ConfErr: A Tool for Assessing Resilience to Human Configuration Errors

Lorenzo Keller, Prasang Upadhyaya, and George Candea
School of Computer & Communication Sciences
EPFL (Lausanne, Switzerland)

Abstract

We present ConfErr, a tool for testing and quantifying the resilience of software systems to human-induced configuration errors. ConfErr uses human error models rooted in psychology and linguistics to generate realistic configuration mistakes; it then injects these mistakes and measures their effects, producing a resilience profile of the system under test. The resilience profile, capturing succinctly how sensitive the target software is to different classes of configuration errors, can be used for improving the software or to compare systems to each other. ConfErr is highly portable, because all mutations are performed on abstract representations of the configuration files. Using ConfErr, we found several serious flaws in the MySQL and Postgres databases, Apache web server, and BIND and djbdns name servers; we were also able to directly compare the resilience of functionally-equivalent systems, such as MySQL and Postgres.

1. Introduction

Human error has long been a dominant cause of downtime in computer systems, especially in mission-critical infrastructures. More than two decades ago, 42% of incidents in high-end mainframe installations were found to be due to human operators [4]. Later on, human errors were found to be the cause of 33% of failures at a major Internet portal and 36% at a global content hosting service [9]. A recent study attributed 58% of reported problems in database systems to mistakes made by database administrators [8].

Configuration errors form a significant fraction of human administrator mistakes. Studies have shown that more than 50% of human-induced errors observed in Internet services are configuration errors [7, 9]. A field study found that 24% of Microsoft Windows NT downtime was caused by system configuration and maintenance errors [19]. Configuration errors are particularly pernicious, because they take a long time to discover and fix, thus leading to long repair times [9] and having wide-reaching effects (e.g., a mere DNS misconfiguration made Microsoft's MSN hosting services unavailable worldwide for 24 hours [12]).

Better operator training helps, but often has limited benefits. First, even highly-trained humans working on well-studied tasks with clear life-safety implications face significant error rates; e.g., nuclear plant operators—required by law to undergo extensive training—were deemed responsible for 44%-52% of all significant reactor problems [13]. Second, large software systems exhibit complex behaviors controlled by a dizzying array of configuration "knobs," which overwhelm even the best administrators; for instance, the Oracle 10g DBMS has 220 initialization parameters and 1,477 tables of system parameters [11], along with a 875-page "Administrator's Guide" [10].

It is therefore imperative that critical software systems be resilient to configuration mistakes, since compensating for a poorly designed system is difficult. For example, redundancy and replication have been shown to not improve availability in the face of operator errors [9]. Better user interfaces can help, but cannot turn a system vulnerable to configuration errors into a resilient one (e.g., graphical interfaces do not prevent errors in firewall configuration [16]).

In order to design systems that are less vulnerable to configuration errors, software engineers need tools that quantitatively measure the benefits offered by different techniques and implementations. A suitable, objective benchmark may also encourage system designers to compete on resilience to configuration errors. Benchmarking methodologies have already been proposed in the past: some take system-specific errors and manually port them to different systems, where they can be injected [15, 7], while others use real human operators [3]; these approaches can be expensive and do not always yield objective results.

Automating configuration error testing is key to making such benchmarks uniform, economical, and comprehensive. In this paper, we present ConfErr, a tool that embodies these goals. ConfErr converts the understanding of human errors developed by psychologists and linguists [13, 14] into an automated benchmark for measuring systems' resilience to configuration errors. To our knowledge, this is the first error injection tool to do so.

ConfErr automatically generates and injects realistic errors into a system's configuration files, assesses the target's resilience to the injected errors, and computes the system's resilience profile. This profile can be used for prompt feedback during development (e.g., to quantify the impact of a new feature/design on the ultimate reliability of the system) or as a benchmark to compare different functionally-equivalent systems (e.g., databases from different vendors).

Despite the emergence of configuration wizards and graphical user interfaces, configuration files still represent the primary means of communicating the desired configuration to programs. Moreover, configuration files typically have a pre-defined syntactic structure, making them amenable to systematic, quantitative evaluation. We designed ConfErr to be extensible and accommodate arbitrary error generation plugins; this enables a wide array of configuration errors types, while preserving both the complete automation and efficiency of testing and benchmarking.

This paper makes three contributions: First, we show how models of human error can be turned into practical, realistic error injection tools. Second, we show that using simple abstract representations of configuration files enables these models to be applied portably across different systems. Third, we show that, after running for less than one hour, ConfErr reveals on its own serious design flaws in systems that are critical components of many computing infrastructures today.

In the rest of the paper we describe the human error models used by ConfErr (§2), present the design of the tool (§3), and describe three error generator plugins for ConfErr (§4). Then we present case studies of using ConfErr to find weaknesses in MySQL, Postgres, Apache, BIND, and djbdns, as well as to compare the error resilience of MySQL to that of Postgres (§5). We then review related work (§6) and conclude (§7).

