Do Not Blame Users for Misconfigurations

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng,
Ding Yuan*, Yuanyuan Zhou, Shankar Pasupathy†
University of California, San Diego, *University of Toronto, †NetApp, Inc.

Abstract

Similar to software bugs, configuration errors are also one of the major causes of today's system failures. Many configuration issues manifest themselves in ways similar to software bugs such as crashes, hangs, silent failures. It leaves users clueless and forced to report to developers for technical support, wasting not only users' but also developers' precious time and effort. Unfortunately, unlike software bugs, many software developers take a much less active, responsible role in handling configuration errors because "they are users' faults."

This paper advocates the importance for software developers to take an active role in handling misconfigurations. It also makes a concrete first step towards this goal by providing tooling support to help developers improve their configuration design, and harden their systems against configuration errors. Specifically, we build a tool, called SPEX, to automatically infer configuration requirements (referred to as *constraints*) from software source code, and then use the inferred constraints to: (1) expose *misconfiguration vulnerabilities* (i.e., bad system reactions to configuration errors such as crashes, hangs, silent failures); and (2) detect certain types of error-prone configuration design and handling.

We evaluate SPEX with one commercial storage system and six open-source server applications. SPEX automatically infers a total of 3800 constraints for more than 2500 configuration parameters. Based on these constraints, SPEX further detects 743 various misconfiguration vulnerabilities and at least 112 error-prone constraints in the latest versions of the evaluated systems. To this day, 364 vulnerabilities and 80 inconsistent constraints have been confirmed or fixed by developers after we reported them. Our results have influenced the Squid Web proxy project to improve its configuration parsing library towards a more user-friendly design.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522727

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

General Terms: Reliability, Design

Keywords: Misconfiguration, Constraint, Inference, Testing, Vulnerability

1 Introduction

1.1 Motivation

Configuration errors are one of the major causes of today's system failures. For example, Barroso and Hölzle report that misconfigurations are the second major cause of service-level failures at one of Google's main services [6]. Similar findings are reported in other studies [12, 22, 24, 27]. Recently, several systems, including Microsoft Azure, Amazon EC2, and Facebook, experienced a number of misconfiguration-induced outages that affected millions of their customers [2, 13, 31].

In fact, misconfigurations affect not only end users but also support and software engineers, because they need to spend time and effort in troubleshooting and correcting them [8, 14]. A recent study [36] shows that configuration issues account for 27% of customer support cases in a major storage company. Regardless of the root causes (software bugs or misconfigurations), the system often misbehaves with similar symptoms (e.g., crashes, missing functionalities, incorrect results). This leaves users no choice but to report the problems to the technical support. When support engineers are misled by such ambiguous symptoms, the diagnosis can take an unnecessarily long time [36].

Recently, many research efforts have been conducted to address the misconfiguration problem including troubleshooting anomalies caused by configuration errors [1, 3, 4, 5, 25, 32, 33, 34, 37, 40], detecting certain types of misconfigurations [11, 35, 38], automating certain con-

figuration tasks [7,9,17], and some others [15,19,26,28, 30,36]. All these studies focus on parameter-related misconfigurations, as they account for the majority of users' configuration errors [36]. Similarly, this paper also focuses on parameter-related misconfigurations.

While the previous work has significantly improved the situation by providing the *last* level of defense, the fundamental problem of misconfigurations probably lies in the configuration design and the target system itself. Unfortunately, not much attention has been paid to these two, partially due to our (software developers') attitude towards misconfigurations (which is quite different from how we treat software bugs). For software bugs, developers typically take a responsible and active role. This is reflected in many ways, such as various choices of bug-tracking databases, patch releases, unit/regression tests, and bug checkers. In contrast, developers often take laid-back roles in handling misconfigurations because "they are users' faults." Such an attitude is reflected in two main aspects: (1) Misconfigurations are much less rigorously tracked; and (2) After a configuration error is identified as the root cause, developers often do not take any further action, such as changing the code or releasing patches in order to avoid the same misconfigurations by other users (which is often the case).

In most cases, even though it is the users who commit the configuration errors, they should not take all the blame. After all, a misconfiguration is referred to as an "error" simply because it does not match our (software developers') requirements for configuration. Therefore, before blaming users for configuration errors, we need to question whether we have the right requirements in the first place. For example, are we assuming too much from users? Users do not write our code and sometimes cannot read our code. How could they have the same level of understanding of the requirements and impact of various configuration settings as we do? Are our configuration requirements too strict or too confusing? After all, users are human beings, and just like us, also make mistakes, especially when the requirements are error-prone.

Moreover, while we are often trained and educated to implement our code to tolerate hardware and network errors, we place little emphasis on tolerating or reacting gracefully to users' configuration errors. In fact, just like hardware errors, human errors are a force of nature, too. Unfortunately, in reality, developers often unconsciously assume correct configurations. As a result, many configuration errors lead to system crashes, hangs, incorrect results, etc., leaving users clueless and forced to report to support engineers for assistance in failure diagnosis. On the other hand, if the software could pinpoint the configuration errors with explicit log messages, users could directly fix their mistakes by themselves without resorting to the technical support. *Different from software bugs, if accurate error messages are provided by the system, most configuration errors can be easily fixed by users themselves.* Therefore, providing good reactions to configuration errors can significantly reduce the number of issues reported to support engineers.

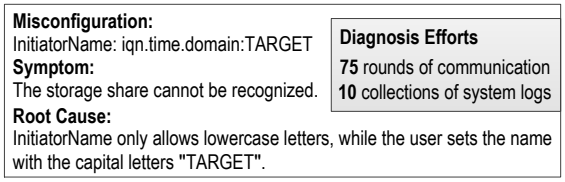


Figure 1: A real-world example from a commercial company. The configuration constraint was too strict and users made mistakes despite two documents explaining it.

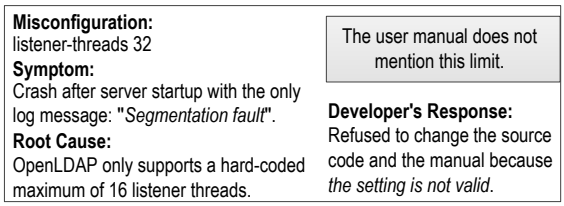


Figure 2: A real-world example from OpenLDAP. The server crashes when "listener-threads" is set to be larger than 16. More real-world examples are given in Figure 7.

Figures 1 and 2 give two real-world examples to further illustrate the points above. As shown in Figure 1, a commercial system[1] required users to type all lowercase letters to configure the initiator names of iSCSI adapters. This requirement is too strict. As a result, several customers made mistakes and had to call the company to help troubleshoot the problem. In this particular case, the diagnosis took over 75 rounds of communication with the customer as well as 10 rounds of debugging message collection. It resulted in not only customers' downtime but also high supporting cost.

The second example, as shown in Figure 2, is from the latest version of OpenLDAP. With the parameter, "listener-threads", configured to be larger than 16, the LDAP server would crash after startup with "segmentation fault." The crash symptom misled at least two users to report it as a software bug. This problem is detected by our tool. Unfortunately, after we reported this problem, the developer refused to take any action, such as changing the configuration design, editing the manual entry, or adding code to check the value and printout explicit error messages. This was mainly due to the common attitude many developers have towards configuration errors: "*It is not a bug, but an invalid setting.*"

Of course, not all developers are like this. Some developers have a more responsible attitude towards handling configuration errors. For example, after we reported the *misconfiguration vulnerabilities* (bad system

[1] We are required to keep the company and the product anonymous.

reactions to configuration errors such as crashes, hangs, silent failures) and error-prone constraints to Squid (an open-source Web proxy and cache server), Squid developers fixed the reported problems immediately. Also, the large U.S. commercial company we worked with has been very cooperative, allowing us to publish our evaluation results of their system.

Certainly, the ultimate solutions to avoid misconfigurations are auto-configuration and completely rethinking, redesigning configuration to prevent user mistakes. While these solutions are revolutionary and fundamental, they are challenging and probably prohibitively difficult, because they have to balance two conflicting goals: usability and flexibility (to adjust the system). In addition, not every configuration parameter can be automatically configured. Moreover, few user studies have been conducted to design configuration in better ways.

