Challenges and Opportunities: An In-Depth Empirical Study on Configuration Error Injection Testing

Wang Li*
NUDT
China
liwang2015@nudt.edu.cn

Zhouyang Jia*
NUDT
China
jiazhouyang@nudt.edu.cn

Shanshan Li†
NUDT
China
shanshanli@nudt.edu.cn

Yuanliang Zhang
NUDT
China
zhangyuanliang13@nudt.edu.cn

Teng Wang
NUDT
China
wangteng13@nudt.edu.cn

Erci Xu
NUDT
China
ercixu08@nudt.edu.cn

Ji Wang
NUDT
China
wj@nudt.edu.cn

Xiangke Liao
NUDT
China
xkliao@nudt.edu.cn

ABSTRACT

Configuration error injection testing (CEIT) could systematically evaluate software reliability and diagnosability to runtime configuration errors. This paper explores the challenges and opportunities of applying CEIT technique. We build an extensible, highly-modularized CEIT framework named CeitInspector to experiment with various CEIT techniques. Using CeitInspector, we quantitatively measure the effectiveness and efficiency of CEIT using six mature and widely-used server applications. During this process, we find a fair number of test cases are left unstudied by the prior research work. The injected configuration errors in these cases often indicate latent misconfigurations, which might be ticking time bombs in the system and lead to severe damage. We conduct an in-depth study regarding these cases to reveal the root causes, and explore possible remedies. Finally, we come up with actionable suggestions guided by our study to improve the effectiveness and efficiency of the existing CEIT techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software configuration management and version control systems**.

*Both authors contributed equally to this research.
†Shanshan Li is the corresponding author.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464799

KEYWORDS

Configuration, Testing, Empirical Study

ACM Reference Format:
Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and Opportunities: An In-Depth Empirical Study on Configuration Error Injection Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464799

1 INTRODUCTION

Configuration error injection testing (CEIT) is a testing technique to systematically evaluate software *reliability* and *diagnosability* regarding runtime configuration errors (a.k.a. misconfigurations). There has been much research on addressing automated CEIT tools [8, 21, 24, 25, 52, 56]. The main idea is to inject configuration errors into the system under test (SUT), and then observe the SUT reaction under system test suites. More specifically, the tools execute the SUT with each injected configuration error against each test case from the test suite. Then, they evaluate the reaction of the SUT in terms of reliability (whether the SUT can pass the test) and diagnosability (whether the SUT can pinpoint the error).

CEIT has attracted much attention, given the severity and prevalence of configuration errors [7, 13, 17, 20, 22, 29, 35, 41, 50, 53], which are reported as the second largest cause of noticeable service-level disruptions in one of Google's services [13] and contribute to 16% of production incidents at Facebook [46]. Continuously running CEIT to actively expose reliability threats has the potential of reducing production failures caused by configuration errors. Moreover, configuration errors impose a major cost of administration and ownership, because they are time-consuming and challenging to diagnose [9–11, 39, 52, 56]. The essential challenge lies in the fact that the system users who change the configurations may not know how the system consumes configuration values internally. CEIT

could expose diagnosability issues to improve the system feedback (e.g., log messages).

The existing CEIT tools detect reliability and diagnosability issues mainly based on the symptoms of the SUT. We find this process may be limited, since different root causes may result in exactly the same symptom. Without deep understanding of the root causes behind the symptom, many test results of the tools may be indeterminate. For example, `autovacuum_work_mem` is a configuration parameter in PostgreSQL, and its minimal value is 1024. The tools may generate an error value for `autovacuum_work_mem` (e.g., 512), but PostgreSQL will change the value to the minimal value:

```
if (autovacuum_work_mem < 1024){
    autovacuum_work_mem = 1024;
}
```

This example is regarded as a vulnerability [52] in the SUT (System Under Test), which may change user intentions without any notification. In another example, `EnableMMAP` is a configuration parameter in HTTPD. The tools may generate an error value by changing the cases (e.g., *off* to *OFF*) [21, 56], whereas HTTPD parses `EnableMMAP` using `ap_cstr_casecmp` (a case insensitive method):

```
if ( ap_cstr_casecmp(EnableMMAP, "off") == 0 ) {
    d->enable_mmap = ENABLE_MMAP_OFF;
}
```

In this example, the injected error (i.e., *OFF*) is actually a false error. The SUTs in both the above examples show the same symptom, i.e., passing test cases and leaving no log messages. The root causes, however, are different — the former is caused by the vulnerability in the SUT, while the latter is caused by the limitation of error generation of the CEIT tool.

In this paper, we build an extensible, highly-modularized CEIT framework named CeitInspector [1] to understand the challenges and opportunities of applying CEIT in real-world software engineering practice. CeitInspector can effectively integrate and test with various CEIT tools proposed in the literature [8, 21, 24, 25, 52, 56]. Also, it embraces the heterogeneity and complexity of real-world software systems by supporting different system test suites, configuration formats, and operation scripts (e.g., for starting/stopping the system). With CeitInspector, we further lead our discussion with the following research questions:

- RQ1: *How good are the existing CEIT tools in exposing vulnerabilities?*
- RQ2: *Despite all previous analysis on the exposed vulnerabilities, are there still cases left unstudied? If yes, do they matter?*
- RQ3: *Are there potential approaches to further improve the CEIT framework?*

With dataset analysis, field error injection with CeitInspector, and validation experiments, we make the following observations to our research questions:

First, we explore the effectiveness and efficiency of three types of current CEIT tools, i.e., random-, mutation- and specification-based tools. These tools expose 12, 15, and 23 vulnerabilities, respectively. Moreover, the efficiency of the three tools are 10, 3 and 8 vulnerabilities per 1000 injections. The random-based tool, while using the simplest and most efficient configuration-error generation method, has limited effectiveness (i.e., with only 12 vulnerabilities are exposed). On the other hand, the specification-based tool is the most effective one, but requires specifications as inputs. The mutation-based tool is a trade-off between effectiveness and human effort, but its efficiency is very limited (i.e., with only 3 vulnerabilities are exposed per 1000 injections).

Second, we discover that previous studies mostly focus on SUT reactions that have observable symptoms, i.e., failing test cases or printing log messages. The reactions without symptoms, however, are still left unstudied. What is worse, the reactions without symptoms may hide vulnerabilities that have already been triggered, thus affect the effectiveness of CEIT tools. We term such reactions as *indeterminate results*, which account for 21.3% of test results. By further analyzing, we conclude three root causes and their distributions: 1) The SUT silently resolves the errors without any log message (20.2%). 2) The configuration-error generation process is problematic and injects false errors (58.3%). 3) The test suite is inadequate and fails to trigger the errors (21.5%).

Third, inspired by the above gaps, we further discuss potential approaches to improve the CEIT framework: 1) The SUT should print log messages whenever changing configuration values. Adding log statements using two simple code patterns can reduce 91.2% of silent resolutions. 2) The error generation methods should avoid false errors. Simply removing two inefficient mutation rules can decrease 79.1% of false errors. 3) The test suite needs to trigger the injected errors. Automated techniques of generating configuration-oriented test cases can ideally reduce 59.7% of indeterminate results caused by inadequate tests. 4) 81% of test cases in the test suite are redundant w.r.t. code coverages. Randomly sampling 30% test cases only have 5.1-12.1% loss of code coverages, while 38 out of 45 vulnerabilities still could be exposed.

To sum up, in this paper, we build CeitInspector to extensively compare the effectiveness and efficiency of the existing CEIT tools. Moreover, we discuss a previously understudied type of reactions, i.e., indeterminate results. By analyzing such reactions, we further observe three existing limitations of the existing CEIT frameworks and propose corresponding improvements.

2 METHODOLOGY

The workflow of CEIT is to execute the SUT with a configuration error against a test case. This process contains three components: configuration errors, SUTs and test cases. In this section, we clarify the methodologies applied, including generating configuration errors, selecting SUTs, and preparing test cases.

2.1 Generating Configuration Errors

Configuration error generation is a key component of CEIT. The output of the generation is a set of configuration errors which are then used to test the SUT. Ideally, the generated configuration errors should be *effective* in exposing reliability and diagnosability issues, and *efficient* in exposing different types of issues within testing time budget. In general, existing configuration error generation methods can be classified into three categories: random-, mutation- and specification-based methods.