2. Configuration Error Models

Psychology offers insights into why human mistakes occur. Several researchers have studied the psychological underpinnings of human error, with one of the most succinct presentations being the Generic Error-Modeling System (GEMS) framework [13].

GEMS identifies multiple cognitive levels at which humans solve problems. The lowest *skill-based level* is used for common, repetitive tasks. Simple slips and lapses at this level account for about 60% of general human errors; typos and mistakes in digit recognition are examples of such slips. The next level up is the *rule-based level* of cognitive processing, where reasoning and problem-solving are performed by pattern-matching the situation at hand with previously-seen situations and applying the solutions discovered in those previous instances (e.g., applying the configuration principles of one database system to another). Mistakes caused by misapplying such rules account for roughly another 30% of human errors. Finally, the highest cognitive level is the *knowledge-based level*, where tasks are approached by reasoning from first principles, without the direct use of previously-formed rules or skills (e.g., configuring a piece of functionality that was previously never encountered). Mistakes at this level account for the remaining 10% of human errors.

Training and experience move task processing from higher cognitive levels to lower ones: a knowledge-based task may move over time to the rule-based level, as it becomes familiar and encoded into mental rules. Similarly, tasks performed on the rule-based level may move to the skill-based level, if they become so familiar as to be "second nature."

The ConfErr error generators embody three error models that span all these cognitive levels: one models mistakes at the level of individual words in configuration files (§2.1), one models mistakes in the structure of configuration files (§2.2) and another one models semantic errors (§2.3).

2.1. Spelling Mistakes

Typographical errors (typos) occur during the process of typing. Highly disciplined individuals will proofread just-edited configuration files before applying them to the corresponding systems; others rely on the system to identify such problems upon startup. As computers become faster, this latter approach becomes increasingly widely spread, in much the same way programmers type up programs and invoke the compiler without re-reading their code.

Adapting the classification in [14], we divide one-letter typos into the following categories:

- Omissions: One character in a word is missing; this corresponds to characters being missed during hurried typing. Intuitively, we would expect single-letter omissions to be more likely in practice than multi-letter omissions, because more than one missing character is generally easier to notice.

- Insertions: A spurious letter is introduced in a word.

- Substitutions: A letter is replaced with another (incorrect) letter. As will be seen later, we use the keyboard layout to produce realistic single-letter substitutions, based on the model of operators accidentally pressing nearby keys.

- Case alterations: This is special substitution error, in which the case of adjacent letters is swapped due to a miscoordination in pressing the Shift key.
- Transpositions: Two adjacent letters in a word are swapped. Letters in different words are rarely swapped, because humans automatically place cognitive boundaries between words.

2.2. Structural Errors

Configuration files generally have a well-specified structure; mistakes related to this structure trace their roots to all three cognitive levels. We reflect this in our model of structural mistakes.

At the skill-based level, we capture mistaken repetitions of configuration directives and misplacement of directives in sections of the configuration file, as might result from copy-paste operations. Another type of error is omission on account of configuration complexity: while editing a configuration with many parameters required in a section (or many sections required overall), one of these may be simply forgotten.

At the rule-based level, we model operator mistakes resulting from the use of a configuration format that is similar but incorrect. An example of such a mistake would be the "borrowing" of a configuration directive or section from another program configured by the same operator.

Mistakes at the knowledge-based level tend to result from a mismatch between the mental model the operator has of the system and the actual operation of the system [13]. For example, reverting a non-functioning configuration to the default configuration "just to get the system started" may omit critical directives without the operator realizing it.

Structural configuration errors abound in practice. For example, numerous cases for the Apache web server can be found in [1]: a common mistake is the omission of a directive that has to be present in each subsection (e.g., the `ServerName` directive in `VirtualHosts` sections) or the duplication of a directive (such as `Listen` or `NameVirtualHost`), with the final replica overriding all previous ones. Other common mistakes are the addition of wrong directives or entire sections of directives via copy-paste or moving directives to similar kinds of sections (e.g., access restrictions in `Directory` sections, index options for a `ScriptAlias` directory).

2.3. Semantic Errors

In addition to mistakes in syntax and structure, configuration errors can take on a semantic nature, resulting from the administrator's wrong understanding of how the system works. Semantic errors are introduced solely when operating at the highest cognitive level.

In our model, we capture two classes of semantic errors. The first one is inconsistent configurations, in which required constraints are not satisfied. For instance, the value of one parameter (e.g., shared memory pool) may be related in a specific way to that of another (e.g., maximum number of client connections), and an ignorant operator may generate a configuration that does not satisfy this relation. A properly configured domain name service (DNS), for example, should provide both forward and reverse mappings for a given name-IP pair, but an operator might forget to set up one of the two mappings.

The second type of semantic errors occurs when the operator does not know exactly the meaning of a given parameter and uses it to configure a similar but different aspect of the system. For instance, DNS provides multiple record types, and an inexperienced administrator may associate an address to a domain via a CNAME record (normally used to declare aliases, not to assign addresses); this results in all other records associated with the domain name becoming inaccessible, since the unaliased domain name is necessary for correct name resolution.

