Studying bugs in the Salt Configuration Management System

Bryan He
Supervisor(s): Thodoris Sotiropoulos, Prof. Dr. ir. Diomidis Spinellis
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Configuration management systems are a class of software used to automate system administrative tasks, one of which is the configuration of software systems. Although the automation is less error-prone than manual configuration done by a human, bugs in the source code can still cause configuration errors. This can result into unwanted consequences for the managed software system, some examples are performance degradation, downtime and security breaches.

Bug studies are conducted on previously reported bugs to find out their characteristics. Such findings help with preventing, detecting and fixing bugs, which ultimately causes software to become more reliable. This paper aims to fill in a research gap in the literature surrounding bug studies conducted on configuration management systems. From Salt, a widely used open-source configuration management system, a data-set of 5,896 bugs with fixes was collected. The main research question answered by this paper is "What are the common patterns can be extracted from bugs found in Salt?" To answer this, 100 bugs were randomly sampled from this data-set to analyze their symptoms, root causes, impact, fixes, system-dependence and triggers.

1 Introduction

Configuration management systems are tools used by system administrators and developers to automate their work. Examples of tasks that get automated are provisioning of infrastructure, maintenance, and configuration of complex software systems. In a file, the user simply describes the desired state for a software system to be in, using declarative language. The usage of a declarative language frees the user from having to provide how-to instructions that need to be performed to reach the wanted state. The configuration management system uses inputted file to devise and perform the step by step operations needed for the software system to reach the specified desired state.

To ensure the availability of services, nowadays many software systems are large in scale and highly distributed and thus consist of many nodes that need to be managed [1]. Handing over the system administrative tasks to configuration management systems, greatly decreases the occurrence of costly configuration bugs inadvertently introduced to a software system by human errors. However, even with the automation provided by configuration management systems, software systems still suffer from incorrect functionality, decreased performance [2], severe service failures [3], downtime and security breaches [1]. This is largely caused by functional bugs present in the source code of the configuration management system. As a result of these bugs, desired states specified by the user are incorrectly translated into wrong operations which are applied to the managed software system, rendering it misconfigured.

Studying the functional bugs found in a software systems contributes to the improvement of its software quality. By analyzing bugs and their fixes, fundamental characteristics can be discovered. Examples of such characteristics are the bugs' root cause, impact and how it was triggered. Additionally, symptoms are useful for detecting a bug's presence. With the insights gained from bug studies, techniques and tools can be devised to detect and fix bugs as they arise [4].

There is extensive study on the consequences of misconfigurations. Oppenheimer et al. [3] studied operation errors of three large Internet and found that more than 50% of those were caused by misconfigurations. The impact of configuration errors on DNS robustness was studied extensively by Pappas et al. [5]. Xu and Zhou conducted a survey that provides a structured overview of the systems that help prevent configuration errors, one of which are configuration management systems [1]. Moreover, there is prior research work which aims to further improve configuration management systems [6]. However, to the best of my knowledge, there are no characteristic studies on the functional bugs previously reported in the configuration management system. Findings of such a bug study can be leveraged by development teams of configuration management system to prevent and detect bugs. One thing to strive for when doing bug studies on any software system, is to discover representative insights which can be extended to other software systems of the same category. This study focused on bugs found in the Salt configuration management system and was done alongside peer researchers who studied bugs from three other configuration management systems which are Ansible, Moby and Puppet [7, 8, 9]. The findings of these bug studies are then compared to discover shared commonalities.

The goal of this bug study is to identify common patterns in bugs found in the Salt configuration management system. To achieve this goal the following research questions will have to be answered:

1. What are **main symptoms** of previously reported bugs in Salt?
2. What are the **root causes** of known bugs in Salt?
3. What is the **impact** of known bugs found in Salt?
4. What are the **fixes** of known bugs found in Salt?
5. Are the bugs found in Salt **system-dependent**[1]?
6. What are the **triggers** of the known bugs found in Salt?

Section 2 presents the methodology followed to collect the bugs analyzed in this bug study. Subsequently the findings that resulted from the bug analysis of a random sample of bugs are presented in section 3. In section 4, a discussion of the results and their implications can be found. Prior research work related to this paper are presented in section 5. Lastly, reflections about the reproducibility and integrity of this study and conclusions are found in sections 6 and 7, respectively.

2 Methodology

Before bugs can be analyzed, they need to be collected and processed so that only relevant bugs are considered. Subsection 2.1 outlines the procedure followed to gather previously reported bugs. Subsequently, subsection 2.2 describes how randomly sampled bugs are analyzed.