1.2 Our Contributions

In this paper, we make one of the first steps towards taking an active role in handling misconfigurations. Our approach is more evolutionary and practical. Specially, the solutions and proposed changes in this paper can easily be adopted by existing software systems. In particular, we aim at improving the configuration design of today's software systems by (1) hardening systems against configuration errors; and (2) detecting certain types of error-prone configuration design and handling.

Achieving the above goals would need the specification of configuration requirements referred to as *configuration constraints* in this paper. A constraint for a configuration parameter specifies its data type, format, value range, dependency and correlation with other parameters, etc., in order to configure the parameter correctly. Since large-scale systems usually contain hundreds or even thousands of configuration parameters, it is time-consuming and error-prone to let developers specify each constraint manually [16]. Another solution is to leverage user manuals. Unfortunately, manuals are written in natural languages and are hard to analyze automatically. Moreover, user manuals are often incomplete and outdated as shown in a recent study [26].

As source code always contains up-to-date information, our idea is to automatically infer configuration constraints from source code by analyzing how the configuration parameters are read and used. We implement our idea in a tool called **SPEX**. Furthermore, SPEX leverages the inferred configuration constraints to: (1) harden systems against misconfigurations by injecting errors that violate the constraints, in order to expose misconfiguration vulnerabilities; and (2) detect certain types of error-prone configuration design and handling, in order to make them more user-friendly.

We evaluate SPEX with the latest versions of one commercial system from a major U.S. storage vendor, and six open-source server software including Apache, MySQL, PostgreSQL, OpenLDAP, VSFTP, and Squid. SPEX automatically infers a total of 3800 constraints for more than 2500 configuration parameters. Based on these constraints, it exposes 743 misconfiguration vulnerabilities that caused system misbehavior such as crashes, hangs, indeterminate failures, etc. It also detects at least 112 error-prone configuration constraints. To this day, 364 vulnerabilities and 80 error-prone constraints have been confirmed or fixed by developers after we reported them. Section 5 reports our experiences in interacting with developers during our work and some good practices we have observed. Our results have influenced the Squid Web proxy server to improve its configuration library towards a more user-friendly design, benefiting more than 150 configuration parameters in Squid.

2 Configuration Constraint Inference

This section describes the design and implementation of SPEX, a tool that automatically infers configuration constraints (i.e., rules that differentiate correct configurations from misconfigurations) from source code. In the next section, we will discuss how we use such constraints to expose misconfiguration vulnerabilities, and to detect error-prone configuration design and handling.

SPEX requires the target software's source code and simple annotations as a starting point to help identify and analyze configuration parameters in source code. In this section, we first describe what kinds of configuration constraints we can infer and then discuss how to infer them. Finally, we discuss the limitations, in particular, what kinds of constraints cannot be inferred by SPEX.

2.1 What Constraints Can Be Inferred?

Many configuration requirements are reflected in the software's source code. Examples include data types, formats, value ranges, multi-parameter dependencies, etc. Some of these can be automatically inferred via static code analysis by leveraging the properties of various operations and system/library APIs when accessing (reading or assigning to) configuration-related variables.

Of course, as we will discuss in Section 2.3, not all configuration constraints are reflected in source code or can be automatically inferred via static analysis. This work provides a first step in this direction. Our evaluation has shown promising results with even a modest real-world impact on both commercial and open-source software, as briefly presented in Section 4.1.

SPEX analyzes source code and infers constraints that manifest through concrete, recognizable program pat-

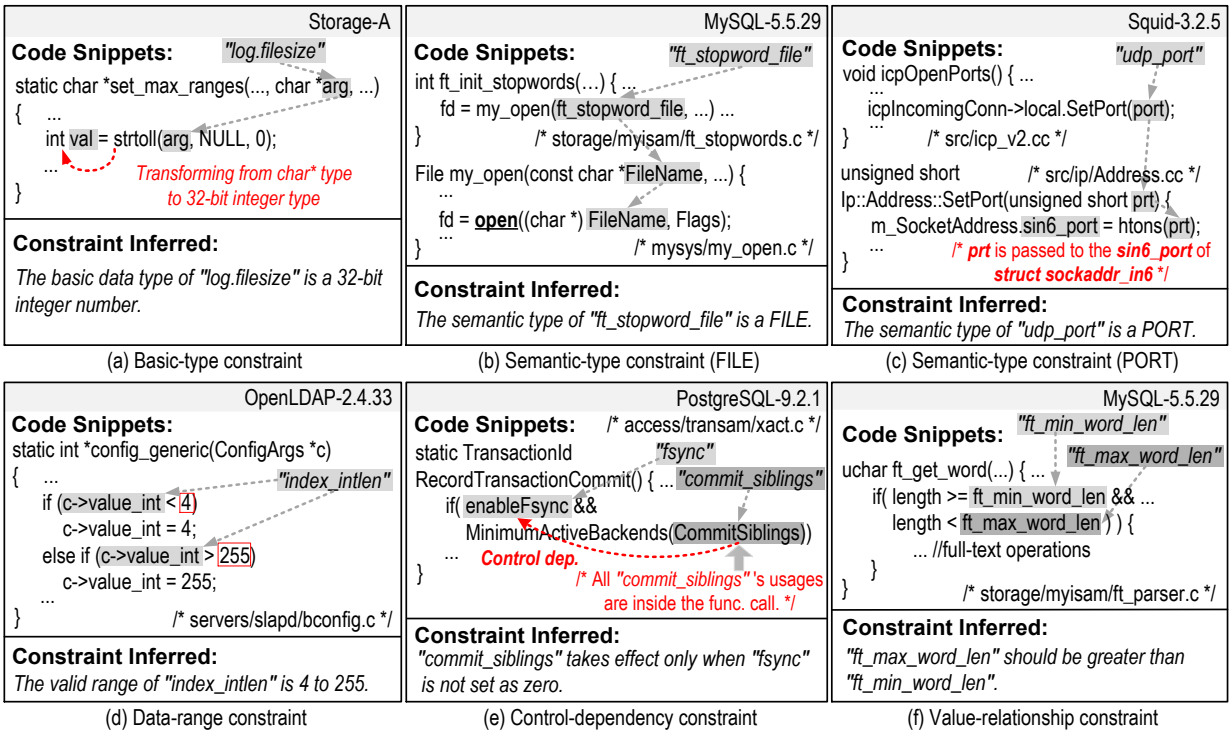


Figure 3: Real-world examples to illustrate what configuration constraints our SPEX infers. The arrows show the data-flow, which motivates SPEX to do data-flow analysis. Configuration parameters are quoted in the figure, and the program variables that store the parameters are shaded. Section 2.2 explains how these constraints are inferred.

terns. These constraints can be classified into *attributes* and *correlations*. The former define the correct settings of a parameter, while the latter specify the correlations among multiple parameters. Figure 3 gives several concrete real-world examples of various kinds of configuration constraints our SPEX infers. We describe each kind in more detail as follows. The next subsection will explain in detail how SPEX infers them, starting from how it identifies configuration variables in source code.

Data Type: To set a configuration parameter correctly, users first need to know the expected data type. We call such constraints *type constraints*. There are two classes of data types for configuration parameters: *basic types* and *semantic types*. The basic-type constraint specifies a parameter's value by the low-level data representation including integer, character, boolean, floating-point number, string, etc.

However, basic types alone may not be sufficient. For example, a "string" parameter may refer to either a file path or an IP address. Each such semantic type has its own specific requirements. For example, a file path has a specific path-like format and should represent a valid file in the file system. In addition to the "file path" and "IP address" types, there are many other types such as user name, port number, timeout, etc. In SPEX, we support the high-level semantic types of most standard libraries.

Figures 3(a), (b), and (c) show three real-world examples of type constraints inferred by SPEX. In the first example, via static code analysis, SPEX infers the parameter, "log.filesize", to be a 32-bit integer number. Figure 3(b) gives an example of the "FILE" type, and Figure 3(c) shows an example of the "PORT" type.

Value Range: Configuration parameters may be further constrained by some acceptable ranges of valid values, such as minimum and maximum values or a list of acceptable values as in the enumerative type. Figure 3(d) shows a range constraint inferred by SPEX from OpenLDAP, in which, as the code indicates, "index_intlen" needs to be between 4 and 255.