3. The ConfErr Framework

The goal of ConfErr is to turn error models, such as the ones described above, into practical tools. ConfErr allows error models to be encoded in generator plugins and glued together into an end-to-end injection and measurement system. ConfErr automatically drives all parsing of initial configuration files, generation of errors, injection, startup and shutdown of the system-under-test (SUT), and measurement of the impact of each error on the system; none of these require human intervention.

After a brief overview of ConfErr's design (§3.1), we describe how it creates and uses abstract representations of configuration files (§3.2) and how error generation models are described (§3.3). Complete source code and documentation can be found at `http://conferr.epfl.ch/`.

3.1. Design Overview

At a high level, ConfErr takes in configuration files, mutates them, and then tests the system-under-test (SUT) with the new configurations. The operation is illustrated in Figure 1.

In order to generate a resilience profile for a given systems *S*, ConfErr takes as input a set of initial configuration files for *S* (e.g., `httpd.conf` and `ssl.conf`

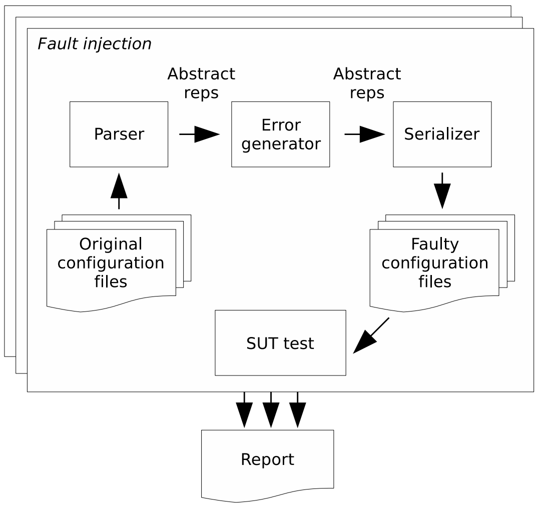


Figure 1. Overview of ConfErr

for Apache), system-specific parsing and serialization plugins, an error generator plugin, and domain-specific functional tests (e.g., database tests, web server tests, etc.). ConfErr then iterates through each configuration file, using the parser plugin to generate the corresponding abstract representations. Error generator plugins synthesize fault scenarios, which are essentially functions that mutate sets of abstract representations of configuration files. The error generator plugins decide where in the configuration(s) and what type of faults to inject. Since the transformation is applied to the entire set of configuration files, it allows for the injection of cross-file errors in addition to errors localized in a single file.

After each mutation, the corresponding serializer plugin generates a new set of (faulty) configuration files from the mutated abstract representations. The newly-generated configuration files are used in place of the original ones; ConfErr starts the SUT and tests its behavior. There are three possible outcomes:

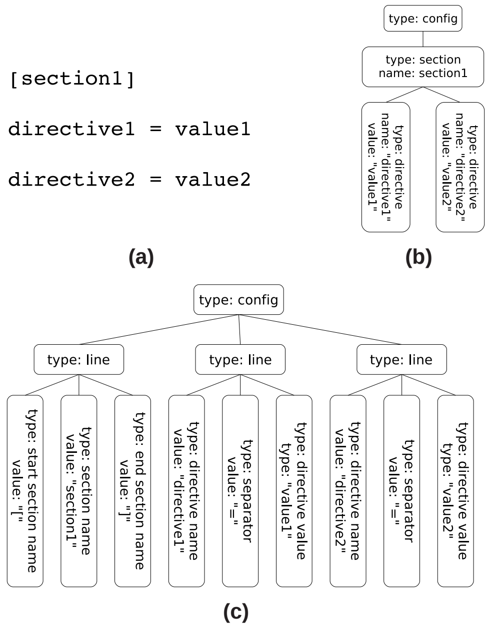
- SUT failed to start; output is recorded in the resilience profile (most likely the SUT has detected a configuration error)
- SUT started but could not complete the functional tests; record failed tests in profile (most likely SUT did not detect a configuration error)
- All tests passed; we record this result as the SUT having successfully handled the configuration mutation

The resilience profile is ConfErr's sole output, and it indicates the result of each synthesized injection test, containing the injected error and the corresponding system behavior. The functional tests executed by ConfErr are based on a set of system-specific scripts; their exit values are used to establish the result of the test, along with recording all the output.

ConfErr can be extended with new error models and system-specific configuration parsers/serializers, as needed.

3.2. Configuration Representation

Configuration files are modeled internally as XML information sets [5], which represent configurations as a collection of information items with a set of associated properties. Some of these properties can point to other information sets, so the model can be seen as a tree of information items. In the rest of the paper we will refer to information sets as trees and to information items as nodes. We chose this data model because it is a good match for the structure of configuration files and it allows us to reuse robust, already-developed libraries and languages for manipulating the configurations.