2.1 Bug Collection

The Salt's developers make use of a GitHub repository to host the code of their project [10]. The GitHub issue tracker of that repository is used for reporting bugs and submit pull requests to fix bugs. This is the source from which bugs are collected.

The collection of bug data is a two-step process, consisting of bug fetching and post-filtering, and it closely follows the procedure outlined in the work done in studying typing-related bugs in JVM compilers by Chaliasos et al. [4]. A summary of the statistics describing the bug collection procedure can be found in Table 1. Bug fetching and post-filtering are elaborated in the following paragraphs.

Total Issues	Earliest	Most Recent	Bug Fetching	Post-Filtering	Has Test Cases
24,509	30 July 2012	29 April 2022	8,920	5,896	2467

Table 1: The total number of issues in the Salt repository, the date of the earliest and most recent issues considered in this bug study. The number of issues after fetching the bugs and post-filtering them. Lastly, the number of issues with a potential fix and test cases.

Bug Fetching. The first step is to fetch relevant issues from the Salt's GitHub repository using the GitHub REST API [11]. The following requirements make up the selection criteria that was imposed

[1]A bug is considered system-dependent if it only occurs on a specific platform but not others. For instance a bug that only manifests on Windows systems but not Linux.

on each issue fetched, with the aim of ensuring the issues are in fact bug reports and have enough information to be analyzed:

- The issue must have the label "Bug" assigned to it. The development team assigns descriptive labels to issues to keep them organized by category. The "Bug" label is used to mark an issue as a bug report. As a result of applying this filter, issues related to feature requests are thus not collected.
- to ensure a bug report has enough information it's status must be closed. This is because bug reports with an open status are either in the process of being confirmed by the development team or in the process of being fixed. Therefore, those bug reports do not have information about how it was fixed, which is crucial information needed to answer RQ4.
- Bug reports that have the labels "won't-fix", "Duplicate" and "Documentation" are excluded. Bug reports with the "won't-fix" labels are excluded for the same reasons why open issues are excluded. Bug reports labeled "Documentation" do not affect Salt's source code. Lastly, duplicate issues are closed and not linked to a fix.

At the time of writing there were 2,276 open issues and 22,233 closed issues in the Salt repository. The total of 24,509 issues formed the input into the bug fetching step. The output is $\mathcal{B}$, a set of HTML URLs of 8,920 bug reports fetched using the selection criteria above.

Post-Filtering. The last step of the bug collection procedure is the post-filtering. The input of the post-filtering step is $\mathcal{B}$ which is the output of the bug-fetching step. Due to multiple reasons, not all closed bug reports in $\mathcal{B}$ have explicit fixes. The aim of post-filtering is to remove bug reports that do not have explicit fixes.

In the post-filtering step, the following procedure is carried out to determine whether a bug report has potential fixes or not. From a bug report URL in $\mathcal{B}$, the issue number is extracted. To find the potential fix of a bug report, Salt's pull request submission guidelines [12] are leveraged. Essentially, it enforces all bug fix pull requests to mention the issue number of the bug report that it fixes. Exploiting this fact, the Search API from GitHub [13] is used to retrieve the URL of all pull requests that mention the said issue number in their title or description.

Sometimes developers make a quick commit directly to the source code in order to fix a bug, hence a pull request is not opened. In order to not miss such fixes, the post-filtering step additionally collects the URL of all commits that contain the said issue number in their commit message. This is done by using the git grep command on a clone of the Salt repository.

Bug reports in $\mathcal{B}$ for which no potential pull request or commit fixes were found, were excluded from the output of the post-filtering step.
The resulting output of post-filtering is, $\mathscr{B}$, a set of 5,896 HTML URLs of bug reports, each accompanied with URLs of the commits and or pull requests which are their potential fixes.

2.2 Bug Analysis

For Salt, 100 bugs are randomly sampled from the set $\mathscr{B}$ consisting of 5,896 bugs. The amount of 100 is relatively small compared to the size of $\mathscr{B}$, but this amount was chosen due to time limitations of this project. Furthermore, manual analysis of bugs is a labour-intensive process which requires quite a lot of time. Additionally, 3 other researchers, involved in the bug studies carried alongside this one, sampled 100 bugs each from Ansible, Moby and Puppet. In order to lessen the degree of subjectivity that is inherent to manual analysis, 2 teams consisting of 2 researchers each, were formed. The researchers in one team analyze bugs from Salt and Ansible, while the researchers from the other team analyzed bugs from Puppet and Moby.