Random. The simplest method of configuration error generation is to generate a random string as an erroneous configuration

[1] Our tool is available at https://github.com/ConfEIT-code/CeitInspector

Table 1: Specification violation rules used in our study. Control dependencies are used as preconditions of error injection.

	Type	Specification	Generation Rules
Basic	Bool/Enum	Options, value set = {"enum1", "enum2", ...}.	Use a value that doesn't belong to set.
	Numeric	Data type, value set = {Integer, Float, Long, ...}.	(1) A type error; (2) out-of-bound values (e.g., INT_MAX+1); (3) an alphabetic string.
		Valid range, range = [MIN, MAX].	Use the values beyond the valid range.
		Unit of measurements, value set = {"ms" "s", ...}.	Use a non-existent unit (e.g., "nounit").
		Value Relationship $(P, V, \diamond)$, $\diamond \in \{<, >, =, \neq, \geq, \leq\}$.	Use invalid value relationship $(P, V, \neg\diamond)$
Semantic	Path	Syntax, ^\/(\w+\/?)+$	Use a random string (e.g., "f5_B0c:146C").
		Path existence, value set = {Yes, No}.	Use a non-existent file path.
		Path type, value set = {Directory, Regular}.	Use a path to a file that violates the file type, e.g., for directory file, use a regular file.
	URL	Syntax, [a-z]+://.*	Use a value that violates the pattern, e.g., http//www.google.com
	IP Address	Syntax, [\d]{1,3}(.[\d]{1,3}){3}	Use a value that violates the pattern, e.g., 255255.255.255
	Port	Valid range, range = [0, 65535].	Use the values beyond the valid range.
		Port should not be occupied.	Use an occupied port number.
	Permission	Data type, value set = {Octet}.	(1) A float-typed number; (2) an alphabetic string.
		Valid range, range = [0000, 7777].	Use the values beyond the valid range.
	Name/ID	Specific value of string.	Use an invalid string.

value. Random is a black-box method. It does not require any knowledge of the configurations or the system under tests. We use only one random string as an erroneous value for every parameter, since different random strings mostly drive the same execution paths and runtime behaviors.

Mutation. Mutation is the most widely-used method for configuration error generation in existing configuration error injection testing tools, including ConfErr [21], ConfInject [8], ConfTest [25], ConfVD [24], and ConfDiagDetector [56]. The key idea is to mutate a given configuration value (often the default value) to generate erroneous values based on predefined mutation rules (e.g. omission, case alteration) to simulate various types of human errors [21]. We implement five mutation rules that used by ConfDiagDetector [56] (the state-of-the-art mutation-based configuration error injection testing tool), which represent mutation rules in mutation-based tools [8, 21, 24, 25, 56]. The five rules are 1) delete the existing value (e.g., "XML"→""), 2) randomly select values of the same data type from a pre-defined pool (e.g., "XML"→"TXT"), 3) randomly select values of a different data type from a pre-defined pool (e.g., "XML"→"123"), 4) randomly inject spelling mistakes (e.g., "XML"→"XLL"), and 5) change the case of text (e.g., "XML"→"xml"). Therefore, we generate *five* erroneous values for each parameter.

Specification violation. Recent work, e.g., Spex-Inj [52] and ConfVD [24], shows that configuration errors can be generated by violating the specifications of configuration parameters, including basic type (e.g., boolean, integer, float, etc), semantic type (file path, IP address, etc), data range, control dependencies, and value relationship. We use the generation rule in ConfVD [24] and Spex-Inj [52] which generate erroneous values based on specifications defined by basic and semantic types. Table 1 lists the error generation rules. For example, if the basic type of a configuration value is NUMERIC with the INTEGER specification, we generate non-integer values (e.g., a string, a float number, and an out-of-bound value). If the semantic type is a PATH with the specifications of "an existent regular file," we generate a non-existent file, a file with insufficient access permissions, and a special file (e.g., a directory).

We collect the specifications for each configuration parameter from both document (user manuals) and source code. We start with user manuals — we find manual pages often include structured descriptions from which specifications can be extracted (refer to [1–6]). On the other hand, we find that many important fine-grained specifications are not well documented, mainly value range of NUMERIC and PATH-related specifications (as "everything is a file" on UNIX-like systems). Therefore, we also inspect the source code to collect value ranges and fine-grained PATH specifications (e.g., whether file existence is required).

Note that the specifications we collected including both control dependencies and value relationships between multiple parameters defined in [52]. We term P and Q for different parameters, and V is a possible value of P. A control dependency means the usage of one parameter depends on another parameter, and can be represented in the form of $(P, V, \diamond) \mapsto Q$. It means Q is only used when $P \diamond V$ returns `true` where $\diamond$ denotes an operator such as $=, \neq, \leq, \geq$, etc. For example, in PostgreSQL, we find the dependency $(\texttt{logging\_collector}, \texttt{on}, =) \mapsto \texttt{log\_directory}$. When injecting errors upon Q, we will also set P to satisfy the condition of $P \diamond V$. On the other hand, a value relationship means the value of one parameter should satisfy certain constraints, and can be represented in the form of $(P, V, \diamond)$, e.g., $(\texttt{max\_connections}, \texttt{max\_wal\_senders}, \geq)$. In this case, we will violate the relationship by setting $(P, Q, \neg\diamond)$.

2.2 Selecting Target Systems

We study six large open-source software systems, as listed in Table 2. The systems are mature, widely-deployed, and representative across a number of software domains. Similar to prior studies [8, 21, 24, 25, 51, 52, 56], we focus on these SUTs for their high reliability and diagnosability requirements. The configuration design and implementation of the studied software systems represent the state-of-the-art of today's systems design.

All the studied systems expose a large number of configuration parameters. We focus on the configuration parameters of the core modules in studied systems instead of all configuration parameters, based on our time and resource budget. For example, in Apache HTTPD, we focus on all the configuration parameters of the Core and MPM modules. In MySQL, we focus on the configuration of two main storage modules, InnoDB and MyISAM. We only study configurations that can be set in configuration files [32]. The second and third columns of Table 2 show the numbers of parameters and the numbers of tested parameters in each SUT.

2.3 Preparing Test Cases

CEIT requires *system test suites* to exercise a SUT and examine the SUT's external behavior from the perspective of the system users.

Table 2: The SUTs studied in this paper.

SUT	# Parameters	# Tested para.	# Test cases	Duration
HTTPD	669	116	1260	5 mins
NGINX	551	277	1451	2 mins
MySQL	461	114	720	25 mins
PGSQL	275	232	178	40 secs
Squid	424	319	6	10 secs
VSFTPD	125	108	4	40 secs

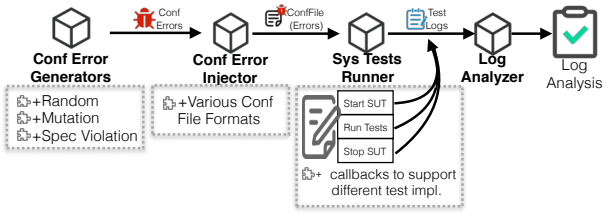


Figure 1: CeitInspector modules and their extensibility.

We use the same system tests of the studied systems described in prior work [21, 24, 25, 51, 52].

To achieve comprehensive testing, we look for more fine-grained system test suites. In the six target systems, HTTPD, NGINX, MySQL, and PostgreSQL have comprehensive, well-designed system test suites, while Squid and VSFTPD do not provide system test suites [2] and none of the prior work share their test code on those two systems. For the former, we use the official test suites and integrate them in CeitInspector; for the latter, we build new test suites. To make sure our tests are reasonable and systematic, we refer to the manuals or guidebooks of Squid and VSFTPD. The last two columns of Table 2 show the numbers of test cases in each test suite and the testing time required to execute the test suite.