[section1]
directive1 = value1
directive2 = value2
(a)
type: config
type: section name: section1
type: directive name: "directive1" value: "value1"
type: directive name: "directive2" value: "value2"
(b)
type: config
type: line
type: line
type: line
type: start section name value: "["
type: section name value: "section1"
type: end section name value: "]"
type: directive name value: "directive1"
type: separator value: "="
type: directive value type: "value1"
type: directive name value: "directive2"
type: separator value: "="
type: directive value type: "value2"
(c)

Figure 2. Representations of SUT configuration: (a) original configuration, (b) tree suitable for structural error injection and (c) tree suitable for typo injection

The exact structure and semantics of configuration trees depends on the error plugin that is used (see Figure 2). For instance, when creating typos, it is best to represent configuration files as lists of words grouped in lines. On the other hand, when injecting structural errors, a better representation is one that groups directives into sections.

To enable flexible injection scenarios, ConfErr divides the parsing process into two stages. First, a configuration file is parsed into a tree with a (system-specific) XML representation that is independent of the desired error generator plugin; this representation contains all the information required to recreate a serialization of the configuration in the system-specific format. As input files, ConfErr currently supports traditional line-oriented configuration files, generic XML configuration files, as well as the formats specific to Apache, BIND, and djbdns.

In the second step, the system-specific representation is mapped to the format required by the error plugin using XSLT [18], a standard transformation language. In order to enable the reverse operation, the mapping function creates some additional information that complements the representation specific to the error plugin.

Since different systems use different ways to express the same configuration, it is not possible to use a single representation that can accommodate all systems and fault types. Mapping to multiple representations permits different types of faults to be defined in terms of different views of the system configuration. Fortunately, the same representation can be reused across multiple types of faults, and the transformation function from the system specific representation to the one suitable for fault injection is usually very simple.

After the fault has been injected in the plugin-specific representation of the configuration, ConfErr has first to check that it is possible to transform this tree into the system-specific representation—differences in the expressiveness of the two representations can prevent this operation from completing successfully (for an example, see Section 5.4). If the configuration can be expressed in the system-specific XML representation, it is then possible to serialize it to the system-specific file format.

3.3. Error Templates

With ConfErr, error models are expressed by instantiating and composing a set of base templates. Templates describe a transformation of a configuration tree, such as deletion or duplication of a node. The templates are parameterized, allowing the user to specify which transformation to apply and under which conditions. Given a template, its parameters, and the input configuration files, ConfErr can generate a set of fault scenarios. ConfErr already provides a collection of templates for generating common transformations, and users can add other custom templates.

The simplest class of templates describes mutations of nodes and subtrees; they take as parameter a description of the nodes that should undergo the template-specific mutation. Since configuration files are represented as XML information sets, target nodes are easily specified via an XPath [17] query. Examples of this type of templates include the node deletion template and one that specifies the duplication of part of a configuration tree. A particularly important template is the abstract modify template, that can be specialized to generate sets of fault scenarios that modify the content of nodes; we used this template in the spelling mistakes plugin (§4.1).

ConfErr also provides a collection of complex templates, that take as parameters sets of fault scenarios defined with other templates. Among these we find, for instance, a template that returns the union of the fault scenario sets given as parameters, as well as one that selects a random subset of fault scenarios of a given size. These special templates can be used to compose multiple error models or limit the number of faults that a given model can return.

4. Error Generator Plugins

ConfErr error generators are in charge of specifying the sequence of mutations to perform on configurations in order to generate a meaningful resilience profile. Besides aiming for realistic errors (§2), we also wanted the error generator plugins to be portable across the configurations of a wide variety of applications and systems. Thus, error models are translated into a sequence of parameterized operations on the abstract representation of the system configuration. In this section we describe the implementation of plugins for spelling mistakes (§4.1), structural errors (§4.2), and semantic errors (§4.3).

4.1. Spelling Mistakes Plugin

For the injection of spelling mistakes, we represent the configuration files as a list of tokens with associated types, as shown in Figure 2.c. The token type is used to restrict the injection to a specific part of the configuration (e.g. mis-spell directive names only).

The plugin implements a collection of submodels, one for each particular type of error (see §2.1): omissions, insertions, substitutions, case alterations, and transpositions. Each submodel extends the abstract modify template. The plugin generates errors by choosing random subsets of typos.

In order to mimic real spelling errors, we use an encoding of a true keyboard. For insertions and substitutions, we first find the position on the keyboard of the key that generates the character currently in the position where the substitution/insertion is to happen. Then we find which modifiers (Shift, Alt, etc.) are necessary to generate that character. Finally, based on the keyboard representation, we find all characters that could be generated by a human mistakenly pressing nearby keys with the same combination of modifiers.

4.2. Structural Errors Plugin

For the injection of structural errors, we represent configuration files as a tree of directives and sections (see Figure 2.b). Nested sections form subtrees.