The 400 bugs sampled were analyzed in 5 iterations, with 20 bugs analyzed per iteration for each configuration management system. The aim is to classify a bug using categories that were formed for each of the 6 research questions. The categories used in the analysis phase were devised by the 4 involved researchers. As more iterations were finished, the researchers could provide their perspectives on the bug analyzed to merge, add or delete categories. At the end of the iterations the 2 researchers who analyzed the same set of bugs, compare the 6 categories assigned to each independently analyzed bug. In case there is a difference in the categories, there was a discussion until a consensus was reached on the categories the bug should be assigned to.

Each bug analyzed had the following sources of information which were examined to determine the categories it belonged to:

- From the bug report, the descriptions and discussions that ensue were inspected. The reported actual behavior and the expected behavior is used to derive information about the symptom (RQ1). Oftentimes, there is also information at this site which help with answering RQ3, RQ5 and RQ6, such as how the user is impacted by the bug, the platform and the setup which triggered the bug and the exact steps taken to reproduce the bug.
- The pull request that rectifies the reported bug has a description and discussions which contain information about what caused the bug and how the bug is fixed (RQ2 and RQ4). At site of the source code where the bug manifested, the difference between the bug-causing code and the bug fix was examined. This is mainly done to answer RQ4, but it also provides information for RQ2. If a bug fix is accompanied by a test case, it is examined for RQ6. The documentation of the files and functions modified due to the bug was perused to make inferences that help with further answering RQ3.

2.3 Threats to Validity

There are potential threats to the internal validity of this study. One of threat is related to the criteria used to fetch issue from Salt's GitHub repository in the bug collection phase. To ensure that only bug reports were fetched, only issues labelled "Bug" were retrieved, this essentially avoids fetching issues which have the label "Feature" and "Enhancement". The subsequent post-filtering step, excluded bug reports for which no potential fixes were found. This is in line with prior research work done by Chaliasos et al. in 2021 [4], and Di Franco et al in 2017 [14].
Another potential threat is related to the validity of the bugs examined in the bug analysis phase. The post-filtering step described in section 2.1 only finds potential fixes for a bug and not the true fix. For instance, a pull request that is closed but not merged into the source source code would not be considered a true fix for a bug report. Such false positive bug reports might compromise the the validity of the findings in this study. To prevent this, all bugs that were randomly sampled in the bug analysis were manually inspected to make sure that there is a true fix.

Threats to external validity are associated with the representativeness of the chosen configuration management system. Salt is the configuration management system that is focused on in this study. The findings from analyzing the bugs found in Salt alone, would not be capable of being generalized to other to other configuration management systems. However, this bug study was carried alongside peer researchers who conducted bug studies on Puppet, Moby and Ansible, which makes it possible for the reader to compare the findings determine which are general enough to be applied to other configuration management systems.
Manually analyzing and categorizing the bugs introduces an element of subjectivity. To address this threat, the bugs from Salt are independently analyzed by 2 researchers. For all bugs that the 2 researchers categorized differently, a discussion ensued until a consensus was reached on which categories are applicable for the bug.

3 Bug Study Findings

This section presents the main findings of this bug study to answer the research questions enumerated in section 1.

3.1 RQ1: Symptoms

In a bug report, the reporter provides the configuration management system's actual behaviour and its expected behaviour along with details on how to trigger the bug. By studying this information from 100 randomly sampled bugs, 5 categories of symptoms were discerned. The *Unexpected Runtime Behaviour (URB), Crash (C), Performance Issue (PI), Misleading Report (MR) and Unexpected Dependency Behaviour Error (UDBE)*. The distribution of these categories is showed in Figure 1. The frequency of these categories and if applicable, their constituent subcategories are elaborated in the following paragraphs.