Control Dependency: Multiple configuration parameters might have dependencies. Often, the resolution to problems like, "*Why does my setting of parameter A not work?*" is simply, "*Turn on parameter B.*" When such dependencies are neither documented in the manual, nor pinpointed explicitly by log messages, it is difficult for users to figure them out. Such constraints are typically manifested as *control dependencies* in source code.

Formally, we define the control dependency of two parameters as $(P, V, \diamond) \mapsto Q$ which means that the usage of parameter Q relies on the setting of parameter P, under the condition of $P \diamond V$, where $\diamond \in \{<, >, =, \neq, \geq, \leq\}$, and V is a constant value. Figure 3(e) shows an example

from PostgreSQL, where "commit_siblings" takes effect only when "fsync" is non-zero.

Value Relationship: In addition to the control dependency between two parameters, the relationship of their values may also impose constraints. For example, in Figure 3(f), the value of "ft_max_word_len" should be greater than that of "ft_min_word_len".

2.2 How to Infer Constraints?

To infer configuration constraints, SPEX first needs to identify configuration variables in source code. It then tracks the data-flow of each program variable corresponding to the configuration parameter, and records any constraint that is discovered along the data-flow path.

We implement SPEX's analysis to be inter-procedural, context-sensitive, and field-sensitive. Inter-procedure is necessary because configuration parameters are commonly passed through function calls. SPEX also needs to be field-sensitive because configuration parameters could be stored in composite data types. SPEX is built on top of the LLVM compiler infrastructure [18].

As a design choice, we do not use symbolic execution for SPEX. Symbolic execution is able to explore all the possible code paths in the program for the given input. However, it suffers from path explosion when applied to large systems such as Storage-A. Moreover, as shown in Section 2, SPEX looks for concrete code patterns on the data-flow path of each configuration parameter, which does not fit the strength of symbolic execution.

SPEX scans the source code twice. In the first pass, it infers the data-flow path of each parameter and looks for data-type and data-range constraints for each parameter. To further infer constraints involving multiple parameters (i.e., control dependencies and value relationships), SPEX scans the code again, but this time only on the program slice containing the data-flow of each parameter.

2.2.1 Mapping Parameters to Variables

To start constraint inference, SPEX has to know the program variables that store the values of configuration parameters. Different software projects may have different conventions. We observe that developers often use clean interface to manage the mapping information. By examining 18 widely-used software projects (shown in Table 1), we find that all but one of them map configuration parameters into program variables via one of the three interfaces: *structure*, *comparison*, and *container*. Correspondingly, SPEX provides three template toolkits to extract the mapping information with minimal annotation efforts.

In structure-based mapping, data structures are used to directly map each configuration parameter to the corresponding variable(s) in source code [as shown in Figure 4(a)], or to the parsing function [as shown in Figure 4(b)]. In the former case, developers only need to provide the structure variable's name and each specific field. For Figure 4(a), three lines of annotations are sufficient to extract the mapping information of 82 parameters in PostgreSQL. In the latter case, developers need to further annotate which parameter in the parsing function is the configuration variable [e.g., `arg` in Figure 4(b)].

Comparison-based mapping, as shown in Figure 4(c), uses string comparison functions (e.g., `strcasecmp`) to match parameters. It further assigns values to the variables in the branch blocks. SPEX recognizes standard string comparison functions. In this case, developers need to annotate the parsing function and the initial input variables holding the parameter names and values.

Container-based mapping, exemplified in Figure 4(d), stores all the configuration parameters in a central container and uses common getter functions to retrieve the

Software	Desc.	Type	Software	Desc.	Type
Storage-A	Storage	struct	Squid	Proxy	comparison
MySQL	DB	struct	Redis	DB	comparison
PostgreSQL	DB	struct	ntpd	NTP	comparison
Apache httpd	Web	struct	CVS	SCM	comparison
lighttpd	Web	struct	Hypertable	DB	container
Ngnix	Web	struct	MongoDB	DB	container
OpenSSH	SSH	struct	AOLServer	Web	container
Postfix	Email	struct	Subversion	SCM	container
VSFTP	FTP	struct	OpenLDAP	LDAP	hybrid

Table 1: Parameter-to-variable mapping in 18 software projects. All of them fall into one of the three conventions: structure, container, comparison, or their combinations.

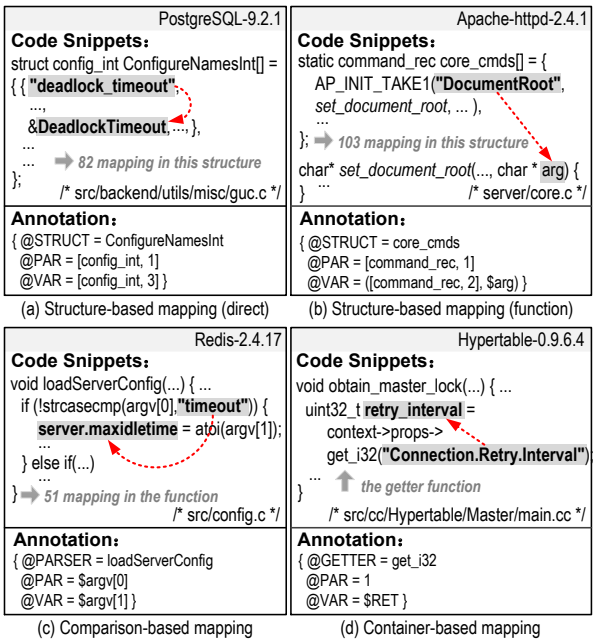


Figure 4: Examples of mapping conventions, and the corresponding annotations to get the mapping information.

value. In such cases, developers need to annotate the getter functions (typically only a few).

By asking developers to annotate the mapping interfaces rather than every mapping pair, the toolkits require a limited amount of information from developers. In the evaluation, the number of annotations needed for most software is less than 10, as shown in Table 4. Note: The annotation only requires modest understanding of source code. The configuration-related code is usually modularized and can be found by simply searching parameter names in source code (e.g., using `grep`).

Starting from the annotations, the SPEX toolkits infer the mapping information in the form of key-value pairs: ("parameter name", `variable name`). For example, the key-value pair in Figure 4(a) is ("deadlock_timeout", `DeadlockTimeout`). In the remainder of this section, we refer to the variables storing the parameters' values as "parameters," to simplify our description.

2.2.2 Data Type Inference

Basic Type: SPEX infers each parameter's basic type from its type information in source code. On the data-flow path of a parameter, its type might be casted multiple times. In such cases, we record the type after the first casting as the basic type, because it is common for a parameter to be first stored as a string (e.g., a `char` array) before being transformed into its real type. Figure 3(a) shows an example from the commercial software Storage-A, in which the parameter is converted from a string to a 32-bit integer. Thus, the basic-type constraint of "log.filesize" is inferred as 32-bit integer.

Semantic Type: SPEX infers semantic-type constraints by searching the following patterns along a parameter's entire data-flow path: (1) the parameter is passed to a known function call (e.g., system- and library-call) or a known data structure; or (2) the parameter is compared with, or is assigned with, the return value of a known function call (e.g., the return value of the `time` syscall).

Figure 3(b) shows an example from MySQL of the first pattern. In this example, SPEX infers the semantic type of "ft_stopword_file" to be a file path because it is used in the `open` syscall. Note: SPEX searches such patterns along the entire data-flow path, even after the parameter is modified, because the modification seldom affects the semantic type. For example, a file path after canonicalization is still used as a file path.

Currently, SPEX supports standard library APIs and data types. In addition, we also allow developers to import their own library APIs and data types by pointing to their header files. For example, for the commercial storage software used in our evaluation, we also imported its proprietary library APIs. For constraint inference, the library APIs included in `.h` files are enough, but for misconfiguration injection described in the next section, we need developers to provide types of configuration errors to inject for each customized data type. Note: They do not need to provide such information for types defined in standard libraries. In our evaluation, such customization is used only for the commercial storage system.

2.2.3 Data Range Inference

SPEX infers range constraints when the parameter is compared with constant values in conditional branches. SPEX infers two types of ranges: numeric and enumerative. For numeric comparison, SPEX treats the constant numbers as *thresholds* of the data range. Enumerative ranges are inferred if the parameter is used in `switch` statements or "`if...else if...else`" logics.