This plugin implements omissions, insertions in the wrong place, and duplication of configuration directives (see §2.2). Omissions of both directives and sections are described using the abstract delete template. Insertions in the wrong place and duplications are expressed with move and copy templates, respectively. The candidates for the mutation depend on the type of error we want to inject: e.g., for changing the order of directives in a section, we restrict the candidates to be directives only, and the destination of the move is restricted to the set of sections containing the currently selected candidate directives. This template-based approach allows ConfErr to simulate human errors without having to simulate the entire reasoning process that leads to the errors.

Note that, depending on the level of expressiveness in the file format used by the SUT, it is possible that some fault scenarios result in abstract representations that cannot be expressed in the system configuration file language. For instance, some systems do not support nested sections. These situations are detected and reported by ConfErr when it tries to serialize the modified configuration.

4.3. Semantic Errors Plugin

A semantic error plugin encodes mistakes that are specific to a class of systems, and are usually based on best practices documents (e.g., RFC-1912 defines a list of common configuration errors for DNS servers). The implementation of these domain-specific errors typically require the definition of a system-independent but domain-specific configuration representation; the errors are then defined using abstract templates applied to this representation. Section 5.4 will describe in detail a semantic error generator for DNS servers.

5. Case Studies

In this section we illustrate the use of ConfErr in assessing the resilience to configuration errors in widely-used systems: the Postgres 8.2.5 database, MySQL 5.1.22 database, Apache 2.2.6 web server, and the ISC BIND 9.4.2 and djbdns 1.05 domain name servers.

We find that configuration error injection can expose serious bugs in production-quality software, and that systematically checking the behavior of systems against common configuration errors can highlight areas where resilience can be improved. We also find that, overall, Postgres is markedly more robust to configuration typos than MySQL; the generated resilience profile reveals simple checks that could significantly improve resilience. Undetected configuration errors often have undesired latent effects, so the earlier they are flagged, the better.

After describing our setup and methodology (§5.1), we present results obtained with the three different plugins (§5.2, §5.3, and §5.4) and then show how ConfErr can be used to objectively compare one system's resilience to that of another.

5.1. Setup and Methodology

ConfErr requires three system-specific components: (a) initial configuration files, (b) parsers / serializers for configurations, and (c) scripts to set up the environment, start/stop the system, and a diagnostic suite to determine the outcome of the error injection. Combining these with the error generators, ConfErr automatically produces a resilience profile for the SUT.

We used the default configuration files that ship with the target systems. For MySQL, Postgres, and Apache, configuration files consist of sections, with Apache additionally having nested sections and Postgres having only one main section. Sections are made up of lines, which can be empty or can contain a directive. A typical directive consists of a name, a separator, and a value; some directives may not have values. Postgres's default configuration has 8 directives, MySQL's has 14, and Apache's has 98. For the DNS server, we carried out the error injection on zone files, that describe the records published by the server; the initial configurations had one forward zone and one reverse zone.

Writing the parsers and transformers to/from plugin-specific representations was trivial, given the simple structure of the configurations.

The error generator plugins impose a hierarchy over the fault space, allowing plugins to declaratively specify broad fault classes and then select one element of each class. Unlike choosing errors randomly over the space of possible errors, ConfErr's approach is considerably more efficient at finding flaws—it introduces a wide variety of errors while eliminating redundancy in injection. For the case studies shown here, the representatives of each fault class were chosen randomly within the class. However, if using ConfErr for benchmarking, the exact faultload could be determined a priori (e.g., based on direct surveys of administrator errors). Also, the evaluation could be carried out with a uniform random distribution and the results processed a posteriori, to reflect any biases that may occur in real life.

ConfErr uses the provided scripts to start/stop the system and evaluate the effect of error injections. The scripts correspond to simple functional tests akin to what an administrator might do to check that a system is OK. For MySQL and Postgres, the diagnosis script creates a database, then creates a table, populates it, and queries it. For Apache, the script performs an HTTP GET operation to download a page from the web server. In the case of DNS servers, the script checks that the server is answering to requests both for the forward and the reverse zone.

Depending on the type of error injected, we expect different outcomes. When parameter values are modified inappropriately, the SUT ought to detect them; when we perform structural modifications that do not change the semantics of the configuration, the SUT ought to operate correctly.

All results were obtained on a Dell Optiplex 745 workstation with an Intel Core2 Duo processor and 2 GB RAM, running Ubuntu Linux. The testing engine, the SUT, and relevant clients were all hosted on the same computer.

5.2. Resilience to Typos

Thoroughly checking configuration files for typos is painstaking but intellectually trivial, so we would expect systems—especially those destined for enterprise use—to excel at it. We therefore set out to measure the resilience of MySQL, Postgres, and Apache to this type of errors.