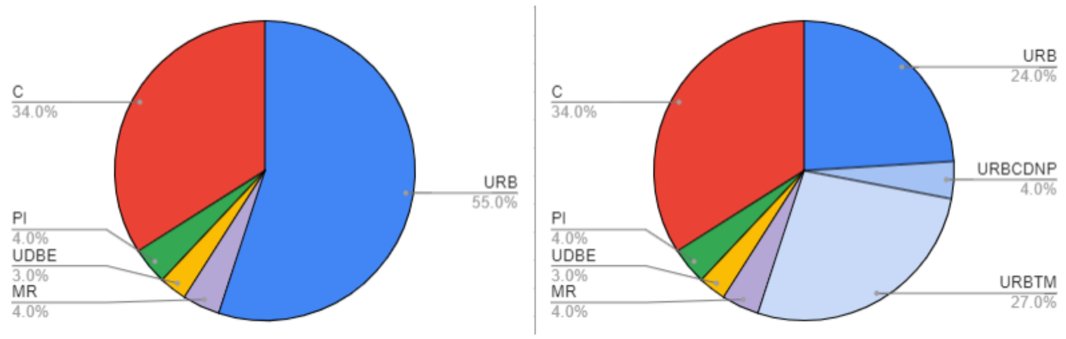

Figure 1: The left pie chart shows the distribution of the symptoms. The right pie chart breaks down the categories of symptoms into its subcategories if applicable.

Unexpected Runtime Behaviour (URB) Bugs involving this symptom are for cases where Salt's API for configuring target machines[2] do not behave in a way that the user expected (55%). A subset bugs in the URB category belong to the subcategory *URB: Target machine misconfigured (URBTM)* which accounts for 27% of the analyzed bugs. This subcategory is strictly used in cases where a Salt completes the process of configuring a target machine[3] without encountering errors, however the target machine does not conform with the user-specified configuration. Another subcategory is *URB: Configuration does not parse as expected (URBCDNP)* (4%), this is for when the user observes that the user-provided values are not parsed correctly by Salt which ultimately leads to a misconfigured target machine. Bugs belonging to the category URB but not URBTM or URBCDNP (24%), are for cases where Salt outputs an error message to the user interface, thereby failing to configure the target machine.

Crash (C) Bugs with this symptom account for 34% of the analyzed bugs, making it the second most common symptom. Whereas bugs from the URB category are able to output a nicely formatted message to the user, crashes are characterized by the stack traces printed to the user-interface.

Performance Issue (PI) Bugs with this symptom are 4% of all analyzed bugs. Bugs showing this symptom causes the Salt configuration management to become slower than usual or unresponsive.

[2]The official names are remote execution modules [15] and state modules [16]

[3]Target machines are alternatively referred to as minions, managed nodes or remote hosts.

Misleading report (MR) In this case the diagnostic error message is displayed for the user, however it suggest the wrong fix, thereby misleading the user. This symptom accounts for 4% of all analyzed bugs

Unexpected Dependency Behavior Error (UDBE) This is the least common symptom (3%). A bug related to this shows up when there's an error related to a dependency used in the source code of the configuration management system.

3.2 RQ2: Root Causes

Understanding the root cause of a bug requires examining the component of the configuration management system's source code where the bug resided. Figure 2 shows the distribution of these categories and their constituent subcategories. In the following paragraphs the categories are elaborated.

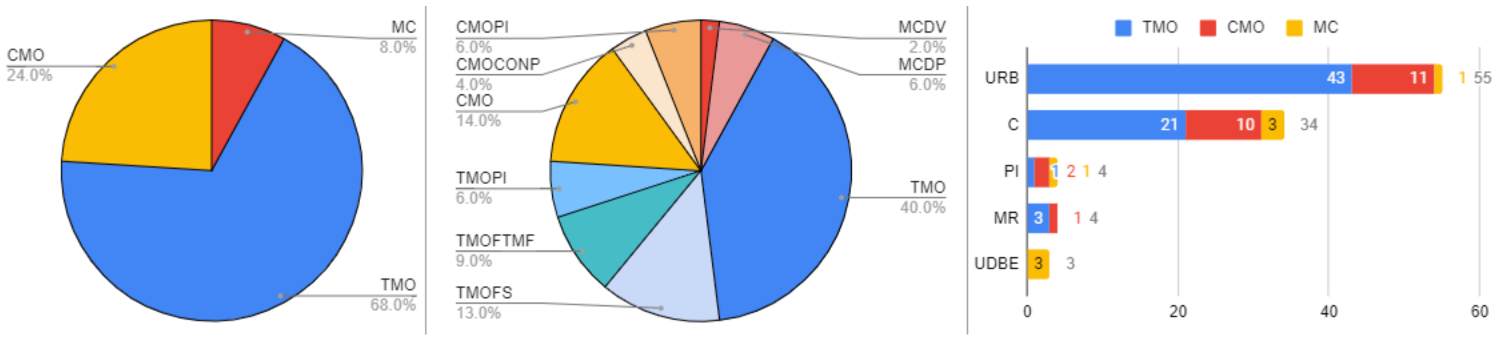

Figure 2: The leftmost pie chart shows the distributions of the main categories of root causes. Frequency of the constituent subcategories are found in the middle. The bar chart on the right shows the distribution of root causes per symptom category.