For each range inferred, SPEX further decides whether the range is valid or not by analyzing the program behavior within the corresponding branch blocks. The reason for inferring such information is to guide misconfiguration injection to expose bad system reactions. If in the branch block, the program exits, aborts, returns error code, or resets the parameter, SPEX treats the range as invalid. Otherwise, it is valid. Figure 3(d) shows an example of range inference from OpenLDAP, in which the range of "index_intlen" is divided into $(-\infty,4)$, $[4,255]$, and $(255,+\infty)$. Both $(-\infty,4)$ and $(255,+\infty)$ are invalid because the parameters are reset in those ranges. The `default` in a `switch` statement or the last `else` in "`if...else if...else`" logics is also treated as invalid. Please note: Since such information is used to guide misconfiguration injection, some false positives are not a major concern. It just wastes some testing time.

As a good practice, range constraints should be explicitly documented, but this is not always the case. As shown in Figure 3(d), OpenLDAP limits index lengths within $[4,255]$. However, this constraint is not documented. If users set out-of-range values, the system misbehaves silently, leaving users suspecting it as a bug.

2.2.4 Control Dependency Inference

To infer control dependencies, SPEX starts from the usage statements of a parameter Q, and looks for conditional branches that dominate these statements in a bottom-up manner. If the condition involves the variable that is part of the data-flow of another parameter P, SPEX records a control dependency between P and Q in the form of $(P,V,\diamond) \mapsto Q$.

Figure 3(e) gives an example of a control dependency from PostgreSQL. Starting from the usage statement of "commit_siblings" inside a function call (omitted in the figure), SPEX goes backwards to check the conditions that allow the execution of this usage and infers the dependency: ("fsync", $0, \neq$) $\mapsto$ "commit_siblings". Note:

Passing a parameter to a function and modifying its value are not considered as *usage* because they do not change program behavior [29]. They have to be used in branches, arithmetic operations, and system-/library-call arguments to be considered as usage statements.

However, if we blindly treated every such occurrence of control dependencies as one constraint, there would be many false ones. For example, VSFTP has three parameters: "listen" (for ipv4), "listen_ipv6", and "listen_port". "listen_port" is used after the check of "listen" and the check of "listen_ipv6". If we blindly generated two constraints: ("listen", $1, =) \mapsto$ "listen_port" and ("listen_ipv6", $1, =) \mapsto$ "listen_port", both would be too strict. To handle this problem, SPEX aggregates all the inferred control dependencies for each parameter from all control-flow paths, and calculates the MAY-belief confidence of each dependency in a way similar to [10]. If the confidence exceeds a predefined threshold (currently set to 0.75), the dependency will be reported. In the above example, each dependency will have a confidence of 0.5, not exceeding the threshold. Therefore, both of them are filtered out.

2.2.5 Value Relationship Inference

Similar to control-dependency inference, the value relationship also involves multiple parameters. SPEX looks for comparison statements in parameters' usage. If two variables from different parameters' data-flow paths are compared with each other, SPEX infers the value relationship of the two parameters in the form of $P \diamond Q$. In addition, the value relationship is transitive, which means it can be transited through intermediate variables. Figure 3(f) gives such an example from MySQL that the min-max relation is transited by a local variable. In the current prototype of SPEX, we only check one intermediate variable for transitivity, which is fast and captures common cases. SPEX further tries to decide whether the inferred relationship indicates a valid setting or not, in a manner similar to that in range-constraint inference.

2.3 Discussion and Limitation

No tool is perfect, and SPEX is no exception. SPEX cannot infer all configuration constraints and it also has false positives, even though our evaluation with commercial and open-source software has shown good results.

Currently, the constraint inference of SPEX is limited within the scope of a single program. However, when we study real-world misconfiguration issues (presented in Section 4.2), we find that cross-software configuration correlations also account for a considerable number of misconfiguration cases. Inferring these constraints requires new techniques to consider the software stacks as a whole, which remains our future work.

Even within a single program, SPEX does not infer all constraints. Some constraints are program-specific without common, concrete program patterns. For example, it is hard for SPEX to understand the complicated string manipulation logics used in parsing certain parameters (e.g., nesting and semi-structured rules), which might appear in software providing services of networking and access controls (e.g., Bind9, Netfilter). Moreover, SPEX cannot infer all the possible semantics of parameters.

The constraints inferred by SPEX are basic and cannot capture certain complicated constraints (e.g., dependencies involving complicated compositions of boolean or arithmetic operations). Fortunately, according to our inspection, systems seldom have these complicated constraints on the configuration, possibly because users cannot handle such complexity.

Not every constraint inferred by SPEX is a true constraint. Section 4.3 provides the evaluation results for false positives. SPEX's inference accuracy is above 90% for most evaluated software. To further improve accuracy, we would need developers to manually examine each constraint and prune out the 10% false ones.

The analysis of SPEX works on LLVM's intermediate code representation (IR), a generic assembly language in the static single assignment (SSA) form [18]. Thus, SPEX is applicable to software programs written in programming languages that can be compiled into LLVM IR. In our evaluation, we use Clang as the front-end tool to compile C/C++ source code into LLVM IR.

3 Use Cases of Configuration Constraints

3.1 Harden Systems against Configuration Errors

Given the configuration constraints inferred by SPEX, we take one step further. We build a misconfiguration injection-based testing tool called SPEX-INJ, to expose misconfiguration vulnerabilities. SPEX-INJ automatically generates configuration errors by violating the constraints inferred by SPEX. Then, it injects the errors to the configuration settings and tests how the system reacts. If the system does not react well (e.g., crashes, hangs, failures), SPEX-INJ reports the bad reactions to the developers. By fixing these vulnerabilities (e.g., adding checks and log messages to detect and pinpoint the errors), developers can harden systems against users' misconfigurations, and allow users to quickly find their configuration errors so as to fix the errors by themselves.

Misconfiguration Generation and Injection: Table 2 summarizes how SPEX-INJ generates configuration errors by intentionally violating the inferred constraints. Each misconfiguration includes one or several erro-

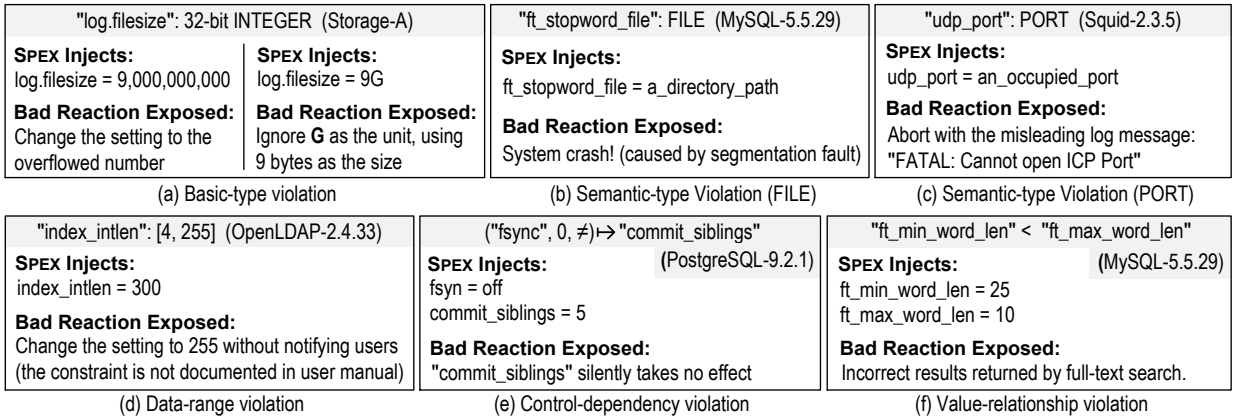


Figure 5: Real-world examples to illustrate the configuration error generation of SPEX-INJ (based on the rules in Table 2), and the exposed misconfiguration vulnerabilities (bad system reactions). How the constraints are inferred from these examples is shown in Figure 3. All the vulnerabilities are detected by SPEX-INJ in the latest versions of the evaluated systems.

Constraint	Generation Rules
Basic type	Generate parameter values with invalid basic types
Semantic type	Generate invalid parameter values specific to different semantic types
Range	Generate out-of-range values
Control dep.	Generate $(P \overline{\diamond} V) \wedge Q$ for $(P, V, \diamond) \mapsto Q$
Value relat.	Generate invalid value relationships

Table 2: SPEX-INJ generates configuration errors for different types of constraints inferred by SPEX.