3 DESIGN OF CEITINSPECTOR

We design an automated testing framework, i.e., CeitInspector, to evaluate the abilities of different CEIT tools on exposing vulnerabilities. Figure 1 illustrates the components of CeitInspector. Our aim is to build CeitInspector as an extensive framework that can be quickly and easily adapted to different software systems and integrated with various configuration error injection testing techniques. Thus, we follow a modularized style in implementing where each component in CeitInspectorcan be replaced with alternatives. Furthermore, CeitInspector provides a number of abstractions to tackle the heterogeneity and complexity of real-world software systems, in terms of configuration file formats, operation scripts, and system test frameworks. The dotted rectangles in Figure 1 show the replaceable components.

Configuration error generators. We implement there configuration error generation methods described in §2.1: 1) For random, CeitInspector returns a random string as an erroneous value for any configuration parameter. 2) For mutation, CeitInspector implements all mutation rules presented in the prior work [56]. 3) For specification violation, CeitInspector implements the error generation rules defined in in ConfVD [24] and Spex-Inj [52]. For future integration, we provide interfaces for developers to implement other generators.

Configuration error injector. We build a versatile configuration error injector based on Augeas [12], which supports a large number of configuration file formats and can be easily extended to support other file formats. CeitInspector uses Augeas to parse and transform the configuration files into tree-based intermediate representations. CeitInspector injects generated configuration errors by modifying the corresponding tree nodes, and finally transforms the modified trees back to configuration files.

[2]We confirmed with the developers through mailing lists.

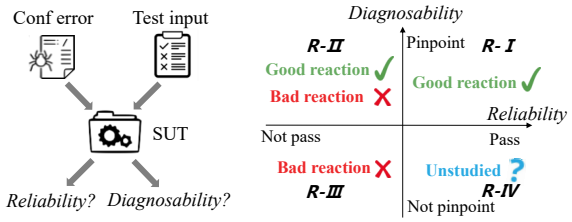
System tests runner. This component runs the SUT with each configuration error against each test case, and collects the corresponding logs (including both application and system-level logs). The key challenge comes from the heterogeneity and complexity of the test suites of different software systems. Here, we adopt the callback mechanism to enable developers to instrument the test suite (i.e., the rightmost dotted rectangle in Figure 1). The callbacks inform CeitInspector about different stages when the test suite is running (e.g., before and after each test case), so that CeitInspector can take different actions accordingly (e.g., checking the health of SUT before each test and collecting logs after each test).

Log analyzer. This component determines the diagnosability of the logs collected above. As long as the logs contain the injected configuration errors (either configuration parameters or erroneous values), CeitInspector assumes they are adequate for error diagnosis. This standard is widely-used in the existing research [24, 52]. CeitInspector implements the analysis based on Whoosh [49], which supports tokenization, stop-word filtering, lowercase/camelcase filtering, and stemming. For parameters, CeitInspector does not require an exact match. As long as the stemmed, tokenized words of the parameter name are included in the stemmed, tokenized words of a log message, CeitInspector considers the logs to be adequate. For the injected value, CeitInspector requires an exact match.

4 EMPIRICAL STUDY

We conduct study on three types of CEIT tools including random-, mutation- and specification-based ones. These tools capture not only the unexpected failures (e.g., runtime crashes and hangs), but also inadequate diagnostic messages [52, 56]. For each configuration error injected into the system configuration, we expect the SUT to 1) report the error *precisely* in error messages to help users resolve configuration-related issues, saving the developers from going over the expensive, time-consuming diagnosis cycles [51, 52, 56]; and 2) react to the error *early* — runtime failures in the production environment may lead to severe impacts like service outages and downtime [16, 17, 33, 35, 41]. Thus, when the SUT fails to pinpoint the injected error (i.e., not precisely) or fails at runtime (i.e., not early), we say the CEIT methods detect a *vulnerability*.

Figure 2 (a) shows the basic workflow of the CEIT tools. The SUT runs with a configuration error against a test input. Then, the tools evaluate the reaction of the SUT in terms of reliability (whether the SUT can pass the test) and diagnosability (whether the SUT can pinpoint the error). Thus, the reaction can be classified into four types as shown in Figure 2 (b), including 1) Pass & Pinpoint (*R-I*): the SUT passes the test, and prints log messages pinpointing the configuration error; 2) Not pass & Pinpoint (*R-II*): the SUT does not pass the test, but leaves log messages pinpointing the error; 3) Not



(a) The CEIT workflow (b) Different types of test results

Figure 2: The workflow and test results of CEIT tools.

pass & Not pinpoint (*R-III*): the SUT neither passes the test, nor prints related error messages; 4) Pass & Not pinpoint (*R-IV*): the SUT passes the test without any log message.

In this section, we explore three research questions:

RQ1: How good are the existing CEIT tools in exposing vulnerabilities? The existing CEIT tools can be classified into three types according to their configuration-error generation methods, i.e., random, mutation and specification violation. We use two metrics to investigate the abilities of the CEIT tools in exposing vulnerabilities: 1) *Effectiveness*: the total vulnerabilities that a CEIT tool could find for each SUT. 2) *Efficiency*: the averaged vulnerabilities found by a CEIT tool per 1000 injected configuration errors.

RQ2: Despite all previous analysis on the exposed vulnerabilities, are there still cases left unstudied? If yes, do they matter? For *R-IV* reactions, although the SUT passes the test, the injected configuration error may be ticking time bombs in the system and lead to severe damage [50]. The reactions are indeterminate, since the problem may be in any component of Figure 2 (a): 1) the injected configuration error may be a valid value; 2) the test input may not trigger the configuration error; 3) the SUT may correct the configuration error silently. These reactions, however, are left unstudied in existing literature [21, 24, 25, 51, 52]. To fill this gap, we classify these indeterminate reactions into three types according to their root causes, then investigate their symptoms and impacts.

RQ3: Are there potential approaches to further improve the CEIT framework? Given that the findings we have in RQ1 and RQ2, we are able to improve the CEIT framework from three aspects, including error generation methods, SUTs, and test cases.

4.1 RQ1: How good are the existing CEIT tools in exposing vulnerabilities?

We investigated CEIT tools using three types of error generation methods (in §2.1) in six SUTs (in §2.2). We manually collected 1166 parameters from core modules of the SUTs, and injected 1166, 5830, and 3006 configuration errors using random, mutation, and specification violation, respectively.

4.1.1 Effectiveness. For each configuration error, the reaction of the SUT is classified into four types as shown in Figure 2 (b). Table 3 summaries the detailed test results. For example, 64, 920, 11 and 171 (the last line in the random sub-table) configuration errors injected by the random method result in *R-I*, *R-II*, *R-III*, and *R-IV* reactions, respectively.

For *R-I* cases, the SUT usually corrects the configuration error to a valid value. This is a good reaction, since the configuration error is well handled, meaning the CEIT methods do not expose any vulnerability.

Table 3: Overall results of different CEIT tools.

Err. Injection		SUT Reaction†					Results	
SUT	Conf. errors	*R-I* Good	*R-II* Good	*R-II* Bad	*R-III* Bad	*R-IV* Ignored	#Vul.	#Vul./ 1k err
Random								
HTTPD	116	4	39	1	0	72	1	9
NGINX	277	1	255	0	5	16	5	18
MySQL	114	34	79	0	1	0	1	9
PGSQL	232	0	225	0	0	7	0	0
Squid	319	25	248	0	0	46	0	0
Vsftpd	108	0	73	0	5	30	5	46
Total	1166	64	919	1	11	171	12	10
Mutation								
HTTPD	580	8	252	6	0	314	2	3
NGINX	1385	4	1112	1	16	252	5	4
MySQL	570	181	280	0	8	101	1	2
PGSQL	1160	2	917	0	4	237	2	2
Squid	1595	143	1008	0	0	444	0	0
Vsftpd	540	0	241	0	21	278	5	9
Total	5830	338	3810	7	49	1626	15	3
Specification Violation								
HTTPD	222	13	85	2	0	122	2	9
NGINX	772	1	758	0	8	5	6	8
MySQL	425	129	258	0	3	35	1	2
PGSQL	695	1	676	0	4	14	3	4
Squid	710	48	598	0	4	60	4	6
Vsftpd	182	0	73	0	13	96	7	38
Total	3006	192	2448	2	32	332	23	8
Mixed Method								
HTTPD	131	4	47	2	0	78	2	15
NGINX	280	1	255	0	7	17	5	18
MySQL	118	33	82	0	3	0	1	8
PGSQL	250	1	230	0	4	15	3	12
Squid	389	31	287	0	2	69	2	5
Vsftpd	134	0	73	0	11	50	6	47
Total	1302	70	974	2	27	229	19	15

† *R-I*: pass & pinpoint; *R-II*: not-pass & pinpoint; *R-III*: not-pass & not-pinpoint; *R-IV*: pass & not-pinpoint.