Target Machine Operations (TMO) This category is the most common (68%) and bigger all other remaining categories combined. The bar chart in Figure 2 shows that TMO is the main root causing the majority of the Crash and Unexpected Runtime Behaviour symptoms. Bugs belonging to this category, reside in the implementation of Salt's various execution and state modules which are executed on the target machines [15, 16]. *TMO: File System Operations (TMOFS)* (13%). Some examples are bugs in Salt's implementation for managing ZFS Datasets and rsync wrapper. *TMO: Parsing Issue (TMOPI)* (6%) is a category used for bugs located in parsing methods used in execution and state modules. *TMO: Fetch Target Machine Facts (TMOFTMF)* are used to label bugs residing in commands that need to be executed on target machines e.g. DNS facts and IP discovery. The root cause of bugs belonging to the category TMO but not TMOPI, TMOFTMF and TMOFS (40%) are diverse. Some examples are bugs in implementation of various modules which then prevent user management, executing CMD scripts and spinning up cloud instances on target machines.

Controller Machine Operations (24%) Bugs involving this root cause are related to the core features of Salt, such as connecting to the target machines specified by the user *(CMO: Connection Problems (CMOCONP) 4%)*. Bugs can also be occur in other operations done by the controller machine such as parsing and compiling the YAML files which contain the configuration desired *(CMO: Parsing Issue (CMOPI))*. Looking at the bar chart in Figure 2, bugs in controller machine operations are the second biggest contributor to URB and C symptoms.

Misconfiguration inside the Codebase (MC) is the least common root cause of all bugs analyzed (8%). This category of root cause is broken down into the following subcategories:

- **MCDV**: Misconfiguration of default values inside the codebase. Bugs with this root cause account for 2% of all analyzed bugs.
- **Misconfiguration of dependencies inside the codebase (MCDP)**: The codebase contained incorrect dependency configuration. This relates to any wrong setup of methods from dependencies, wrong dependency import, wrong dependency version. 6% bugs analyzed have MCDP as a root cause.

3.3 RQ3: Impact

While analyzing the impact of bugs we grouped the bugs in 2 categories. A bug is assigned one label for describing the level of impact they had and one for the consequence the bug has from the user's perspective. Figure 3 show the distribution of these 2 groups of categories. The first group category describe how disruptive the unwanted behavior is to a system and consists of the following categories:

- **Low**: System works overall besides in specific edge cases. Bugs with a low level impact affect very few users or a work around exists for them (29%).
- **Medium**: System starts and works for the majority of cases but fails when performing one important task (53%).
- **High**: System will not compile or start and it fails in performing two or more important tasks (18%).

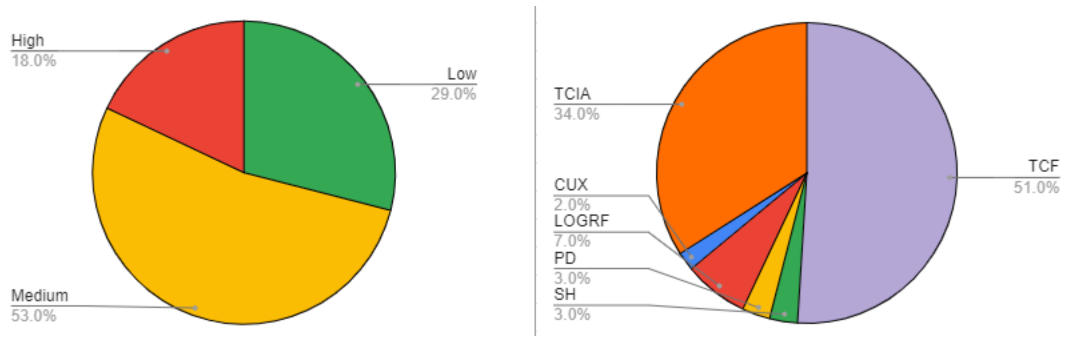

Figure 3: The leftmost pie chart shows a distributions of the impact that a bug had. The rightmost pie chart shows the distributions of the category of consequence of the bugs from the user's perspective

The other group of categories relating to the consequences are:

- **Target Machine Configuration Failure(TCF)**: Salt fails to configure the target machine. This is the most common consequence accounting for 51% of all bugs
- **Target Machine Configuration Inaccurate (TCIA)**: Salt misconfigures a target mchine. This is the second most common consequence (34%).
- **Log Reporting Failure (LOGRF)**: Salt fails to log the correct information to the user. 17% of the analyzed bugs had this conquence.
- **Performace Degradation (PD)**: The system performance is affected significantly. (3%).
- **Security Hazard (SH)**: 3% of bugs of bugs introduce a security problem.
- **Confusing User Experience (CUX)**: It is not clear to the user how a feature in Salt can be used. This is the least common consequence (2%).

