A Study of Bugs Found in the Puppet Configuration Management System

Mykolas Krupauskas
Responsible Professor: **Diomidis Spinellis**
Supervisor: **Thodoris Sotiropoulos**
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

This research studies the symptoms, root causes, impact, triggers, fixes, and system dependency of bugs in the Puppet configuration management system. Puppet is a widely used open-source configuration management system that performs various administrative tasks on machines based on a central specification. This paper aims to fill the research gap and help improve the understanding of these systems. 2146 registered bugs and fixes are collected and a random sample of 100 is analyzed to answer the research questions. The most common bug types have the symptom of unexpected runtime behavior. The root cause is generally incorrect target machine operations. The impact severity is typically medium and the impact consequence of most bugs is target configuration failure. Bug fixes tend to be small method code changes and usually occur in the execution component. Bugs are generally system-independent and are caused by logic errors that can be triggered by specifically executing an erroneous module. The analysis shows that applying automated bug discovery methods to the most error-prone execution and parser components of the system would lead to the largest improvement in the system's quality.

1 Introduction

Configuration management systems are highly complex pieces of software that are widely used in the industry. However, there is a visible gap in the research on these systems. Bug studies can help us better understand the properties of these systems, especially in the bug detection and prevention areas. The knowledge gained from bug studies can be used to better educate engineers and build tooling to detect bugs in these systems automatically. Both of these methods would contribute greatly to the quality and reliability of configuration management systems at large.

This study takes inspiration from previously conducted bug studies that explored different systems and problem spaces. The paper studying Java Virtual Machine compilers and their typing-related bugs [1] is the basis for the methodology of this study, where bug reports and fixes are collected from open sources, a random sample of bugs is categorized as well as analyzed, and conclusions are then drawn from the categorizations. Research on actor concurrency bugs [2] inspires the bug collection and categorization methodology with modifications to the data sources and the categories used for analysis. The bug fix type categorization builds off of previous research conducted by Yangyang Zhao in 2017 [3] with modifications specific to the configuration management system context. While the papers mentioned serve as inspiration, this study presents a novel combination of the methodology of a bug study to explore a complex, crucial, yet often neglected class of software known as configuration management systems.

The research aims to identify and categorize various bugs and their fixes in the Puppet configuration management system. Puppet is an open-source system that provides an automated way for system administrators to manage their low-level IT infrastructure based on a centralized configuration specification. The categorization of bugs in Puppet in terms of symptoms, triggers, impact, and system-dependency aims to reveal patterns that would assist in better understanding the underlying system and creating potential solutions to fixing bugs more effectively, which would positively affect the quality and reliability of the system.

This research outlines the process of studying bugs in configuration management systems which helps answer the research questions posed in section 1.2. Additionally, the data collection process, automated tools created for it, and the data itself are documented and open-sourced to be used for further research and analysis.

The paper is structured in terms of sections. Section 1.1 provides context around the concepts explored

in the research. Section 1.2 highlights the research questions posed and explored by the paper. Section 2 describes the methodology and approach of the bug study. Sections 3 and 4 outline the bug study setup and its results. Section 5 discusses the ethical implications and the reproducibility of the research. Finally, sections 6, 7, 8, and 9 discuss the findings, explore related research in the context of this study, raise conclusions, and outline future work, respectively.

1.1 Concepts

Before exploring the study in detail, it is important to discuss the main definitions and terminology that is relevant to the paper. Section 1.1.1 defines what a configuration management system is, section 1.1.2 describes bug analysis studies, and section 1.1.3 highlights the Puppet system in general.

1.1.1 Configuration Management Systems

Configuration management systems are a class of software that allows system administrators to define a central configuration for a set of managed machines to be configured by [4]. The configuration can define the desired state of the system, which is then used as a template to get a managed machine to that state automatically. These systems are often responsible for the lowest layer of an organization's IT infrastructure. Using a configuration management system to manage infrastructure in an automated way is highly preferable to manual configuration which is vulnerable to misconfiguration due to human error that can cause downtimes, loss of productivity, security breaches, and compliance issues.

1.1.2 Bug Analysis

A failure in software is an instance of it behaving in an undesired or unexpected manner which can cause issues for the end-user. A bug is defined as a mistake in the implementation of the software which causes the failure.

Bug analysis is a process that is done to deepen the understanding of a system and to improve its quality as well as reliability. This process can be broken down into two distinct steps of bug collection and bug analysis. Bug collection is the process of gathering bug reports and their fixes to a common collection. The source of the bugs can be issue trackers as well as code repositories of the software project. Bug analysis is then performed on the collected sample, where bugs are categorized and grouped. Finally, conclusions can be drawn from the analyzed dataset on the properties of the system and its issues.

1.1.3 Puppet

Puppet is a widely used, open-source configuration management system. "Puppet, an automated administrative engine for your Linux, Unix, and Windows systems, performs administrative tasks (such as adding users, installing packages, and updating server configurations) based on a centralized specification." [5]. The system is used by enterprises like Uber, the New York Stock Exchange, and Twitter to allow system administrators to manage the lowest level of their IT infrastructure automatically based on a central configuration [6].

1.2 Research Questions

The goal of this study is to analyze and categorize a set of bugs found in the Puppet configuration management system. This categorization procedure can be split up into multiple research questions that are answered. With the answers to these questions, conclusions can be drawn about the common properties of bugs in configuration management systems.