For *R-II* cases, the SUT does not pass the test but has pinpointing logs. This happens in the following situations: 1) *Early termination*: the SUT refuses to start; 2) *Runtime termination*: the SUT crashes, hangs or throws exceptions; 3) *Runtime misbehavior*: the SUT dysfunctions or outputs wrong results. For early termination, the CEIT tools consider the SUT has a good reaction, since it prevents runtime failures, and the pinpointing logs could help to diagnose the errors. While runtime termination and misbehavior are considered as bad reactions (or vulnerabilities), even with reasonable log messages [50]. Taking a HTTPD case as an example of a bad reaction, the parameter `LimitRequestFiledSize` limits HTTPD request field size, whose valid range is [0, `INT_MAX`]. HTTPD uses `atoi` to parse the value, which could translate a randomly-generated string into 0. During the test, all HTTP requests are denied as their field size is positive. In this example, a randomly-generated configuration error may trigger a runtime failure in HTTPD. The CEIT tools consider HTTPD has a bad reaction w.r.t. the error, although HTTPD prints log messages.

For *R-III* cases, the SUT also does not pass the test. All the three situations (i.e., early termination, runtime termination and misbehavior) are considered as vulnerabilities, as the SUT cannot pinpoint

either the misconfigured parameter's name/value or its location when configuration errors occur [52].

For *R-IV* cases, the SUT has no reactions w.r.t. the injected errors. The CEIT tools are hard to determine if there are vulnerabilities or not, and thus ignore this type.

The three CEIT tools reported 12, 56 and 34 bad reactions in total (the fifth and sixth columns). However, we found 16 bad reactions reported by the mutation-based tool are false positives. Taking NGINX as an example, The mutation tool uses the case-alteration rule to change the value "off" into "OFF", which also is a valid value for case-insensitive parameter `fastcgi_keep_conn`. When using "off/OFF", the test `fastcgi.t` will fail since the parameter `fastcgi_keep_conn` is not turned on. This failure is not caused by the configuration errors, instead, this is an expected behavior when the feature is turned off.

Therefore, the three tools exposed vulnerabilities for 12, 40 (56 - 16) and 34 injected configuration errors, which involve 12, 15, 23 (the second last column) configuration parameters since one parameter may have multiple injected errors. The detailed numbers are shown in the second last column of Table 3. Specification violation exposes the most vulnerabilities among the generation methods. The reason is that a significant percentage of the bad reactions can only be exposed by semantic errors, i.e., errors generated based on the semantics of configuration parameters (e.g., file path, URI, IP address, etc). Taking file path as an example, random and mutation based methods can only generate nonexistent file paths, while many bad reactions are triggered by file paths that are existent but have wrong types (e.g., special files like directories and devices) and wrong permissions. Such errors may pass basic checking such as syntactic and file existence check most of the time.

4.1.2 Efficiency. According to Table 3, the CEIT tools are able to expose 10, 3 and 8 (the last column) vulnerabilities per 1000 injections using random, mutation, and specification violation, respectively. Random is the most efficient method, given that we only generate one random value for each parameter. It proves that some SUTs have poor syntactic checking code that even simple random value can raise unexpected system failures. For example, the parameter `proxy_store` should be a directory path used by NGINX, which does not check validity of the parameter at startup time. When a test tries to use the parameter, a randomly-generated configuration error may raise exceptions, leading to runtime misbehavior of NGINX.

As for efficiency per fixed time-budget, it include two parts: preparing efficiency (e.g., collecting specifications) and testing efficiency. The preparing phase is manually performed by testers. In our study, the specification and mutation techniques require about 27 and 7 person-hours to expose 32 and 15 vulnerabilities, respectively. The preparing efficiency is thus around 1.2 and 2.1 vulnerability/hour. Mutation method only requires parameter type and random method does not require preparation. The testing phase is automatic. The three methods can expose 10, 3 and 8 vulnerabilities per 1000 injections, which take about 21.4 hours on average. Therefore, the testing efficiency is about 0.47, 0.14 and 0.37 vulnerability/hour using random, mutation, and specification violation, respectively.

4.1.3 Summary. The three error generation methods wield different trade-offs among effectiveness, efficiency, and human effort.

Finding 1: Random is the simplest and the most efficient method with limited effectiveness. At one end of the spectrum of the trade-offs, random is the simplest generation method, and achieves the highest efficiency. It does not need to understand anything about the SUT. However, it is hard to generate sophisticated errors. Take file-path parameter as an example, random is hard to generate errors regarding file type and existence. Among all the test results, random exposes 12 vulnerabilities from 1166 injections, averagely 10 per 1000 injections.

Finding 2: Specification is the most effective method with moderate efficiency and great human effort. At the other end of the spectrum, specification violation takes system-specific knowledge to generate sophisticated errors, as shown in Table 1. This input requires great human effort. Meanwhile, it exposes the most effectiveness test results and achieves relatively high efficiency. With specification violation, specification exposes 23 vulnerabilities from 3006 injections, averagely 8 per 1000 injections.

Finding 3: Mutation has a trade-off between effectiveness and human effort, but its efficiency is very limited. Mutation is in the middle of the spectrum. It uses heuristics to generate different errors in order to achieve more effective test results than random, but does not attempt to understand configuration specifications. The mutation method can be viewed as a compromise between specification- and random-based methods. A smarter mutation rule is more like to mutate the input according to specifications, while a less intelligent mutation rule behaves like randomly mutating. However, mutation leads to a large number of indeterminate reactions (type 4 in Table 3), which significantly decrease the efficiency. Using the five state-of-the-art mutation rules, mutation exposes 15 vulnerabilities from 5830 injections, averagely 3 per 1000 injections.

Implication: The three tools show different advantages. The specification-based tool is effective to generate sophisticated errors, while the random-based tool is efficient in exposing shallow vulnerabilities. In the regard, we consider combining different error generation methods. We use the most effective method (i.e., specification violation) to test the parameters with semantic types (e.g., `path` in Table 1), which have much more constraints compared to the basic types. Meanwhile, we use the most efficient method (i.e., random) to handle the parameters with basic types (e.g., `numeric` in Table 1).

The mixed method could be more balanced between effectiveness and efficiency with moderate human effort. Compared to specification violation, the mixed method only needs to obtain constraints from 202 of 1166 parameters, and finds 19 vulnerabilities from 1302 injections (15 per 1000 injections) as shown in Table 3.

4.2 RQ2: Despite all previous analysis on the exposed vulnerabilities, are there still cases left unstudied? If yes, do they matter?

The existing research mainly focused on the SUT reactions that have observable symptoms, i.e., failing test cases or printing log messages. The reactions without symptoms, however, are still left unstudied, since the root causes of these reactions are indeterminate.

Table 4: The overall statistics of indeterminate test results.

Error generation Methods	Silent Resolutions		False Errors				Inadequate Tests		Total
	Value correction	Type casting	String	Case alt.	Value sel.	Dependency	Conditions	Oracles	
Random	6 (3.5%)	41 (24.0%)	52 (30.4%)	0	0	8 (4.7%)	46 (26.9%)	18 (10.5%)	171
Mutation	21 (1.3%)	157 (9.7%)	185 (11.4%)	542 (33.3%)	424 (26.1%)	30 (1.8%)	195 (12.0%)	72 (4.4%)	1626
Specification Violation	18 (5.4%)	188 (56.6%)	0	0	0	0	105 (31.6%)	21 (6.3%)	332
Total	431 (20.2%)		1241 (58.3%)				457 (21.5%)		2129

We manually analyze the test results and source code of these indeterminate reactions, and find the root causes can be classified into three situations:

- **Silent resolutions.** The erroneous value is corrected by the SUT without any log message for informing the users.
- **False errors.** The injected error is in fact not erroneous due to the limitations of error generation techniques.
- **Inadequate tests.** The test suites are not effective in exposing the destructive reactions regarding the injected errors.