3.4 RQ4: Bug Fixes

Zhao et al.[17] devised a code change taxonomy to classify bug fixing code. This taxonomy was used to study the code changes made to Salt's source code to fix a bug:

- **Change Branch Statement (CBS)** 41% of the bug fixes were involved the modification of code related to the control flow e.g. if and switch statements.
- **Change Assignment Statements (CAS)** 17% of the bug fixes involve a changes in assignment statements.
- **Invocation of a Method (IM)** 17% of the bugs were fixed by removal or addition of a method invocation.
- **Add Methods (AM)** 12% of the bugs were fixed by implementing new methods.
- **Change in Dependency (CDEP)** Change dependencies (4%)
- **Change Method Interfaces (CM)** The method declaration is changed e.g. adding or removing parameters of a method (5%)
- **Change in return statements (CRS)** 3% of the bugs were fixed by changing return statements.
- **Change in loop statements (CLS)** 1% of the analyzed bugs were fixed by changing loop statements.

3.5 RQ5: System-Dependence

Out of the 100 bugs analyzed, 72 were system-independent, meaning they manifest regardless of the underlying system. The remaining 28 bugs were dependent on the underlying system. The left column chart in Figure 4 shows that the system-dependent bugs mainly depend on Linux and Windows systems. It also shows other systems which bugs depended on, such as SUSELinuxEnterprise, MacOs, IBM AIX etc.

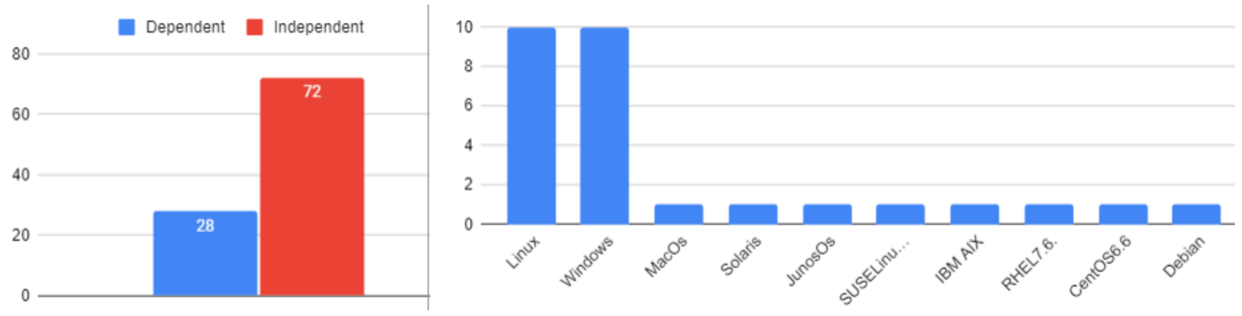

Figure 4: The left column chart shows how many bugs were dependent on the underlying system. The right column chart shows the number of bugs for each system.

3.6 RQ6: Triggers

The bugs were assigned one category for the type of error that triggered the bug. *Logic Errors (LE)* is the most common error (54%). Examples of logic errors are missing branches and cases, incorrect sequence of operations and passing incorrect parameters to a function. *Programming Errors (PE)* are the second most common (26%). Access of null references or variables that are out of bounds and unchecked exceptions are examples of this type of error *Algorithmic Errors (AE)* account for 13% of the errors that triggered a bug. In this case the implementation of a functionality offered in Salt is wrong or uses the wrong algorithm. *Configuration Errors (CE)* are the least common (5%) and are related to dependencies of Salt's source code, such as wrong or outdated imports.