Reaction	Description
Crash/Hang	The system crashes or hangs.
Early termination	The system exits without pinpointing the injected configuration error.
Functional failure	The system fails functional testing without pinpointing the injected error.
Silent violation	The system changes input configurations to different values without notifying users.
Silent ignorance	The system ignores input configurations (mainly for control-dependency violation).

Table 3: The category of bad system reactions.

neous parameter values that violate a specific constraint. SPEX-INJ may generate several misconfigurations in various aspects for a parameter: violating the constraints of its data type, its data range, its dependencies and correlations with other parameters. Every generation rule is implemented as a plug-in, which can be extended for customization. Figure 5 lists several real-world examples for each rule along with the exposed vulnerabilities.

SPEX-INJ injects misconfigurations by replacing the default parameter values with the generated erroneous values in configuration files. We use the configuration file parser in ConfErr [15] to parse a template configuration file into an abstract representation (AR), and transform the modified AR with errors injected to a usable configuration file for testing. In fact, other configuration file parsing tools such as Augeas can also be used.

Category of Misconfiguration Vulnerabilities (Bad System Reactions): When a misconfiguration occurs, the system should pinpoint either the misconfigured parameter's name/value or its location information (e.g., line numbers in the file). Otherwise, SPEX-INJ considers the system reaction as a *misconfiguration vulnerability*.

Table 3 categorizes different types of misconfiguration vulnerabilities. The first category, system crashes and hangs, is considered as severe vulnerabilities, especially for server applications where availability is crucial. Such symptoms would mislead users and support engineers to suspect them as software bugs. The second category, early termination without pinpointing message, is also undesirable. In this case, the system terminates itself but does not give useful feedback for users to fix the problems by themselves. Similarly, function failures without pinpointing error messages can also confuse users, as shown in the MySQL example in Figure 5(f). As for the last two categories, it is still unacceptable (maybe less severe) to silently violate or ignore the users' intention, which might cause users' confusion or sophisticated problems (e.g., performance issues, feature not activated), as shown in Figure 5(a) and (d).

In this paper, we do not consider performance issues caused by misconfigurations, mainly because of the difficulties in objectively judging whether the performance is acceptable. Unless the performance degradation affects the system usability (belonging to "hang"), we consider it acceptable as long as the functionality is correct.

Testing and Analysis: SPEX-INJ leverages each software's own test infrastructure, including test cases and test oracles, for accepting/rejecting test results. For each generated configuration file (containing one misconfiguration), SPEX-INJ first launches the target system. If the system successfully starts, SPEX-INJ will further ap-

ply existing functional test cases one by one and monitor the system status and output. During testing, SPEX-INJ records all the system and console logs. If the test results fail to pass the test oracles, SPEX-INJ checks the logs to see whether the system pinpoints the misconfiguration. If not, it generates an error report for the developers.

The error report (the output of SPEX-INJ) contains the constraint, the injected error, and the failed test cases, associated with all the log messages. Therefore, the developers can know what misconfigurations caused what problems. SPEX-INJ reports silent violation/ignorance if the system does not pinpoint errors but passes testing.

The testing process can be slow, as $N \times T$, where N is the number of misconfigurations SPEX-INJ generates and T is the time to run all input test cases once. To shorten the time, we apply two optimizations. First, for each misconfiguration, SPEX-INJ stops immediately after the first failed test case. Second, we sort the running time of each test case and run the shortest test case first. By using these optimizations, the testing time of SPEX-INJ on the evaluated software is under 10 hours. Note: This is a one-time cost because SPEX-INJ can be made incrementally. Only those constraints affected by code modification during each revision need to be retested.

3.2 Detect Error-Prone Design

Configuration settings which are expected to be performed by users, should be intuitive and less prone to errors. Carefully-designed configuration constraints can prevent users' confusion and mistakes. More specifically, since the configuration setting is also one type of software interface exposed to the users, it should follow the interface design principles [20, 23].

We expect configuration design to be (1) *consistent* in constraints of different parameters, (2) *explicit* to users when changing (violating) their settings, and (3) *complete* in documenting the requirements of parameters (i.e., constraints). In this section, we show how to leverage the constraints to detect error-prone configuration design and handling that break these three principles.

Design Inconsistency: Consistency is a primary interface design principle to prevent user mistakes. The inferred constraints provide opportunities for detecting two types of configuration inconsistency: (1) case sensitivity, and (2) unit granularity. Such inconsistency is error-prone because users are likely confused by the contradictory requirements for parameters of same types.

Figures 6(a) and (b) show two real-world examples of the two types of constraint inconsistency. In Figure 6(a), different from most string case-insensitive configuration parameters in MySQL, the values of parameter "innodb_file_format_check" are case sensitive. In

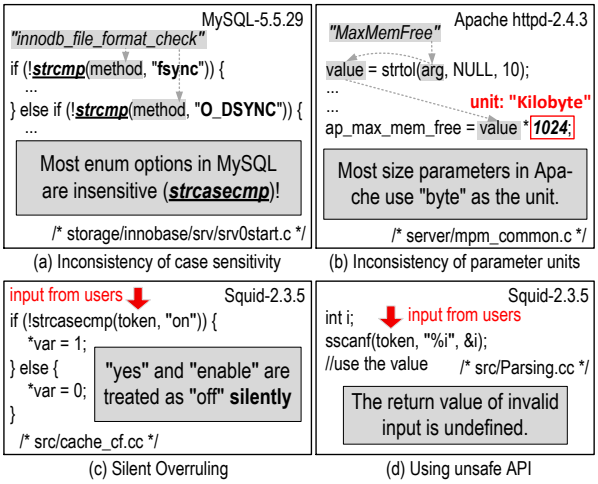


Figure 6: Real-world examples of error-prone configuration design and handling in source code.

Figure 6(b), different from the other *size* parameters in Apache that use *Bytes* as the unit, "MaxMemFree" uses *KBytes* as the unit. Therefore, users can easily make mistakes here due to the inconsistency. As shown in Section 4.1, we find that more than half of the evaluated systems have these two kinds of inconsistency.

The inconsistency is detected based on SPEX's inference of semantic-type constraints. Remember that SPEX records the API calls that use the parameters. The case sensitivity is inferred by identifying string comparison functions. If the parameter is used in comparison functions like `strcasecmp`, it is case insensitive. Otherwise it is sensitive when used in functions like `strcmp`. Similarly, the unit information is inferred according to the API's unit. For example, parameters used in `sleep` have the unit second, while parameters used in `usleep` are of unit microsecond. We also consider the transformation of the parameter, along its data-flow path before it falls into the API call, as shown in Figure 6(b).

Silent Overruling: Silent overruling refers to the case that the system changes an unacceptable user setting into the default value without notifying the user. It may cause silent violation of user intention as one type of misconfiguration vulnerabilities. As shown in Figure 6(c), Squid silently treats any boolean parameter as "off" as long as it is not set to "on", even if its value is "yes" or "enable". Such design can easily confuse users because the system behavior would not match their expectation.

To detect silent overruling, for enumerative range constraints inferred in "`if...else if...else`" or `switch` logics, if the parameter is silently overwritten in the `else` block or `default` case, we flag it as silent overruling. In Squid and Apache, we detect many silent overruling cases that affected 74 parameters. All of these have been fixed by developers after we reported them.

We do not consider static initialization of configuration parameters as silent overruling. It is mainly used to assign default values that would be overwritten by user settings. Thus, it is not relevant to users' configuration.

Unsafe APIs: Using unsafe APIs in configuration handling can also create confusing behavior. For example, unsafe string-to-number transformation APIs, including `atoi`, `sscanf` and `sprintf` are vulnerable to erroneous user inputs. Taking `atoi` as an example, there is no way to check unexpected characters [`atoi(1O0)` returns 1] and overflow issues [`atoi(INT_MAX)` returns -1]. These APIs are handy in controlled contexts but should be avoided in configuration parsing since user inputs may not be trustworthy and can easily be misspelled [39]. Instead, a good practice is to use safe APIs such as `strtol` and check errors through `errno` and end pointers. Most bug detection tools do not report these vulnerabilities because they cannot know whether a variable comes from user settings. SPEX can detect them exactly because it is starting from parameter settings. Our evaluation shows that many systems use unsafe APIs, affecting large numbers of parameters as exemplified in Figure 6(d).