Table 4 shows the breakdowns of the root causes using the three existing CEIT tools. We will first discuss the three root causes, and then conclude our findings in the end.

4.2.1 Silent Resolutions. Silent resolutions refer to the *undesired* behaviors that the SUT corrects or ignores the erroneous values, without informing users with explicit log messages. We find the SUT may silently resolve the errors in two ways:

Out-of-bound value correction. The injected error value may be beyond the value range required by the parameter. In this case, the SUT may intentionally change the value into the minimal, maximal, or default value, without informing users about the change. For example, in PostgreSQL, the minimal value of the parameter `autovacuum_work_mem` is 1024. When the injected value is smaller than 1024, PostgreSQL will correct it to 1024 without any log message. In §1, we give the code snippet example.

Unsafe type casting. The SUT tries to change the value type, but unintentionally changes the value at the same time when using some type casting operations. These operations are unsafe since they may change users' intentions and introduce unexpected behaviors. For example, HTTPD, Squid, and VSFTPD parse string values into numeric values using `atoi`, which is known to be unsafe and provides no error checking. When users input an error value '10O' (misspelling the last number 0 into the letter O), the SUTs will translate the value into 10, which is different to the user intention. Another example is MySQL, which changes the parameter `innodb_limit_optimistic_insert_debug` (the valid range is [0, UINT_MAX]) from `unsigned long long` into `unsigned integer`. For values larger than `UINT_MAX`, the parsed value is undefined and can accidentally fall into valid ranges.

4.2.2 False Errors. False errors mean the injected errors values are in fact correct values, indicating the error-generation process is problematic. In the case of random, not all the random strings are necessarily erroneous — a random string could be a valid text, file name, or identity. In the case of mutation, not all the mutation rules can necessarily generate errors — a case-insensitive SUT can accept the values mutated by the case-alternation rule. We find false errors may occur under four situations:

Arbitrary strings. Some parameters can take arbitrary strings, such as `err_html_text` in Squid which is used as "*text to include in error messages*". For those parameters, any strings generated by random or mutation is a false error. Even for strings with constraints (e.g., file path), random and mutation could also lead to false errors. For example, a random string for a path type parameter could lead to false error, the SUT may 1) interprets a string without a slash (`/`) as a file name in the current working directory, and 2) creates a new file with a random string.

Case sensitivity. One rule commonly used in mutation-based error generation is case alternation [21, 56]. This rule is included in the five rules of ConfDiagDetector [56] and is adopted by us (§2.1). However, we find that case alternation leads to almost half (47.1%) of false errors in mutation-based test. This is because the SUTs are case insensitive to the configuration values. In fact, all the SUTs in our study are case insensitive for boolean and enumerative typed configuration values. Therefore, case alternation as a mutation rule should not be generally applied.

Value selection. Besides case sensitivity, we also find the mutation rule, i.e., random selection in the predefined pool with same type (e.g., "XML"→"TXT"), leads to a large number (36.8%) of false errors in mutation-baed testing. This is because many random selected values fall into the valid data range, especially for numeric parameters. Therefore, this rule as a mutation rule should not be generally applied either. In comparison, another rule, using random value with different data type, causes much less false errors.

Control dependency. For a control dependency $(P, V, \diamond) \mapsto Q$, Q is only used when $P \diamond V$ returns `true` (please refer to §2.1). Since random and mutation does not understand such control dependencies, they simply inject errors to Q without setting P to satisfy $P \diamond V$. If $P \diamond V$ does not satisfy by default, whatever errors injected to Q would lead to no effect. Specification violation does not suffer from this problem, because control dependency is one type of specification — in order to test the target parameter Q, CeitInspector will set P to satisfy $P \diamond V$ as the pre-condition for testing Q.

4.2.3 Inadequate Tests. Inadequate tests mean the test suite shipped with the SUT does not expose the injected errors. This happens in two scenarios: the test inputs fail to trigger the errors, or the test oracles fail to capture the errors.

Missing triggering conditions. The majority of indeterminate results caused by test inadequacy are caused by missing triggering conditions. In such cases, the tests fail to drive the execution to reach the statements that consume the erroneous configuration values. We inspect these cases and find the following patterns:

Specific workload. Some parameters are used when the SUT performs specific operations. For example, HTTPD uses `AddHandler` to assign handler to the specific file. Only if the file is fetched by

the user, the handler will be invoked to handle the request. The configuration can hardly be tested if the tests do not access the file.

Failure events. Some parameters are only used in fault tolerance or error handling code only executed upon failure events (e.g., protection faults and runtime exceptions). Without an effective mechanisms to emulate those events in the tests, such parameters cannot be tested.

Specific environment. Some parameters are only used in special environment, such as browser types. The test suite cannot simulate different browsers, the configuration values are not exercised.

Disabled macro. Some parameters do not take effect due to disabled macros at the compilation time. The following code snippet shows an example from the PostgreSQL: only if the macro `OPENSSL_NO_ECDH` is not defined, the parameter `ssl_ecdh_curve` (store in the variable `SSLECDHCurve`) will take effect.

```
#ifndef OPENSSL_NO_ECDH
  nid = OBJ_sn2nid( SSLECDHCurve );
  ecdh = EC_KEY_new_by_curve_name(nid);
#endif
```

Missing oracles. Many indeterminate results come from the fact that the test cases do not have the oracles to capture the symptoms, even though the test inputs have triggered the injected errors. One such example is the injection testing for `AddDefaultCharset` in HTTPD, which is exposed by the test execution and cause a HTTP response with "`charset=error`". HTTPD's test does not check the charset of the response and passes the test. As a comparison, NGINX has a similar parameter `charset_type`; NGINX's test suite checks the validity of the charset in HTTP responses, and thus do not have indeterminate results.

4.2.4 Summary. Despite all the indeterminate reactions have the same symptoms w.r.t. the test results and logs, they might have extremely different implications.

Finding 4: 20.2% of indeterminate reactions are caused by silent resolutions, which may change user intentions without notifications. Silent resolutions commonly leads to user confusion and frustration due to mismatches between system behavior and user intents—the systems ignore or correct users' configurations without any feedback or notifications. In principle, silent resolutions should be seen as vulnerability and CEIT tools should capture silent resolutions to enhance the system diagnosability [52, 56]. However, in the existing CEIT tools, without manual inspection, these vulnerabilities will never be exposed since they hide inside the indeterminate results. In Table 4, silent resolutions contribute to 47 (6+41), 178 (21+157), 206 (18+188) indeterminate results in three existing CEIT tools.

Finding 5: 58.3% of indeterminate reactions are caused by false errors, which lead to useless tests and affect the efficiency of CEIT. False errors fundamentally break the principle of configuration error injection testing which exercises the system behavior upon erroneous values. False errors not only waste testing time, but also lead to indeterminate results that has to be manually filtered out. CEIT tools should avoid generating false errors. In our study, false errors lead to 60 and 1181 indeterminate results in random and mutation. Specification violation can avoid false errors given the comprehensive understanding of constraints.

Table 5: The test results of prototype logging tool.

Software	Silent Resolutions	# of the improved	Ratio
HTTPD	169	166	98.2%
NGINX	0	0	NULL
MySQL	62	60	96.8%
PostgreSQL	1	0	0.0%
Squid	40	26	65.0%
VSFTPD	159	141	88.7%
Total	431	393	91.2%

Finding 6: 21.5% of indeterminate reactions are caused by inadequate tests, which may lead to false negatives and affect the effectiveness of CEIT. CEIT tools depend on existing system tests to expose vulnerabilities. Prior works [21, 56] use a small number of tests which are unlikely adequate. Even official system test suites suggested in [52] do not have sufficient coverage. We use gcov to measure the statement coverage of the SUTs under the official suites and find the coverage is merely 29%–68% across the studied systems. A future direction is to investigate automated or semi-automated techniques for selecting or generating test cases for configuration error injection testing based on how each test exercises configuration parameters. In our study, inadequate tests result in 64, 267, and 126 indeterminate results in three CEIT tools.