Additionally, at more categories are assigned which describe what must be done to reproduce the bug. The rightmost pie chart of Figure 5 shows the distribution of the categories of trigger conditions. The various trigger conditions are elaborated below:

- **Specific Invocation (SI)** is the most common (60.4%). Bugs belonging to this category are reproduced by invoking a specific function in the Salt state and execution modules with specific circumstances.
- **Test case (TC)** (21.6%) For bugs belonging to this category, a test cases accompanied the corresponding fix, this test case can be run to trigger
- **OS specific execution (OSSE)** The bug can be reproduced by running the confguration management operations on a target machine with a specific OS (12.6%).
- **Faulty Dependency Usage (FDEPU)** The bug can be reproduced by using a faulty version of a dependency (3.6%).
- **Environment setup (ENVS)** The bug can be reproduced by specific setup in the service environment on the target machine (1.8%).

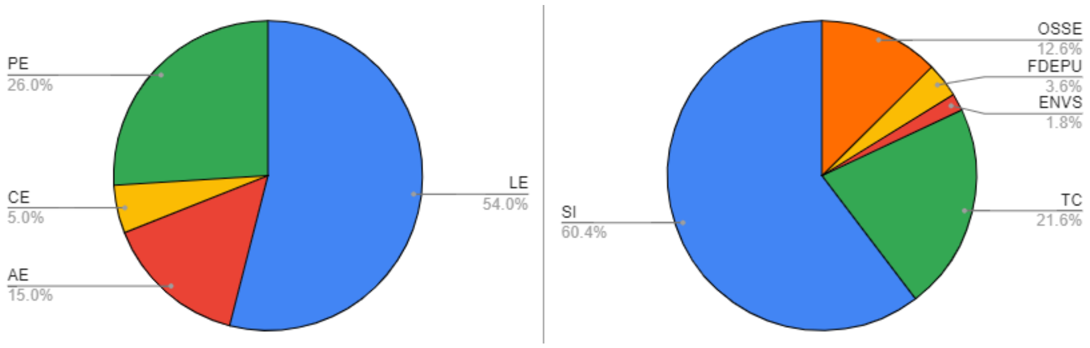

Figure 5: The left pie chart shows the distribution of the types of errors that triggered the bug. The rightmost pie chart shows the distribution of the categories used to describe how the big was triggered.

4 Discussions

The findings of the bug analysis describe a clear pattern that most bugs found in Salt follow. The majority of the bugs have the symptom unexpected runtime, wih erroneous target machine operations. As a consequence, Salt fails to configure the target machine. .....

5 Related Work

In the scientific literature, there is lack of of bug studies on configuration management systems. As a result, bug studies on other classes of software were consulted to get an idea of how to conduct bug studies.

The methodology used in this paper to study bugs found in Salt, is adopted from the bug study done by Chaliasos et al. [4], with some minor modifications. The post-filtering step done here differs with the one done in bug study conducted by Chaliasos et al. [4] in that it does not exclude bug reports if their potential fixes are not accompanied by test cases. This decision was made because our supervisor noted that it is possible to gather trigger information needed for RQ6 even if there isn't a test case.

While studying the fixes of the bugs

studied the symptoms, bug causes, bug fixes and test case characteristics for typing-related bugs in JVM compilers.

6 Responsible Research

The Netherlands Code of Conduct for Research Integrity [18] provides principles which serve as guidelines researchers so that they make the right choices in all circumstances. Adhering to this code of conduct ultimately leads to ethical and responsible research practices. A reflection on the ethical aspects and the reproducibility of this bug study is presented in this section.

Upon reflection, it was realized that the bug fetching could be improved to be more ethical. In addition to collecting the URLs using the GitHub REST API [11], the Python script collected the GitHub usernames of the people who reported the issue and the people assigned to work on the issue were also inadvertently collected by the Python script. They were collected to check whether the assignee is the same person as the reporter, but this could have been determined in a more anonymous fashion. The great majority of the GitHub usernames collected are not the real names of the users, but it undoubtedly compromises the an the anonymity of a few.
In order to avoid repeating this mistake in the future, the author of this paper will avoid collecting usernames and other personal information in the future. To prevent people who decide to reproduce the bug fetching procedure from making this same mistake, the author has modified the Python script to not collect usernames anymore.

Full transparency is indispensable if one is to make sure the conducted research is reproducible. The methodology of this paper documented the different steps taken to arrive at the bug data set used in this bug study. To further improve the reproducibility of the bug collection process, the scripts utilized for gathering and post-filtering are made available on this open-source GitHub repository.