We injected three types of errors:

- *Deletion of entire directives* – mutate the default configuration file to miss a given directive
- *Typos in directive names* – for each section in the default file, randomly select 10 directives and introduce a typo in each one's name
- *Typos in directive values* – similar to those above, but introduce typos in the directive value instead

Some errors are detected by the system upon startup, others by the tests, and others not detected at all. Each error injection experiment took on the order of seconds (2.2 sec for MySQL, 6 sec for Postgres and 1.1 sec for Apache). Table 1 summarizes the results.

Table 1. Resilience to typos

	MySQL	**Postgres**	**Apache**
# of Injected Errors	327 (100%)	98 (100%)	120 (100%)
Detected			
by system at startup	270 (83%)	76 (78%)	46 (38%)
by functional tests	1	0	6 (5%)
Ignored	56 (17%)	22 (22%)	68 (57%)

Functional tests do not offer significant additional detection power compared to startup-time detection, with the exception of typos in listening ports (which is why 5% of Apache errors were caught by functional tests). Apache and Postgres ignore a large number of the typos in part because their directive names are case-insensitive.

The resilience profiles (not listed here, for brevity), reveal several unexpected weaknesses in the SUTs.

For example, MySQL silently ignores values that are out of bounds and chooses defaults instead (e.g. `key_buffer_size=1` is accepted and ignored, although the value has to be at least 8). Directives that specify numeric values accept a suffix to indicate a multiplier (i.e. K, M, G for Kilo, Mega, Giga, respectively); when parsing such values, MySQL stops after the first multiplier symbol. The result is that a value like "1M0" is accepted as valid, whereas it is clearly an unintended value (the operator likely meant "10M"). Numeric values that start with one of the mentioned suffixes (and are thus invalid) are also silently ignored and defaults are used instead. Directives specified without a value are also accepted and replaced with defaults by MySQL. We believe all of these are obvious typos that could be easily caught by MySQL's configuration file parser.

A design flaw in MySQL invites latent errors: MySQL has a shared configuration file used for the DB server as well as the various auxiliary tools, such as backup. When starting the database daemon, only errors in the server-specific directives are detected, with the rest of the file not being parsed at this time. This means that, if an administrator inadvertently inserts an error in one of the other sections, it will become apparent at the earliest on the next run of the corresponding tool. This is dangerous, because some of these auxiliary tools run unattended, launched by cron jobs during the night, so the administrator does not have direct feedback on the typos.

Apache's parser has weaknesses as well. For instance, directives related to MIME types (`AddType` and `DefaultType`) should take values in the format "type/subtype", as defined in RFC-2045. Apache, however, accepts freeform strings instead, without checking conformance to this pattern. Another weakness is in the `ServerAdmin` directive: according to the manual, it should take a URL or an email address; just like in the MIME case, freeform strings are readily accepted here. Similarly, `ServerName` should only accept DNS host names, but instead accepts anything. Such laxity can prevent the system from functioning at a much later date (e.g., the server administrator may not receive failure notifications because of a malformed address entry).

An interesting feature of Postgres is that it enforces constraints across directives. For example, a typo injected in the `max_fsm_pages` directive (replacing 153600 with 15600) caused Postgres to immediately shutdown with an error message explaining that `max_fsm_pages` must be at least 16 $\times$

`max_fsm_relations`. Such constraint checking helps to promptly correct typos that could have hard-to-diagnose implications if they went unnoticed.

5.3. Resilience to Structural Errors

Configuration files of different systems frequently share a similar structure. This similarity invites operators to reuse the mental model of a given system to configure a different one. However, slight differences in configuration methods can make such reuse error-prone.

An ideal system should accommodate as many ways to express a configuration as possible, while ensuring that such flexibility does not impact the ability of the system to detect wrong settings. This allows the system to be resilient to minor changes in administrators' mental models. ConfErr automatically creates variations of configuration files that can be used by a developer to check the correctness of their implementation, or be used by the end user to automatically discover which classes of variations are supported by a given system.

We would expect the following classes of variations to be accommodated by Apache, MySQL, and Postgres:

- Any ordering of sections is allowed
- Any ordering of directives is allowed within a section
- Directive and section names are case insensitive
- Redundant whitespace between directive names, separators and values are ignored
- Directive names can be truncated, if this does not generate a collision of names

We ran a set of experiments on the 3 SUTs to find which of these variations are supported and whether their implementation is correct. We created variations of the same configuration files used in the experiments of the previous section. For each class of variations, we tested each system with 10 different configuration files. Given that we knew the size of the configuration files, we chose the number of configurations such that a random choice of modifications would yield a reasonable coverage of the possible faults. When the size of the input files is not known a priori, ConfErr can be asked to automatically choose the number of faults depending on the structure of the files. Table 2 summarizes the results.

We find that all SUTs accept most of the mutations, but neither one accepts all of them. While our tests did not uncover any specific implementation errors (i.e., either all configuration files created with a class of variations are accepted or none is), we do believe that all three systems should offer the flexibility of all mutations.