4.3 RQ3: Are there potential approaches to further improve the CEIT framework?

In § 4.1, we find up to 21.3% test results of the existing CEIT tools are indeterminate and ignored by the prior research work. In § 4.2, we conduct a comprehensive study addressing the indeterminate results, and found their root causes may come from error generation methods, SUTs and test cases. In this section, we explore possible remedies of improving the CEIT framework from three aspects guided by the above findings.

4.3.1 Improving SUTs. Silent resolutions contribute up to 20.2% indeterminate reactions (as shown in Table 4), which are regarded as bad reactions of the SUT. Therefore, improving the SUT could reduce indeterminate reactions generated by the CEIT framework.

Silent resolutions occur in two situations, i.e., out-of-bound value correction and unsafe type casting, as discussed in §4.2.1. Both the situations change the parameter values without informing users with explicit log messages. In this regard, we consider building a log automation tool to locate the silent resolutions, then add pinpointing logs. Log automation typically has two tasks: *what to log* [54] and *where to log* [19, 54, 58]. For the what-to-log task, the warning logs should pinpoint the name, original value, and new value of the parameter. For the where-to-log task, the log locations are depended on the different situations of silent resolutions:

1) Out-of-bound value correction. This situation usually has an explicit assignment statement, which changes the configuration value. Therefore, the log statement can be placed right after the assignment statement:

```
foo (string para_name...){
  int para_value = read_int_from_conf(para_name);
  int old_value = para_value;
  if is_not_correct(old_value):
     para_value = new_value;
+    printf("%s: changing the value from %d to %d",
```

Table 6: The evaluation of mutation rules.

Rules	Injections	Indeterminate	False Errors	Vulnerabilities
Rule 1	1666	83	33	4
Rule 2	1666	533	432	6
Rule 3	1666	196	93	7
Rule 4	1666	217	120	14
Rule 5	1666	597	503	9
Total	5830	1626	1181	15
Rule 1,3,4	3498	496	246	15

```
+       para_name, old_value, new_value);
}
```

The log tool first uses ConfMapper [57] to identify configuration parameters and their corresponding variables in source code. When a variable is assigned to a new value without any log statement, we add a log statement after the assignment statement.

2) Unsafe type casting. This situation usually uses APIs like `atoi` to change the value types. When the parsing APIs change the value at the same time, there should be log statements:

```
bar (string para_name, ...){
 string para_value_str = read_str_from_conf(para_name);
 int para_value_int = atoi(para_value_str);
+ if string(para_value_int) != para_value_str:
+     printf("%s: changing the value from %s to %s",
+         para_name, para_value_str, string(para_value_int));
}
```

We propose a simple oracle to determine if the parameter value (e.g., `para_value_str`) is changed: when transferring the return value (e.g., `para_value_int`) back to its original type (e.g., `string`), the output value (e.g., `string(para_value_int)`) does not equal to the original value (e.g., `para_value_str`). Using this oracle, the log tool first detects whether the SUT uses unsafe parsing APIs like `atoi, atof, atol`, etc. If yes, the tool transfers the return value back into its original type. When the transferred value is inconsistent with the original value, a log statement will be placed. For example, `atoi` will transfer the string "10O" into the integer 10. If converting the integer 10 back to string, the tool finds "10" is not equal to "10O", and thus adds a log statement.

With the help of this tool, 91.2% of silent resolutions can be transferred into pinpointing reactions. The detailed result is shown in Table 5. This result suggests the logging tool can fix most silent resolutions by checking two simple code patterns. Meanwhile, there are still some cases that are too complex to be handled. For example, some silent corrections happen when one parameter does not satisfy the constraint with another parameter. Our prototype tool cannot handle the cases involving multiple parameters.

4.3.2 Improving Configuration Error Generation. Among three generation methods, the mutation method leads to the most (27.9%) indeterminate results, and 72.6% of them are caused by false errors, as shown in Table 4. These false errors significantly impede the efficiency of CEIT. For example, in MySQL, the mutation method can only trigger one vulnerability with 570 injections, each of which requires 25 minutes. It means the mutation method can find only one vulnerability using nine days. In this regard, we evaluate the effectiveness and efficiency of each mutation rule respectively, and explore possible improvements. All the mutation rules we evaluate are widely-used in the previous work [21, 56].

The evaluation results are shown in Table 6. The second (random selection within a pre-defined pool) and fifth (case alteration) rules introduce 533 and 597 indeterminate reactions, while the rest only have 83-217 ones. The reason is that these two rules are more likely to generate valid values: the second rule often chooses values in the valid range; the fifth rule is not effective in SUT using case-insensitive parsing methods like `strcasecmp`. For example, the second rule may generate an integer 1024 for a numeric parameter, but 1024 is in the valid range. For the fifth rule, all the SUT use `strcasecmp` to parse boolean or enum parameters. Therefore, nearly all the case alteration injections lead to false errors. Besides, these two rules do not find new vulnerabilities compared to the rest three rules.

Here, we use the first, third and forth rules when applying the mutation method. We compare the original mutation method and the mutation method excluding these two rules. The new mutation method does not miss any vulnerability, while the required injections decrease from 5830 to 3498. And false errors decrease from 20.3% (1181/5830) to 7.0% (246/3498). Considering the mutation method needs no system-specific knowledge, the new mutation method could still find 25% more vulnerabilities than the random method but in a more efficient way compared to the original one. In practice, when new mutation rules are introduced, we suggest these rules should avoid vast amount of false errors.

4.3.3 Improving Test Suites. The test suite of the SUT may significantly affect CEIT framework. On one hand, the test suite may lack test cases to trigger the configuration errors, and downgrade the effectiveness. On the other hand, the test suite may have redundant test cases, and downgrade the efficiency. Therefore, the test suite can be improved from two aspects: removing redundant test cases and generating configuration-oriented test cases.

Generating configuration-oriented test cases. Inadequate test cases contribute up to 21.5% indeterminate reactions, as shown in Table 4. This result suggests many configuration errors are not triggered by the test cases at all. It could be for the reason that CEIT has always been conducted as black-box testing [8, 21, 24, 25, 52, 56], which means it could be inefficient since testers could not decide which part to test. Moreover, current test suite is not designed for CEIT and could not guarantee all the misconfigurations could be triggered. Because current test suites concentrate on the functionalities while CEIT most focus on the error-checking and -handling which might not be tested during the functional tests.

To the opposite, white box testing could allow users to concentrate on testing the parameter-related code. Therefore, generating test cases that can trigger the configuration errors can help reduce indeterminate reactions and probably detect more vulnerabilities. To achieve this, we propose an automated technique of generating configuration-oriented test cases as the future work. The workflow is illustrated in Figure 3. The tool first locates the parameter-related code by using taint analysis, then uses directed fuzzing and symbolic execution methods to generate test inputs that can trigger the parameter-related code. Ideally, the tool could solve the missing trigger conditions excluding disabled macro, which account for 59.7% of inadequate tests.

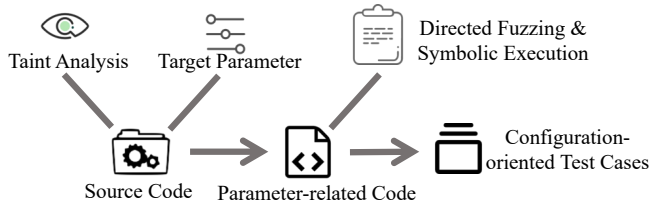


Figure 3: Generating configuration-oriented test cases.

Removing redundant test cases. In general, the purpose of test suites is to test the SUT functionalities as many as possible. Therefore, code coverage is an important metrics to evaluate the quality of a test suite. As a result, developers tend to design more test cases to achieve higher code coverages. When two test cases are redundant (i.e., covering the same code), the overhead is limited since all test cases are executed once automatically. For CEIT, however, thousands of configuration errors should be tested under each test case. Accordingly, the overhead of redundant test cases is magnified thousands of times, thus may significantly affect the CEIT efficiency. Plenty of time is probably wasted when running the code irrelevant to the injected configuration errors.