The 100 bugs that were analyzed in the bug analysis phase were randomly sampled. It was decided to use this method as opposed to hand picking bugs from the bug data-set, in order to ensure that the results representative and not skewed. This decision introduced an element of randomness, thereby making it difficult for people who aim to reproduce the bug analysis, to obtain the bugs analyzed in this study. To solve this concern, the 100 bugs that were randomly sampled and the constructed data-set from which they were sampled are available on the aforementioned GitHub repository.
It is important for research work to be reproducible because it necessary for establishing the validity of the reported scientific findings [19]. Despite the fact that the bugs in this study are independently analyzed by 2 researchers, the effects of personal bias are not completely eliminated. To facilitate validation of the results in this paper, the categorization of all bugs analyzed is also publicly available.

7 Conclusions and Future Work

- Briefly summarize the main research questions.
- Provide the answers to the research questions.
- Highlight interesting elements, contributions.

- Discuss open issues, possible improvements, and new questions that arise from this work; formulate recommendations for further research.
- IMPORTANT: No new concepts here!
- IMPORTANT: It should be readable without having read the earlier sections and accessible to anyone with a bachelor degree in Computer Science.

This bug study aimed to find of bugs previously found in the Salt configuration management system.

References

[1] Tianyin Xu and Yuanyuan Zhou. “Systems Approaches to Tackling Configuration Errors: A Survey”. In: *ACM Comput. Surv.* 47.4 (July 2015). ISSN: 0360-0300. DOI: 10.1145/2791577.

[2] Guoliang Jin et al. “Understanding and Detecting Real-World Performance Bugs”. In: *SIGPLAN Not.* 47.6 (June 2012), pp. 77–88. ISSN: 0362-1340. DOI: 10.1145/2345156.2254075.

[3] David Oppenheimer, Archana Ganapathi, and David A. Patterson. “Why Do Internet Services Fail, and What Can Be Done about It?” In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS’03. Seattle, WA: USENIX Association, 2003, p. 1.

[4] Stefanos Chaliasos et al. “Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485500.

[5] Vasileios Pappas et al. “Impact of configuration errors on DNS robustness”. In: *IEEE Journal on Selected Areas in Communications* 27.3 (2009), pp. 275–290. DOI: 10.1109/JSAC.2009.090404.

[6] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. “Asserting Reliable Convergence for Configuration Management Scripts”. In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 328–343. ISSN: 0362-1340. DOI: 10.1145/3022671.2984000.

[7] Red Hat Ansible. *Ansible is Simple IT Automation — ansible.com*. https://www.ansible.com/. [Accessed 15-Jun-2022].

[8] *Moby — mobyproject.org*. https://mobyproject.org/. [Accessed 15-Jun-2022].

[9] Puppet Webteam. *Powerful infrastructure automation and delivery — Puppet — puppet.com*. https://puppet.com/. [Accessed 15-Jun-2022].

[10] *GitHub - saltstack/salt: Software to automate the management and configuration of any infrastructure or application at scale. Get access to the Salt software package repository here: — github.com*. https://github.com/saltstack/salt. [Accessed 15-Jun-2022].

[11] *GitHub REST API - GitHub Docs — docs.github.com*. https://docs.github.com/en/rest. [Accessed 24-Apr-2022].

[12] *Pull Requests — docs.saltproject.io*. https://docs.saltproject.io/en/latest/topics/development/pull_requests.html. [Accessed 15-Jun-2022].

[13] *Search - GitHub Docs — docs.github.com*. https://docs.github.com/en/rest/search. [Accessed 15-Jun-2022].

[14] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. “A comprehensive study of real-world numerical bug characteristics”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 509–519. DOI: 10.1109/ASE.2017.8115662.

[15] *Remote Execution — docs.saltproject.io*. https://docs.saltproject.io/en/latest/topics/execution/index.html. [Accessed 18-Jun-2022].

[16] *How Do I Use Salt States? — docs.saltproject.io*. https://docs.saltproject.io/en/latest/topics/tutorials/starting_states.html. [Accessed 18-Jun-2022].

[17] Yangyang Zhao et al. "Towards an understanding of change types in bug fixing code". In: *Information and Software Technology* 86 (2017), pp. 37–53. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.02.003.

[18] KNAW; NFU; NWO; TO2-federatie; Vereniging Hogescholen; VSNU. "Nederlandse gedragscode wetenschappelijke integriteit". In: (2018). DOI: 10.17026/dans-2cj-nvwu.

[19] David B. Allison, Richard M. Shiffrin, and Victoria Stodden. "Reproducibility of research: Issues and proposed remedies". In: *Proceedings of the National Academy of Sciences* 115.11 (2018), pp. 2561–2562. DOI: 10.1073/pnas.1802324115.