Table 2. Resilience to structural errors

	MySQL	Postgres	Apache
Order of sections	Yes	n/a	n/a
Order of directives	Yes	Yes	Yes
Spaces near separators	Yes	Yes	Yes
Mixed-case directive names	No	Yes	Yes
Truncatable directive names	Yes	No	No
% of assumptions satisfied	80%	75%	75%

5.4. Resilience to Semantic Errors

Internet RFC-1912 defines a list of common DNS configuration errors; these occur at multiple levels, from the choice of names to the relationship between records on different servers. Many of the described errors are related to the structure of records published by servers. We used ConfErr to test the behavior of BIND and djbdns when faced with such misconfigurations. We started from a set of configuration files containing a forward zone with several hosts, the corresponding mail exchanger records, several TXT, RP and HINFO records and several aliases, and a reverse zone that maps IP addresses to their names. We then injected record-level errors described in RFC-1912.

The error generation is system independent; it is defined on an abstract representation that shows the DNS records published by each server. A simple transformation maps the data parsed from the configuration files of each SUT into this representation. Another transformation, that maps the record representation to the system-specific configuration representation, is used to construct the faulty configuration files.

Table 3 shows a small subset of the configuration errors described in the RFC and the corresponding behavior for both DNS servers. The fault injection was carried out on a larger set of errors but, for brevity, we only show some of the more interesting ones here.

Table 3. Resilience to semantic errors

Err#	Description of fault	BIND	djbdns
1.	Missing PTR	not found	N/A
2.	PTR pointing to CNAME	not found	N/A
3.	dupl name for NS and CNAME	found	not found
4.	MX pointing to CNAME	found	not found

The configuration file format used by djbdns allows the administrator to define with a single directive multiple related records; e.g., it is possible to define the A (address) record and corresponding PTR record with one directive. Our test configuration file uses this directive and therefore the fault injection tool cannot find any fault to inject for error (1) and (2), because all record representations containing such faults cannot be transformed back to a configuration file. In BIND, the administrator has to define each record separately and thus this system is subject to such errors. This choice of configuration is a plus for djbdns.

BIND is effective in detecting errors of class (3) and (4). It stops loading the zone and signals the operator the reason for this. Running the same tests on djbdns reveals that, despite its strength resulting from the configuration format, it does not check the consistency of its data from this point of view.

Using ConfErr to inject real world configuration errors that go beyond mere syntactical issues enable the study and testing of whole-system behavior, not just of the configuration parser. A developer can thus use ConfErr to identify non-parse-related areas of the system that require improvement.

5.5. Comparing Error Resilience

A configuration process can be viewed as the transformation of an initial configuration file (usually the default one shipped with the system) into a new configuration file. Such transformation is achieved by adding, deleting and/or modifying directives. An ideal system is able to detect all errors introduced during this transformation. We measure resilience of the system to configuration errors by simulating multiple times this configuration process and determining the percentage of errors that are detected by the system across all experiments.

ConfErr uses a benchmark script to automatically transform initial configuration files into new, valid files; afterward, it creates faulty configuration files based on these new files, and verifies the system behavior. Errors are injected in close proximity to the place where the file has been (validly) modified, thus aiming to simulate the common way in which errors sneak into configurations. This procedure simulates the human configuration process, thus constituting a primitive human error benchmark.

We used this approach to compare Postgres and MySQL. In order to generate the faulty configuration files, we iterated through typos in values of all directives. We did not inject errors in directive names, because all these errors are known to be detected by both systems (§5.2). For the starting configuration, we used a file containing most of the available directives, along with the default values; we skipped all directives that have no default value. Since neither Postgres nor MySQL accept typos in directives with boolean values, we excluded them from the test.

We find that Postgres is more resilient to typos than MySQL. We ran 20 experiments for each directive. In each experiment, we injected one typo in the corresponding directive value. For each directive, we computed the percentage of experiments in which the system detected the error. Figure 3 summarizes the distribution of directives across four ranges of detection: poor (0-25% of faults detected), fair (25%-50%), good (50%-75%), and excellent (75%-100%). Postgres was able to detect more than 75% of the typos in almost 45% of its directives, while MySQL detected less than 25% of the typos in the same fraction of its directives.

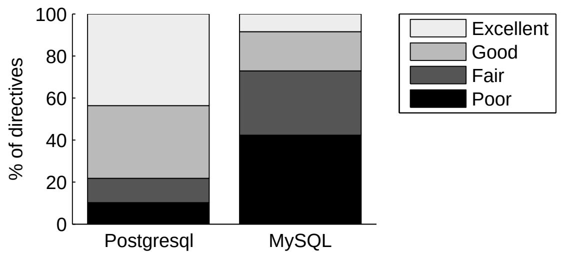


Figure 3. Resilience to typos in MySQL and Postgres, across all directives

The results can be explained by the fact that Postgres features a strong constraint checking mechanism for its numeric parameters, that can detect many typos. Moreover, the MySQL flaws mentioned in §5.2 increase the number of typos that go undetected.