To remedy this, we evaluate the code coverage of each test case in official test suites, and find that 81% of test cases have no coverage increase. These tests are usually redundant for CEIT. A simple solution to mitigate the problem is to sample the test cases. When sampling 10% test cases, we find the loss of code coverage is 5.49-29.38% compared with using all test cases. When the sample rate is 30%, the code coverage loss is 5.10-12.10%. Also, the impact for the CEIT effectiveness is limited during the sampling process. Our experiment shows that 38 out of 45 vulnerabilities (excluding Squid and VSFTPD since there is no official test suites) can be exposed after 30% sampling. It indicates that test-case sampling can significantly improve the CEIT efficiency with limited loss of effectiveness.

5 DISCUSSION

We use CeitInspector as a vehicle to evaluate the effectiveness and efficiency of CEIT. In this section, we summarize the challenges we encountered when using CeitInspector to apply the testing. The challenges are seldom discussed in existing tools but are critically important to the testing results. We share our solutions which have been integrated in CeitInspector.

Noises in logs. A key challenge in log analysis is to deal with the noises. Specifically, we experience two outstanding patterns of noises which could significantly affect the testing results if not handled appropriately. First, some configuration parameters and values will be printed out in the log messages during the normal execution, without any injected configuration errors. Common cases are `INFO` logs that expose internal system status and actions. In such cases, any injected errors of the related parameters lead to adequate reactions unless the errors crash the system, because there is always a log message containing the related parameter.

CeitInspector filters out those noises by using the log messages of normal execution as baseline. It only considers log messages that do not appear in the baseline logs. This feature requires users to annotate the general log message format in order to filter out execution-specific information such as timestamps and process IDs.

Table 7: Testing time of specification violation. All the tests run in Docker containers on a CentOS 7.7 with 2.5 GHz dual-core CPU and with 16 GB RAM.

Software	Total	Average	Test Suite Runtime
HTTPD	640 mins	2.82 mins	5 mins
NGINX	360 mins	0.47 mins	2 mins
MySQL	2220 mins	5.23 mins	25 mins
PostgreSQL	120 mins	0.17 mins	3 mins
Squid	41 mins	3.68 secs	10 secs
VSFTPD	61 mins	16.40 secs	40 secs

Second, some systems adopt the practice of dumping the entire configuration file content upon failures. Such messages are not helpful for users to diagnose the injected configuration errors even if the root-cause configuration parameters and values are included in the message. Currently, CeitInspector provides an interface to annotate such log messages using regular expressions—the log messages will be ignored in log analyzer.

Compound data types. We find in total 28 parameters from HTTPD, NGINX, and Squid come up with *compound* data types, in which the parameter can be typed differently. For example, `user_expires` in NGINX can either be a numeric value, or an enumerative string $\in$ {`"max"`, `"off"`}. For specification violation, we build support for compound types in CeitInspector which ensures that a generated error violates *all* specifications of all the possible types.

Testing time. One concern of CEIT is the testing cost – the total testing time could be $N \times T$ where N is the number of injected errors and T is the running time of the test suite (the fifth column in Table 2). In practice, we find that the actual testing time is much shorter than $N \times T$ because many injected errors lead to early terminations that skip the entire test suites, or test failures that skip subsequent tests.

Table 7 shows the testing time using specification violation based error generation. The actual test time is 1.77×–17.65× shorter than the projected overall time ($N \times T$).

6 THREATS TO VALIDITY

Our study may suffer threats to the external and internal validity.

Threats to the external validity. The software projects we choose might affect the generalization of our empirical study. To handle this, the projects in our study are all mature and widely-used, with at least 15 years of development and maintenance history. Many bugs that lead to crashing behavior have been already exposed in prior work [21, 52] and fixed by developers. We expect different testing results in new, immature software programs. Specifically, we believe configuration error injection testing can expose more crashing bugs in new codebases, in a similar vein as other modern fuzzing techniques [15, 23, 26, 34, 36, 37, 42, 45]. For now, this study mainly focuses on reaction analysis, instead of crashing behavior.

Threats to the internal validity. We do not use automated approaches to generate specifications, but instead encode specifications manually. We collect configuration specifications manually from document and also from source code as discussed in §2.1. The results are based on human-based specification engineering. Similarly, we manually check the reaction results in §4. For bad

reactions, we check the SUT logs to see whether they are caused by injected configuration errors to identify *vulnerabilities* and *false positives*. For indeterminate reactions, we first compare the parameter's specification and the injected value to see whether it is a *false error*. If not, we inspect the source code and test code to find if it is a *silent resolution* or *inadequate tests*.

Therefore, the human factor may pose a threat to the confidence of our study. To control this threat, two authors separately perform the manual study, and compare their answers to make the results credible. When they diverge, the third author is consulted for additional discussion until consensus is reached.

7 RELATED WORK

Prior work on configuration error injection testingmostly focuses on two main components: configuration error generation and system reaction analysis.

Configuration error generation. In general, existing configuration error generation methods can be classified into the following three categories. The simplest method of configuration error generation is to generate a random string as an erroneous configuration value, known as fuzzing [30, 31] [3]. Random is a black-box method. It does not require any knowledge of the configurations or the system under tests.

Mutation is the most widely-used method for configuration error generation in existing configuration error injection testing tools, including ConfErr [21], ConfInject [8], ConfTest [25], ConfVD [24], and ConfDiagDetector [56]. The key idea is to mutate a given configuration value (often the default value) to generate erroneous values based on predefinned mutation rules (e.g. omission, case alteration) in order to simulate various types of human errors, such as slips, lapses, etc [21].

Recent work, e.g., Spex-Inj [52] and ConfVD [24], shows that configuration errors can be generated by violating the specifications of configuration parameters, including basic type (e.g., boolean, integer, float, etc), semantic type (file path, IP address, etc), data range, control dependencies, and value relationship. For example, if a configuration parameter specifies an integer typed value, a non-integer value can be generated.

Prior studies show that specification violation based error generation is effective in exposing reliability and diagnosability issues. A number of recent work [24, 25, 27] has developed automated techniques to extract or infer configuration specifications from source code [40, 52], documents [27, 38, 47, 48], or field configuration data [43, 44, 55]. Many software companies have the practice of encoding specifications of configuration parameters [14, 18, 29, 46] (mainly for configuration validation)—those specifications can directly be used for configuration error generation.

Reaction Analysis As system reactions to configuration errors are embodied in the console output and/or system logs (`syslog`), automated text-based analysis has been used to identify inadequate diagnostic messages. The standard criteria to evaluate the diagnosability of the message content is that the message is useful for users to diagnose the (injected) configuration error as long as the message content contains the root-cause configuration parameter name or the erroneous value. This criteria is used in most of the existing work [21, 24, 25, 27, 52]. Xu et al. [52] points out that configuration errors could also result in system crashes, hangs and indeterminate termination during the test runs. Such reasons can be captured at the system level (e.g., checking the running process and analyzing `syslog`).

Zhang and Ernst [56] further propose to use natural language processing (NLP) techniques to evaluate the informativeness of the message content by comparing the similarity between log messages and document entry of the configuration parameter.

[3]The term "fuzzing" was originally refer to "generating a stream of random characters to be consumed by a target program [30]." Since then, the concept of fuzzing has been broadened as is not necessarily randomized [28].

8 CONCLUSION

This paper presents an in-depth study on the effectiveness and efficiency of configuration error injection testing to evaluate software reactions upon configuration errors. We design and implement CeitInspector, an extensive and highly-modularized configuration error injection testing tool which can be easily applied to different SUTs and integrated with various configuration error injection techniques. We experiment three major existing configuration error generation methods including random, mutation and specification violation on six mature server applications. We first conduct a comparative study of different error generation methods including their effectiveness, efficiency and cost trade-offs. Then, we reveal the limitations of current CEIT tools through indeterminate test result analysis. Finally, we propose approaches to improve the CEIT framework. This work can benefit future configuration error testing work.

ACKNOWLEDGEMENT

We sincerely thank the anonymous reviewers for insightful suggestions, and Tianyin Xu for his guidance to this work. Also, we thank Yu Jiang, Yanyan Jiang, Jianyan Chen and Lucheng Bao for their feedback and suggestions on this paper and tools. This research was substantially supported by National Key R&D Program of China (Project No. 2017YFB1001802); National Natural Science Foundation of China (Project No. 61872373 and No. 61872375).