Undocumented Constraints: The inferred constraints are also useful for developers to check whether the constraints are documented in any form (e.g., user manuals, error messages, or even accurate parameter naming). Our evaluation shows that some configuration constraints have never been documented in any form. As the consequence, users can easily make mistakes with them.

4 Evaluation

We evaluate the effectiveness of our tools using one commercial system and six open-source systems as listed in Table 4. The commercial system, Storage-A, is from a major storage vendor in the U.S. It is a distributed operating system used for managing network attached storage devices. It serves storage over networks using both file-based protocols (including NFS, CIFS, FTP, HTTP) and block-based protocols (including FC, FCoE, iSCSI). The system provides users with a large number of configuration parameters. The open-source systems are mature, widely-used server applications with considerable numbers of configuration parameters.

The test cases we use to drive SPEX-INJ are from the test suites shipped with the software projects or provided by the developers. To collect related warning and error log messages, we set sufficient logging verbosity.

Table 4 also shows the numbers of annotations we added in each software so that SPEX can use them as the starting points to identify and analyze configuration-related variables in source code. As shown in Table 4, the annotation efforts in terms of lines are acceptable.

Software	Proprietary	LoC	#Parameter	LoA
Storage-A	Commercial	–	–	5
Apache	Open source	148K	103	4
MySQL	Open source	1.2M	272	29
PostgreSQL	Open source	757K	231	7
OpenLDAP	Open source	292K	86	4
VSFTP	Open source	16K	124	5
Squid	Open source	180K	335	2

Table 4: Evaluated software systems. "–": We are required to keep the concrete numbers of Storage-A confidential. "LoA" is the abbreviation of lines of annotations.

4.1 Overall Results

We first present the end results exposed by SPEX-INJ: the misconfiguration vulnerabilities (bad system reactions) and error-prone configuration design and handling. Later in Section 4.3, we will show the intermediate results: the constraints inferred by SPEX.

Misconfiguration Vulnerabilities: Table 5(a) shows the number of misconfiguration vulnerabilities (bad system reactions) exposed in the latest versions of the evaluated systems. SPEX-INJ exposes a total of 743 vulnerabilities (they are true vulnerabilities verified by us). To this day, 364 of them have been confirmed or fixed by the developers. The vulnerabilities exposed by SPEX-INJ are of various kinds in all the evaluated systems. Most notably, all the open-source systems experienced bad reactions such as crashes, hangs, and early terminations under some misconfigurations. In addition, silent violation and ignorance are more prevalent compared with terminations and failures. This once again reflects that developers pay less attention to defending against misconfigurations as long as they do not affect the system's own execution. Figure 7 gives five additional examples for each type of vulnerabilities exposed by SPEX-INJ.

Since one source-code location could affect the constraints of several configuration parameters, Table 5(b) further shows the number of unique code locations that cause these vulnerabilities. The 743 vulnerabilities are caused by 448 locations in source code, and the 364 confirmed bad reactions can be fixed by 97 code patches.

Error-Prone Configuration Design and Handing: Table 6 shows the distribution of the case-sensitivity requirements for string parameters in each system. We can see that more than half of the systems have inconsistent case-sensitivity requirements. The inconsistent requirements of 80 parameters in Apache, MySQL, and Squid have been confirmed and fixed after we reported them.

Table 7 shows the unit requirements for *size* and *time* parameters. More than half of the systems have inconsistent *size* and *time* units. For example, in Storage-A, 20 *size* parameters use Bytes as their units except three pa-

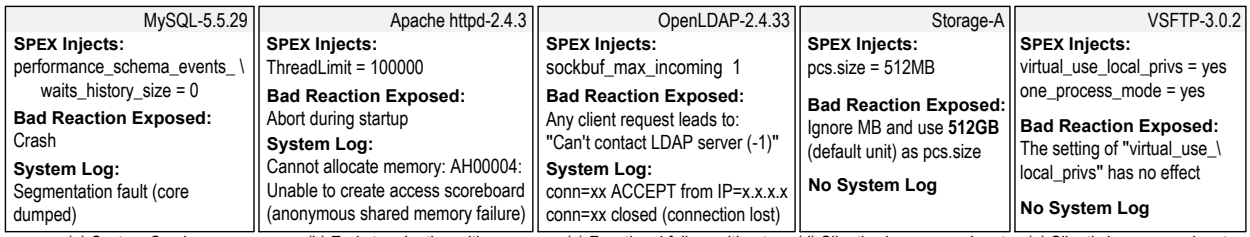
Software	Crash/ Hang	Early terminat.	Functional failure	Silent violation	Silent ignor.	Total
Storage-A	0 (0)	0 (0)	7 (5)	74 (72)	83 (0)	**164 (77)**
Apache	5 (2)	4 (3)	9 (3)	29 (2)	5 (1)	**52 (11)**
MySQL	5 (5)	10 (3)	12 (4)	71 (70)	16 (0)	**114 (82)**
PostgreSQL	1 (0)	10 (1)	2 (0)	1 (0)	35 (2)	**49 (3)**
OpenLDAP	1 (0)	3 (0)	6 (0)	7 (0)	0 (0)	**17 (0)**
VSFTP	12 (12)	5 (0)	18 (0)	23 (0)	68 (0)	**126 (12)**
Squid	2 (2)	3 (2)	29 (1)	173 (173)	14 (1)	**221 (179)**
Total	26 (21)	35 (9)	83 (13)	378 (317)	221 (4)	**743 (364)**

(a) Misconfiguration vulnerabilities (bad system reactions)

Software	Source-code location
Storage-A	119 (34)
Apache	52 (1)
MySQL	46 (16)
PostgreSQL	44 (3)
OpenLDAP	17 (0)
VSFTP	107 (12)
Squid	62 (21)
Total	**448 (97)**

(b) Corresponding code locations

Table 5: The number of exposed misconfiguration vulnerabilities, and the corresponding source-code locations. A patch to one source-code location might fix multiple vulnerabilities. The numbers in "()" are the numbers of confirmed or fixed cases by the developers after we reported them. The cases that have not been confirmed are discussed in Section 5.1.



(a) System Crash (crash/hang) (b) Early termination with misleading message (c) Functional failure without pinpointing message (d) Silently change user inputs (silent violation) (e) Silently ignore user inputs (silent ignorance)

Figure 7: Examples of different types of misconfiguration vulnerabilities (categorized in Table 3) exposed by SPEX-INJ.

Software	Case sensitivity		Developers' fixes
	Sensitive	Insensitive	
Storage-A	**32 (7.1%)**	**453 (92.9%)**	being investigated
Apache	**3 (11.5%)**	**26 (88.5%)**	all sens.→insens.
MySQL	**1 (1.7%)**	**58 (98.3%)**	all sens.→insens.
PostgreSQL	0 (0.0%)	92 (100.0%)	N/A
OpenLDAP	0 (0.0%)	9 (100.0%)	N/A
VSFTP	0 (0.0%)	73 (100.0%)	N/A
Squid	**85 (52.8%)**	**76 (47.2%)**	all insens.→sens.

Table 6: Case-sensitivity requirements of different configuration parameters in the evaluated systems.

Software	Size				Time				
	B	KB	MB	GB	μs	ms	s	m	h
Storage-A	**20**	**1**	**1**	**1**	**2**	**10**	**53**	**12**	**4**
Apache	**20**	**1**	0	0	0	**1**	**26**	0	0
MySQL	**29**	0	0	0	**2**	**2**	**13**	0	0
PostgreSQL	**1**	**3**	0	0	**1**	**12**	**9**	**1**	0
OpenLDAP	**2**	0	0	0	0	0	**3**	0	0
VSFTP	**1**	0	0	0	0	0	**6**	0	0
Squid	**18**	**2**	0	0	**1**	**6**	**33**	0	0

Table 7: The different units of size- and time-related configuration parameters in the evaluated systems.