Our comparison only gives an estimate of the overall resilience of the systems to typos in directive values. However, using the same procedure, one could do more focused configuration-task-oriented benchmarks. Using domain-specific knowledge, it is possible to define a subset of directives that are relevant to the task of interest, and obtain a more precise comparison of the task-specific resilience by only taking into account configurations with errors in these directives.

For increased thoroughness, the benchmark can include other types of errors as well, like omissions and duplications. Similarly, a benchmark could include domain-specific semantic error models (e.g., the one in §5.4).

6. Related Work

Several researchers have already recognized human errors as an important factor in system dependability; here we sample prior work related to ConfErr and contrast it to our approach.

Brown and Patterson [2] proposed benchmarks that include the operator as a component of the system that can either increase or reduce dependability: (s)he can help the system recover from external faults faster or can introduce new faults that degrade dependability. With ConfErr, we directly simulate human errors, which results in time and cost savings, but may be less realistic.

Nagaraja et al. [7] described a testbed for fault tolerance techniques aimed at human errors. They emulate the operator via test scripts that embody specific errors observed during tests with real human operators. This emulation technique allows the reuse of faults, thus amortizing the initial cost. Our approach extends this method by synthesizing general error models, thus enabling the simulation of a wider set of errors than the ones initially observed, as well as the automatic introduction of variations.

Vieira and Madeira [15] aimed to assess recoverability in DBMSes by emulating both faults and the recovery procedure carried out by the operator. While their work assumes the error detection capability of the system in order to analyze the recovery procedure, ConfErr directly measures this capability.

Brown and Hellerstein [3] introduced a method for measuring the complexity of configuration processes in terms of time taken to complete the configuration and probability of completing without errors. In our work, instead of focusing on the probability of error, we focus on the system's ability to handle the configuration error.

Finally, the field of human-computer interaction has seen an abundance of work on the design, evaluation and implementation of interactive computing systems. For example, Maxion and Reeder [6] analyzed the genesis of human errors and the impact interfaces have on them. To our knowledge, no automatic tool for error generation has been proposed; such existing work can be leveraged to extend ConfErr's models.

7. Conclusion

Configuration errors are dominant causes of system downtime, but are rarely taken into account when designing, testing, and evaluating systems. Direct testing for this type of errors traditionally involves real humans, so it can be complex, subjective, and hard to reproduce.

In this paper we presented ConfErr, a tool that automatically tests the behavior of a system when faced with human configuration errors. Instead of directly relying on human subjects, the tool relies on models that psychologists and linguists have distilled from their studies of human behavior. ConfErr automatically generates realistic configuration errors, injects them in a system-generic fashion, and assesses their impact. ConfErr is designed to be extensible, thus allowing for the addition of new error generation plugins.

We showed that ConfErr enables a system developer to test with little effort the resilience of real systems—we reported case studies on MySQL, Postgres, Apache, BIND, and djbdns; testing each SUT took less than one hour. We found flaws in these popular server applications and showed how to compare one system to another, thus taking a further step toward dependability benchmarks that include the human factor.

References

[1] Apache HTTP Wiki. http://wiki.apache.org/httpd/.

[2] A. Brown, L. C. Chung, and D. A. Patterson. Including the human factor in dependability benchmarks. In *Proc. DSN Workshop on Dependability Benchmarking*, 2002.

[3] A. Brown and J. Hellerstein. An approach to benchmarking configuration complexity. In *ACM SIGOPS European Workshop*, 2004.

[4] J. Gray. Why do computers stop and what can be done about it? In *Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[5] XML information set. http://w3.org/TR/xml-infoset.

[6] R. A. Maxion and R. W. Reeder. Improving user-interface dependability through mitigation of human error. *Int. J. Hum.-Comput. Stud.*, 63(1-2), 2005.

[7] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in Internet services. In *Symp. on Operating Systems Design and Implementation*, 2004.

[8] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and validating database system administration. In *USENIX Annual Technical Conference*, 2006.

[9] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? In *USENIX Symp. on Internet Technologies and Systems*, 2003.

[10] Oracle Database 10g Release 2 administrator's guide. Oracle Corp., May 2006.

[11] Oracle Database 10g Release 2 reference. Oracle Corp., May 2006.

[12] S. Pertet and P. Narasimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

[13] J. Reason. *Human Error*. Cambridge University Press, 1990.

[14] B. van Berkel and K. D. Smedt. Triphone analysis: a combined method for the correction of orthographical and typographical errors. In *Proc. 2nd Conf. on Applied Natural Language Processing*, 1988.

[15] M. Vieira and H. Madeira. Recovery and performance balance of a COTS DBMS in the presence of operator faults. In *Intl. Conf. on Dependable Systems and Networks*, 2002.

[16] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6), 2004.

[17] XML path language (XPath). http://w3.org/TR/xpath.

[18] XSL transformations (XSLT). http://w3.org/TR/xslt.

[19] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT system field failure data analysis. In *Proc. Pacific Rim Intl. Symp. on Dependable Computing*, 1999.