REFERENCES

[1] 2020. HTTPD configuration manual page. http://httpd.apache.org/docs/2.4/mod/directives.html.
[2] 2020. MySQL configuration manual page. https://dev.mysql.com/doc/refman/5.6/en/server-option-variable-reference.html.
[3] 2020. Nginx configuration manual page. http://nginx.org/en/docs/dirindex.html.
[4] 2020. PostgreSQL configuration manual page. https://www.postgresql.org/docs/11/bookindex.html.
[5] 2020. Squid configuration manual page. http://www.squid-cache.org/Doc/config/.
[6] 2020. VSFTPD configuration manual page. https://security.appspot.com/vsftpd.html.
[7] George Amvrosiadis and Medha Bhadkamkar. 2016. Getting Back Up: Understanding How Enterprise Data Backups Fail. In *Proceedings of 2016 USENIX Annual Technical Conference (ATC'16)*. Denver, CO.
[8] Fahad A. Arshad, Rebecca J. Krause, and Saurabh Bagchi. 2013. Characterizing Configuration Problems in Java EE Application Servers: An Empirical Study with GlassFish and JBoss. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE'13)*. Pasadena, CA, USA. https://doi.org/10.1109/ISSRE.2013.6698919
[9] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. 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)*. Hollywood, CA, USA. https://doi.org/10.5555/2387880.2387910

[10] Mona Attariyan and Jason Flinn. 2008. Using Causality to Diagnose Configuration Bugs. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC'08)*. Boston, MA, USA. https://doi.org/10.5555/1404014.1404037

[11] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. Vancouver, BC, Canada. https://doi.org/10.5555/1924943.1924960

[12] Augeas. 2018. Augeas - a configuration API. http://augeas.net/

[13] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines (Third Edition)*. Morgan and Claypool Publishers. https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[14] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17), Industrial Track*. https://doi.org/10.1145/3154448.3154453

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)* (Dallas, Texas, USA). https://doi.org/10.1145/3133956.3134020

[16] Jim Gray. 1985. Why Do Computers Stop and What Can Be Done About It? *Tandem Technical Report 85.7* (Jun. 1985). https://doi.org/10.1.1.59.6561

[17] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. Santa Clara, CA. https://doi.org/10.1145/2987550.2987583

[18] Peng Huang, William J. Bolosky, Abhishek Sigh, and Yuanyuan Zhou. 2015. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. Bordeaux, France. https://doi.org/10.1145/2741948.2741963

[19] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71. https://doi.org/10.1109/SANER.2018.8330197

[20] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. 2008. Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. San Jose, CA, USA. https://doi.org/10.1145/1416944.1416946

[21] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. 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)*. Anchorage, AK, USA. https://doi.org/10.1109/DSN.2008.4630084

[22] Stuart Kendrick. 2012. What Takes Us Down? *USENIX ;login:* 37, 5 (Oct. 2012), 37–45. https://www.usenix.org/publications/login/october-2012-volume-37-number-5/what-takes-us-down

[23] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (Montpellier, France). https://doi.org/10.1145/3238147.3238176

[24] Shanshan Li, Wang Li, Xiangke Liao, Shaoliang Peng, Shulin Zhou, Zhouyang Jia, and Teng Wang. 2018. ConfVD: System Reactions Analysis and Evaluation Through Misconfiguration Injection. *IEEE Transactions on Reliability (Early Access)* (Sep. 2018). https://doi.org/10.1109/TR.2018.2865962

[25] Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. 2017. ConfTest: Generating Comprehensive Misconfiguration for System Reaction Ability Evaluation. In *The International Conference*. 88–97. https://doi.org/10.1145/3084226.3084244

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)* (Paderborn, Germany). https://doi.org/10.1145/3106237.3106295

[27] Xiangke Liao, Shulin Zhou, Shanshan Li, Zhouyang Jia, Xiaodong Liu, and Haochen He. 2018. Do You Really Know How to Configure Your Software? Configuration Constraints in Source Code May Help. *IEEE Transactions on Reliability* 67, 3 (Sep. 2018), 832–846. https://doi.org/10.1109/TR.2018.2834419

[28] Valentin J.M. Manés, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, , Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *arXiv:1812.00140* (Apr. 2019). https://doi.org/10.1109/TSE.2019.2946563

[29] Ben Maurer. 2015. Fail at Scale: Reliability in the Face of Rapid Change. *Commun. ACM* 58, 11 (Nov. 2015), 44–49. https://doi.org/10.1145/2838344.2839461

[30] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[31] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. Technical Report 1268. University of Wisconsin-Madison, Computer Sciences Department.

[32] MySQL Parameter. 2019. MySQL parameter from Innodb and myisam. https://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html

[33] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2004. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*. San Francisco, CA, USA. https://doi.org/10.5555/1251254.1251259

[34] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18)* (Pau, France). https://doi.org/10.1145/3167132.3167289

[35] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. 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)*. Seattle, WA, USA. https://doi.org/10.5555/1251460.1251461

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)* (Beijing, China). https://doi.org/10.1145/3293882.3330576

[37] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. In *Proceedings of the ACM on Programming Languages (OOPSLA'19)*. https://doi.org/10.1145/3360600

[38] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. 2015. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)*. https://doi.org/10.14778/2824032.2824079

[39] Ariel Rabkin and Randy Katz. 2011. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. Lawrence, KS, USA. https://doi.org/10.1109/ASE.2011.6100053

[40] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *Proceedings of the 33th International Conference on Software Engineering (ICSE'11)*. Honolulu, HI, USA. https://doi.org/10.1145/1985793.1985812

[41] Ariel Rabkin and Randy Katz. 2013. How Hadoop Clusters Break. *IEEE Software Magazine* 30, 4 (Jul. 2013), 88–94. https://doi.org/10.1109/MS.2012.73

[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS'17)* (San Diego, CA, USA). https://doi.org/10.14722/ndss.2017.23404

[43] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing Configuration File Specifications with Association Rule Learning. In *Proceedings of 2017 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*. https://doi.org/10.1145/3133888

[44] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic Automated Language Learning for Configuration Files. In *28th International Conference on Computer Aided Verification (CAV'16)*. Toronto, Canada. https://doi.org/10.1007/978-3-319-41540-65

[45] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS'16)* (San Diego, CA, USA). https://doi.org/10.14722/ndss.2016.23368

[46] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating System Principles (SOSP'15)*. Monterey, CA, USA. https://doi.org/10.1145/2815400.2815401

[47] Ozan Tuncer, Nilton Bila, Sastry Duri, Canturk Isci, and Ayse K. Coskun. 2018. ConfEx: Towards Automating Software Configuration Analytics in the Cloud. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN-W.2018.00019

[48] Ozan Tuncer, Nilton Bila, Canturk Isci, and Ayse K. Coskun. 2018. *ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud*. Technical Report RC25675 (WAT1803-107). IBM Research.

[49] Whoosh. 2018. Whoosh 2.7.4 documentation. https://whoosh.readthedocs.io/en/latest

[50] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA. https://doi.org/10.

5555/3026877.3026925

[51] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. 2017. How Do System Administrators Resolve Access-Denied Issues in the Real World?. In *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)*. Denver, CO. https://doi.org/10.1145/3025453.3025999

[52] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*. Farmington, PA, USA. https://doi.org/10.1145/2517349.2522727

[53] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. Cascais, Portugal. https://doi.org/10.1145/2043556.2043572

[54] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* 30 (2012), 4. https://doi.org/10.1145/1950365.1950369

[55] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*. Salt Lake City, UT, USA. https://doi.org/10.1145/2644865.2541983

[56] Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. Baltimore, MD, USA. https://doi.org/10.1145/2771783.2771817

[57] Shulin Zhou, Xiaodong Liu, Shanshan Li, Wei Dong, Xiangke Liao, and Yun Xiong. 2016. Confmapper: Automated variable finding for configuration items in source code. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 228–235. https://doi.org/10.1109/QRS-C.2016.35

[58] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proc. of ACM/IEEE ICSE*. https://doi.org/10.5555/2818754.2818807