Software	Silent over-ruling	Unsafe trans-form.	Undoc. Constraints		
			Data range	Ctrl dep.	Val. rel.
Storage-A	0	28	2	0	2
Apache	1	27	0	1	0
MySQL	0	0	4	3	1
PostgreSQL	0	0	3	3	2
OpenLDAP	0	0	2	0	0
VSFTP	0	20	3	47	1
Squid	73	115	3	4	4

Table 8: Other types of error-prone configuration design and handling in the evaluated systems.

rameters, each of which uses different unit size, namely KBytes, MBytes, and GBytes. Storage-A mitigates the inconsistency via naming, including the unit information in parameter names (c.f., Section 5.2). However, none of the open-source systems makes such effort, so the inconsistencies may confuse users and cause mistakes.

Table 8 shows other types of error-prone constraints. SPEX detects 74 parameters with silent overruling in Apache and Squid, all of which were fixed by the developers after we reported them. In addition, more than half of the systems use unsafe transformation APIs for large numbers of parameters. Moreover, a number of inferred constraints are not documented in any form.

However, it might be arguable whether the cases in Table 7 and 8 are really confusing and error-prone to users. To be conservative, we did not report them to the developers. For the same reason, we did not include them in the results presented in the abstract and introduction sections.

Software	Parameter misconfig.	Bad reactions that can be potentially avoided by SPEX
Storage-A	246	68 (27.6%)
Apache	50	19 (38.0%)
MySQL	47	14 (29.8%)
OpenLDAP	49	12 (24.5%)

Table 9: Real-world misconfiguration cases that can be potentially avoided among all sampled historic cases.

Software	Inference incapability		Conform to constraints	Good reactions
	Single-SW	Cross-SW		
Storage-A	19 (7.7%)	51 (20.7%)	76 (30.9%)	32 (13.0%)
Apache	5 (10.0%)	12 (24.0%)	9 (18.0%)	5 (10.0%)
MySQL	1 (2.1%)	12 (25.5%)	18 (38.3%)	2 (4.3%)
OpenLDAP	9 (18.4%)	4 (8.2%)	12 (24.5%)	12 (24.5%)

Table 10: The breakdown of misconfiguration cases that cannot benefit from SPEX/SPEX-INJ. "Conform constraint" and "Good reactions" are explained in the text.

4.2 Benefits to Real-World Configuration Problems

It is hard to predict the benefits of SPEX in avoiding future misconfiguration reports and in reducing misconfiguration diagnosis time. To provide some estimation of the end benefits, we have to leverage past misconfiguration cases committed by real users and evaluate how many customer reports could have been avoided if our tools had been used. Note: The results in this section are from the perspective of system vendors. We do not consider the users' downtime and frustration.

We study real-world historical misconfiguration cases from four systems: Storage-A, Apache, MySQL, and OpenLDAP. For Storage-A, we randomly sampled 246 parameter misconfiguration cases from the company's customer issue database. For open-source applications, we randomly collected 177 parameter misconfigurations from official forums, mailing lists, and ServerFault.com (a popular system administration forum). The data have been presented in our early paper [36].

As shown in Table 9, 24%–38% of the misconfiguration cases could have been potentially avoided if SPEX had been used to improve the configuration design and harden the system against misconfigurations. The results may not sound impressive. However, if we consider the total number of configuration issues encountered in today's server systems, eliminating approximately one-third of the issues is noteworthy. Here, we consider *all* parameter-related configuration errors as the denominator. The percentages will be larger if we consider only one subtype such as illegal misconfigurations [36]. As a *first* step in the direction of improving configuration design, we believe that 24%–38% is a promising result.

To guide future research in this direction, Table 10 further breaks down the misconfiguration cases that cannot benefit from our tools. First, as discussed in Section 2.3, SPEX cannot infer all the configuration constraints. In addition, a configuration setting might conform to the constraints, but does not match the users' intention. For example, a permission setting might be valid from the constraints' perspective, but insufficient for the user to access files. Finally, even if the system already provides "good reactions" by our criteria (i.e., printing log messages containing the faulting parameters), users might still report the problem because the semantics of the text messages might be confusing.

Software	Data type		Data range	Ctrl dep.	Value rel.
	Basic	Semtc			
Storage-A	922	111	490	81	20
Apache	103	22	42	1	9
MySQL	272	74	213	35	10
PostgreSQL	231	52	186	44	6
OpenLDAP	75	15	20	0	2
VSFTP	130	34	84	68	1
Squid	258	46	120	14	9
Total	1991	354	1155	243	57

Table 11: Configuration constraints inferred by SPEX.

Software	Data type		Data range	Ctrl dep.	Value rel.
	Basic	Semtc			
Storage-A	97.0%	95.7%	87.1%	84.1%	94.1%
Apache	96.1%	91.7%	94.6%	100.0%	81.8%
MySQL	100.0%	98.7%	99.1%	94.7%	71.4%
PostgreSQL	100.0%	96.3%	97.3%	91.7%	85.7%
OpenLDAP	88.2%	93.7%	73.1%	N/A	50.0%
VSFTP	100.0%	100.0%	100.0%	63.9%	100.0%
Squid	77.0%	100.0%	100.0%	77.8%	100.0%

Table 12: Accuracy of constraint inference.

4.3 Configuration Constraint Inference

Table 11 breaks down different kinds of constraints inferred by SPEX. It infers a total of 3800 constraints from the evaluated systems. We can see that basic types can be inferred for most configuration parameters. In comparison, the number of semantic types is much smaller. SPEX cannot extract the semantic type for every parameter. It can only infer the semantic type if the parameter interacts with known APIs. Data range and inter-parameter correlations, especially control dependencies, are also common in the evaluated systems.

Table 12 shows the accuracy of constraint inference. We manually and carefully examined all of the 3800 constraints inferred by SPEX. SPEX achieves over 90% inference accuracy in most cases. We find that the inaccuracy is mainly caused by pointer aliasing. If a configuration parameter is pointed by aliased pointers, and/or there are complicated pointer arithmetic logic, SPEX may lose the correct mapping from the configuration parameter to the program variable, and thereby infer con-

straints that do not belong to the right parameter. Currently, SPEX does not perform any pointer-alias analysis. This explains why OpenLDAP has the lowest accuracy: many of its parameters are referenced through pointers. However, our overall accuracy is still over 90%, because most of the configuration parameters are not aliased.

5 Experience and Practice

5.1 Interaction Experience

We reported the detected vulnerabilities and error-prone constraints to developers through the official bug reporting systems. To this day, 364 of our reported vulnerabilities and 80 inconsistent constraints have been confirmed or fixed by the developers. The others are ignored or rejected or being investigated. Here, we share our experiences in interacting with developers.

Positive Experience. We are encouraged by the positive feedback from many developers of the evaluated systems, and we appreciate their help.

- **Storage-A:** Misconfigurations account for one-third of the customer issues of Storage-A in this major U.S. storage company. It has incurred significant financial cost for troubleshooting these issues. Therefore, they actively investigate solutions to misconfigurations and have been very supportive to our work, including providing us with source code, test cases, and allowing us to include Storage-A's results in this paper. All the exposed issues have been sent to the corresponding developing teams. Many of them have been fixed (c.f., Table 5), and others are under investigation.
- **Squid:** The developers immediately paid great attention to our reported misconfiguration vulnerabilities. We worked together and improved their configuration parsing library by adding more checks for configuration errors and more logging in reporting errors.

Negative Experience. Not all interaction with developers is positive. Some of our reports and patches so far have been rejected or ignored. The following summarize the typical negative responses: (1) Some developers think the information is clearly described in the document, so there is no need for systems to check or to pinpoint the configuration errors in log messages — "*The manual states, near the top...*" However, users may not read manuals line by line, especially given that manuals for large systems are usually lengthy (e.g., MySQL-5.5's manual has 4502 pages). Also, users may have problems understanding manual contents because many users come from a different background. (2) Some open-source developers tend to assume that administrators read source code (since it is open sourced) when they configure systems. In the response to one of our patches, the developer wrote, "*Most users never adjust these values. Those who do, read the code.*" Note: Users can read open-source code, but this does not mean that users have time or are willing to read the code. (3) Some developers optimistically assume that users will not make mistakes, "*If you work exactly and carefully, it does not matter; if not, you should not maintain the server at all.*" As a result, it is not uncommon that developers closed the report with comments like, "*This is not a bug.*" The implication is that "*the user must be a novice or not thinking.*" However, such optimistic assumptions are often proved unrealistic as partially demonstrated in our work and previous work on misconfiguration.

