values. Therefore, silent misconfiguration patterns such as miss handling default, implicit overwrite, and advanced ordering are out of their scope, because they do not understand deep interactions between code and configurations. More importantly, neither Spex nor cDep can detect silent misconfigurations—the former is designed for finding defects in the code, while the latter stops at analyzing the dependencies without use cases.

Configuration dependencies have also been studied in other contexts, such as compile-time feature flags for variability modeling [Franz et al. 2020; Kuo et al. 2020; Medeiros et al. 2020; Nadi et al. 2014; Nadi et al. 2015; Tartler et al. 2014], performance modeling [Herodotou et al. 2011; Hu et al. 2020; Jamshidi et al. 2017, 2018; Li et al. 2020; Nair et al. 2018; Van Aken et al. 2017], and security [Bauer et al. 2011; Bouchet et al. 2020; Das et al. 2010; Xiang et al. 2019]. Nadi *et al.* provides an empirical study of configuration constraints of compile-time features in kconfig (Linux kernel configuration) and develops tooling to extract the constraints [Nadi et al. 2015]. ConfigX focuses on runtime, user-facing configurations, instead of compile-time configurations (the differences are discussed in [Meinicke et al. 2020]). Techniques designed for compile-time feature flags cannot be directly applied to user-facing, runtime configurations—the former are mostly boolean values in the form of C preprocessors, while the latter have more complex types and transformations in the program. A performance model often takes configuration values as inputs [Herodotou et al. 2011; Hu et al. 2020; Jamshidi et al. 2017, 2018; Li et al. 2020; Nair et al. 2018; Van Aken et al. 2017]. LearnConf [Li et al. 2020] identifies the dependencies among configurations that affect system performance via static analysis. Violet [Hu et al. 2020] outputs a performance impact model for poor configuration values via symbolic execution. Security misconfigurations (e.g., access-control misconfigurations) could also be silent, in the sense that the resultant vulnerabilities are typically not revealed till security incidents (e.g., data breaches) happen. Some recent work studies specific configurations for access-control policies and rules [Bauer et al. 2011; Bouchet et al. 2020; Das et al. 2010; Xiang et al. 2019]. ConfigX focuses on functional properties instead of performance or security. Other studies verify and repair for a domain specific configuration language: Puppet [Fu et al. 2017; Shambaugh et al. 2016; Sharma et al. 2016; Weiss et al. 2017], while ConfigX has different scope on user-facing configurations.

ConfigX stands on the shoulders of state-of-the-art configuration analysis techniques based on static program analysis [Chen et al. 2020; Dong et al. 2015; Lillack et al. 2014, 2018; Rabkin and Katz 2011a,b; Xu et al. 2016a, 2013], including mapping configuration values to program variables, tracking propagation and transformation of configuration values, identifying code affected by configuration values, etc. Our contributions are: 1) a comprehensive analysis on the complex interaction of code blocks related to configurations, 2) a sound reachability analysis to systematically prune out the spurious rules, and 3) considering the default values in the analysis.

9 CONCLUSION

This paper presents ConfigX, a tool for detecting possible silent misconfigurations. By deriving the specification of configurations, ConfigX detects silent misconfigurations which the state of the art cannot detect. Inspired by the fact that silent misconfigurations are prohibitively expensive for users to troubleshoot, we build a ConfigX tool that proactively detects and reports silent misconfigurations.

ACKNOWLEDGMENTS

We thank OOPSLA reviewers for their insightful comments. We also thank Mark Santolucito and Julien Lepiller for their valuable feedback on the early version of this work. We also thank John Kolesar and Matt Elacqua for proofreading this work. Jialu Zhang is supported in part by NSF grants, CCF-1715387. Ruzica Piskac is supported in part by NSF grants, CCF-1715387, CCF-1553168 and CNS-1565208. Tianyin Xu is supported in part by NSF grants, SHF-1816615, CNS-1956007, CCF-2029049, and CNS-2130560, and a Facebook Distributed Systems Research award.

REFERENCES

2012. Silent misconfiguration: Advanced Ordering. https://stackoverflow.com/questions/9943042/htaccess-order-deny-allow-deny

2014. Microsoft service outage on November 20th, 2014. https://www.datacenterknowledge.com/archives/2014/11/20/microsoft-says-config-change-caused-azure-outage

2014. Silent misconfiguration: Implicit Overwrite. https://stackoverflow.com/questions/21338450/conditional-request-not-honored-in-includes

2017. Amazon service outage on February 28th, 2017. https://aws.amazon.com/message/41926

2018. Google service outage on July 17th, 2018. https://status.cloud.google.com/incident/cloud-networking/18012

2021a. Apache. https://httpd.apache.org

2021b. Apache User Manual. https://httpd.apache.org/docs/

2021. PostgreSQL. https://www.postgresql.org

2021. Server Fault. https://serverfault.com

2021. Stack Overflow. https://stackoverflow.com

2021. vsftpd. https://security.appspot.com/vsftpd.htmlg

Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA).

Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada).

Lujo Bauer, Scott Garriss, and Michael K. Reiter. 2011. Detecting and Resolving Policy Misconfigurations in Access-Control Systems. *ACM Transactions on Information and System Security (TISSEC)* 14, 1 (May 2011), 1–28.

Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 281–291. https://doi.org/10.1145/3368089.3409728

Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *In Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. Virtual Event.

Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*.

Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. 2010. Baaz: A System for Detecting Access Control Misconfigurations. In *Proceedings of the 19th USENIX Security Symposium*.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Zhen Dong, Artur Andrzejak, and Kun Shao. 2015. Practical and Accurate Pinpointing of Configuration Errors using Static Analysis. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. Bremen, Germany.

Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2020. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. *CoRR* abs/2012.15342 (2020). arXiv:2012.15342 https://arxiv.org/abs/2012.15342

Weili Fu, Roly Perera, Paul Anderson, and James Cheney. 2017. muPuppet: A Declarative Subset of the Puppet Configuration Language. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.12

Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Towards Automatic Optimization of MapReduce Programs.

Yigong Hu, Gongqi Huang, and Peng Huang. 2020. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 719–734. https://www.usenix.org/conference/osdi20/presentation/hu

Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: A systematic configuration validation framework for cloud services. In *10th European Conference on Computer Systems (EuroSys)* (Bordeaux, France).

Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: an exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 497–508. https://doi.org/10.1109/ASE.2017.8115661

Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 71–82. https://doi.org/10.1145/3236024.3236074

Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. In *Proceedings of the 2020 ACM SIGMETRICS Conference (SIGMETRICS'20)*.

C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically Inferring Performance Properties of Software Configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3342195.3387520

Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 445–456. https://doi.org/10.1145/2642937.2643001

Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering (TSE)* 44, 12 (Dec. 2018), 1269–1291.

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

Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Larissa Braz, Christian Kästner, Sven Apel, and Kleber Santos. 2020. An Empirical Study on Configuration-Related Code Weaknesses. In *SBES '20: 34th Brazilian Symposium on Software Engineering, Natal, Brazil, October 19-23, 2020*, Everton Cavalcante, Francisco Dantas, and Thaís Batista (Eds.). ACM, 193–202. https://doi.org/10.1145/3422392.3422409

Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, B. Ashok, Chetan Bansal, Chandra Maddila, Christian Bird, Sumit Asthana, and Aditya Kumar. 2020. Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*.

Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options. In *ICSE SEIP*.

Robert Tappan Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. 1999. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*, David Kotz and John Wilkes (Eds.). ACM, 217–231. https://doi.org/10.1145/319151.319166

Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining configuration constraints: static analyses and empirical results. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 140–151. https://doi.org/10.1145/2568225.2568283

S. Nadi, T. Berger, C. Kastner, and K. Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

Vivek Nair, Rahul Krishna, Tim Menzies, and Pooyan Jamshidi. 2018. Transfer Learning with Bellwethers to find Good Configurations. *CoRR* abs/1803.03900 (2018). arXiv:1803.03900 http://arxiv.org/abs/1803.03900

Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. 2015. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)*.

Ariel Rabkin and Randy Katz. 2011a. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*.

Ariel Rabkin and Randy H. Katz. 2011b. Precomputing possible configuration error diagnoses. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 193–202. https://doi.org/10.1109/ASE.2011.6100053

Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing configuration file specifications with association rule learning. *PACMPL* 1, OOPSLA (2017), 64:1–64:20. https://doi.org/10.1145/3133888

Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic Automated Language Learning for Configuration Files. In *28th Computer Aided Verification (CAV)* (Toronto, CAN).

Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: a configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 416–430. https://doi.org/10.1145/2908080.2908083

Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 189–200. https://doi.org/10.1145/2901739.2901761

Todd Spangler. 2019. Facebook Apologizes for Outages, Says It Has Resolved "Server Configuration" Error. https://variety.com/2019/digital/news/facebook-apologizes-outages-server-configuration-error-1203163429/#!/

StackOverflow #43239190. 2017. Apache mod rewrite rule not working. https://stackoverflow.com/questions/43239190

StackOverflow #6070335. 2011. Retain original request URL on mod_proxy redirect. https://stackoverflow.com/questions/6070335

Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving configuration management with operating systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)* (Stevenson, Washington).

Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. Virtual Event.

Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd D. Millstein, Yuval Tamir, and George Varghese. 2021. Campion: debugging router configuration differences. In *ACM SIGCOMM (SIGCOMM)*.

Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA.

Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and automatically updating in-network ACL configurations with Intent language. In *ACM SIGCOMM (SIGCOMM)*.

Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3035918.3064029

Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*.

Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. 2003. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*.

Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 625–636. https://doi.org/10.1109/ASE.2017.8115673

Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 265–280. https://www.usenix.org/conference/atc20/presentation/xiang

Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. 2019. Towards Continuous Access Control Validation and Forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*.

Tianyin Xu. 2017. Misconfiguration dataset. https://github.com/tianyin/configuration$_d$atasets.

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy).

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

Tianyin Xu and Owolabi Legunsen. 2020. Configuration Testing: Testing Configuration Values as Code and with Code. *CoRR* abs/1905.12195 (2020). arXiv:1905.12195 https://arxiv.org/abs/1905.12195

Tianyin Xu, Vineet Pandey, and Scott Klemmer. 2016b. An HCI View of Configuration Problems. *arXiv:1601.01747* (Jan. 2016).

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA).

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

Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *ACM SIGCOMM (SIGCOMM)*.

Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal).

Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys'06)*.

Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *USENIX Annual Technical Conference (USENIX ATC)* (Portland, OR).

Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, UT).

Sai Zhang and Michael D. Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*.

Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In *In Proceedings of the 43rd International Conference on Software Engineering (ICSE'21).*