The negative experiences indicate that the battle to have developers take an active role in misconfiguration handling is challenging. The main impediment is the controversial responsibilities of misconfigurations between users and developers. Often, it is only until the system suffers considerable support cost or failures (caused by misconfigurations) will the importance of active handling be appreciated by developers. We believe one way to raise this awareness is through education on user-friendly configuration design, hopefully leveraging the trend and attention in good user-interface design raised by Apple's success. As articulated in [24], developers should view system administrators and operators as their first-class users.

5.2 Practice

We highlight some of the good practices we have observed from the evaluated software projects.

Hiding Critical Configurations from Users: Despite the trend that systems expose more and more configuration knobs to users, some systems choose to hide advanced and critical parameters from users, in order to avoid careless mistakes. Storage-A provides two levels of configuration interfaces: one for normal users and the other for advanced administrators. Moreover, it does not allow users to directly modify system configuration files. Users' configuration settings are enforced to go through the interfaces which perform basic checking. In fact, developers sometimes are struggling with the configurability. For example, eight Squid parameters have the following explanation in their manual entries:

> "*Heavy voodoo here. I can't even believe you are reading this. Are you crazy? Don't even think about adjusting these unless you understand the algorithms in comm_select.c first!*"

A good practice should hide such esoteric parameters from users, or forewarn users with clear log messages when they are trying to configure these parameters.

Handling Inconsistency: We observe two efforts in Storage-A in handling unit inconsistency. First, the unit information is exposed in naming (e.g., "cleanup.*msec*", "takeover.*sec*") which serves as both constraint descriptions and mnemonics for users. Second, some parameter settings enforce users to specify unit suffixes to help them express their intention explicitly.

Exploiting Data Structures: Storage-A, MySQL, and PostgreSQL use global data structures which enforce developers to specify the data type and the minimum, maximum value for each configuration parameter. In this way, the systems easily enforce uniform validity checking for configuration settings. Consequently, they have fewer misconfiguration vulnerabilities that violate type and range constraints, as shown in Table 5.

6 Related Work

The major research efforts in addressing misconfiguration problems focus on detecting [11, 35, 38] and troubleshooting [1,3,4,5,21,25,32,33,34,37,40] configuration errors in a timely manner. While these studies provide remedies to find root causes of misconfiguration-induced system failures and anomalies, it is often too late to alleviate users from frustrating experiences.

Our work is different but complementary to misconfiguration detection and troubleshooting. We propose to improve the configuration design, to harden systems with graceful reactions to misconfigurations, and to provide users with explicit log messages so as to enable users to fix configuration errors by themselves. Doing these can help eliminate many configuration errors, or at least help users self-diagnose the problems quickly (based on system error messages) without the need to run any extra detection or troubleshooting tools. Although we have made only a modest step in this direction, we strongly believe that having developers take a more active role to improve configuration design and anticipate/tolerate configuration errors should be the ultimate solution (maybe not immediately achievable).

ConfErr [15] pioneers the configuration testing direction. Since it is not guided by configuration constraints, it makes generic alternations to valid configuration settings (e.g., omissions, substitutions, and case alternations of characters). Similarly, fuzz testing can be used to generate random data as configuration settings. Our work is complementary to ConfErr and fuzz testing. The major part of our work focuses on configuration constraint inference. Based on the inferred constraints, our injection are guided to be program- and constraint-specific. Take the range constraint as an example, SPEX-INJ generates values exactly covering in and out of the specific range. Besides misconfiguration injection, we also leverage the constraints to detect error-prone configuration design and handling. Although not demonstrated in the paper, *the inferred constraints can also be used as references for developers or UI engineers to examine if the configuration constraints are too complicated or unnatural, or not backward-compatible, etc*.

Rabkin and Katz extract configuration parameters together with their data types from Hadoop-like programs [26]. Our work differs from theirs in the following three aspects. First, we have different objectives. Their objective is to understand the types of configuration parameters, whereas ours is to advocate and enable developers to take an active role in reducing configuration-related issues. Second, their work focuses on data types only, whereas our work also extracts other kinds of constraints including data ranges, control dependencies, and value relationships. Third, their work focuses on the characteristics but shows no use case of the extracted information, whereas our work uses the inferred constraints to expose misconfiguration vulnerabilities and to detect error-prone configuration design and handling.

7 Conclusion

This paper advocates the importance for software developers to take an active role in handling misconfigurations. It makes a concrete useful step by providing tooling support for developers to expose misconfiguration vulnerabilities, and detect error-prone configuration design and handling. Our tools have exposed 743 vulnerabilities and at least 112 error-prone constraints in both commercial and open-source systems. To this day, 364 vulnerabilities, together with 80 inconsistent constraints, have been confirmed or fixed by developers after we reported them. Our results have influenced the Squid Web proxy project to improve its configuration parsing library towards a more user friendly design. We hope our work can inspire developers to improve their practices as well as follow-up research in this direction.

8 Acknowledgement

We would like to express our great appreciation to our shepherd, Haibo Chen, who was very responsive and provided us with valuable suggestions to improve our work. We also thank the anonymous reviewers for their insightful comments and suggestions. We are grateful to the Opera group in UCSD, the ATG group in NetApp, Laurent Nicolas, Kevin Nomura, Raghavan Kalkunte, Matus Telgarsky, Brad Chen, and Marc Dacier for their feedback and insights. We thank all the developers who reviewed our reports and patches and interacted with us. This research is supported by NSF CNS-1017784, NSF CNS-1321006, and a NetApp Faculty Award.

References

[1] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)*, April 2009.

[2] Amazon Web Services Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648, 2011.

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

[4] M. Attariyan and J. Flinn. Using Causality to Diagnose Configuration Bugs. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX'08)*, June 2008.

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

[6] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[7] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu. Generic and Automatic Address Configuration for Data Center Networks. In *Proceedings of the 2010 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'10)*, August 2010.

[8] Computing Research Association. *Grand Research Challenges in Information Systems, Technical Report*, September 2003.

[9] S. Duan, V. Thummala, and S. Babu. Tuning Database Conguration Parameters with iTuned. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*, August 2009.

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

[11] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2nd USENIX Symposium on Networked System Design and Implementation (NSDI'05)*, May 2005.

[12] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Tandem Technical Report 85.7*, June 1985.

[13] R. Johnson. More Details on Today's Outage. http://www.facebook.com/note.php?note_id=431441338919, 2010.

[14] A. Kappor. Web-to-host: Reducing Total Cost of Ownership. *Technical Report 200503, The Tolly Group*, May 2000.

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

[16] S. Kendrick. What Takes Us Down? *USENIX ;login:*, 37(5):37–45, October 2012.

[17] N. Kushman and D. Katabi. Enabling Configuration-Independent Automation by Non-Expert Users. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, March 2004.

[19] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfigurations. In *Proceedings of the 2002 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'02)*, August 2002.

[20] D. J. Mayhew. *Principles and Guidelines in Software User Interface Design*. Prentice Hall, October 1991.

[21] J. Mickens, M. Szummer, and D. Narayanan. Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations. In *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML'07)*, April 2007.

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

[23] D. A. Norman. Design Rules Based on Analyses of Human Error. *Communications of the ACM*, 26(4):254–258, April 1983.

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

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

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

[27] A. Rabkin and R. Katz. How Hadoop Clusters Break. *IEEE Software*, 30(4):88–94, July 2013.

[28] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A Declarative Language Approach to Device Configuration. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, March 2011.

[29] M. Sridharan, S. J. Fink, and R. Bodík. Thin Slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, June 2007.

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

[31] Y. Sverdlik. Microsoft: Misconfigured Network Device Led to Azure Outage. http://www.datacenterdynamics.com/focus/archive/2012/07/microsoft-misconfigured-network-device-led-azure-outage, 2012.

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

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

[34] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*, December 2004.

[35] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating Range Fixes for Software Configuration. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

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

[37] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st EuroSys Conference (EuroSys'06)*, April 2006.

[38] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar. Context-based Online Configuration Error Detection. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX'11)*, June 2011.

[39] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging (2nd Edition)*. Morgan Kaufmann Publishers, June 2009.

[40] S. Zhang and M. D. Ernst. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the 35th Internationl Conference on Software Engineering (ICSE'13)*, May 2013.