1. What are the **symptoms** of bugs in Puppet?
2. What are the **root causes** of bugs in Puppet?
3. What is the **impact** of bugs in Puppet?
4. What are the **fixes** to bugs in Puppet?
5. What is the **system-dependency**[1] of bugs in Puppet?
6. What are the **triggers** of bugs in Puppet?

1.3 Conference Submission

This paper, together with related papers studying bugs in configuration management systems Ansible, Salt, and Moby, will be submitted to the 45th International Conference on Software Engineering [7] under the technical track for configuration management systems to be shared with the wider research community.

2 Methodology

The methodology of the research can be split into 3 parts. Firstly, bug reports and bug fixes are collected to a central data store. The collected bugs are then analyzed based on the categories that are observed in a randomly selected sample in multiple iterations. Finally, conclusions can be drawn from the statistics derived from the analyzed sample. These statistics will help uncover trends in the bugs and answer the research questions posed in section 1.2. This process is inspired by the study of JVM compiler bugs [1] which follows a similar methodology. Each step of the process is explained in greater detail in the rest of this section.

The bug collection process requires 2 data sources to fetch the data needed for analysis. One data source is used for bug reports and the other one is used for bug fixes. The Puppet project maintains an open Jira issue tracker [8] where users can report bugs and track the progress of their fix. To manage the project's code and pull requests, Puppet uses a Github code repository [9] which is the data source for the fixes of the bugs. The bug reports and fixes are fetched in a normalized JSON format via automated tooling. Specifically, the open-source project Perceval [10] is used for data collection from both Jira and Github. The data is filtered based on various criteria like the issue type. The data is also mapped and aligned to fit the common bug schema found in appendix A.1. After the data is collected, filtered, and mapped, it is then stored in the central bug database.

The bug analysis process starts with a random sampling of 100 of the bugs collected in the previous step. The sample is then split into multiple iterations of 20 bugs. The first iteration is used to create initial categories for each of the properties of the bugs mentioned in the research questions found in section 1.2. To minimize the effect of personal bias in the categorization the process is repeated between pairs of researchers. The results of the analysis are discussed after the iteration and consensus on how the bug should be categorized is reached through discussion.

The random sample of analyzed bugs provides a basis from which statistics and conclusions about the trends in the configuration management system can be made. The research questions regarding the symptoms, root causes, triggers, system-dependency, impact, and fixes of the bugs are also answered

[1] A bug is said to be system dependent when it only manifests in a specific configuration of a system. For example, a bug affecting Windows but not Linux is system dependent.

by analyzing this categorization. The categorizations are stored in the central bug database and made available for further analysis and discussion.

3 Bug Study Setup

The bug study setup section aims to explain the research setup in detail. The setup is further split into bug collection and bug analysis. The challenges that were faced, the decisions, and the adjustments that were made to accommodate them are also documented in this section.

3.1 Bug Collection

The bug collection process combines multiple data sources for bug reports and bug fixes into one bug schema which is then stored in a central bug database to be sampled and analyzed. Automated tools and custom scripts are used to fetch, filter, map, and store this data. Specifically, the open-source project Perceval [10] is used for the data fetching from both Jira and Github in a structured JSON format which is then further refined. Additional custom scripts are used to perform data transformation operations and their documentation is open-sourced on Github [11] to enable independent verification of the research.

Jira is the data source for all registered bug reports for the Puppet project [8]. The bug report serves as the basis of the bug as it is used as the source of truth for the unexpected system behavior which caused a user to file a bug report in the first place. The total number of unique issues registered in Jira as of publication is 10136. To filter for issues only representing bugs the issue type field with the "Bug" value is used. After applying this filter on the issue type 4359 issues concerning bug reports are left. After filtering the Jira issue data is cleaned up by removing irrelevant fields and aligning the schema to the common bug schema via automated scripts.

Github is the data source for the source code, pull requests, and fixes for the Puppet project [9]. The total number of pull requests collected at the time of publication is 8906. The pull requests often contain additional context around the proposed change or bug fix. Additionally, test cases in pull requests are highly encouraged to demonstrate that a proposed fix can trigger a bug and ensure that the fix produces the correct behavior in the system. This provides crucial information that is used in the bug analysis stage. The pull request data is also mapped and cleaned up to include only relevant data fields.

The final stage of the data collection process is associating the bug report data with the bug fix data and merging it into a final common bug schema. This is done with the use of automated scripts that iterate through the list of bug reports and associate a unique Jira issue number to a pull request. This is done by relying on the convention of the Puppet project which states that a pull request related to a registered Jira issue must contain the Jira ticket number in its title. This convention is strictly enforced by the maintainers for outside contributions and is documented in the project's contributing guidelines [12]. The bug report and bug fix association result in 2031 bugs with related pull requests which can then be analyzed.

Since the analysis is only performed on bugs with fixes, it is important not to miss bug reports that might have fixes associated with them in the data collection stage. During the initial iteration of the bug collection, it was discovered that Puppet project maintainers could circumvent the requirement to associate a pull request to a Jira issue by committing a bug fix to the project directly. This case would lead to the initial bug report and fix process to misclassify certain bug reports as missing fixes. To remediate this, the complete history of the commits of the Puppet repository is also parsed. This results in 34123 commits being collected. The bug collection process was also modified to account for

Type	Number
Issues	10136
Issues with the "Bug" Type	4359
Pull Requests	8906
Commits	34123
Issues with an Associated Pull Request	2031
Issues with an Associated Commit and no Associated Pull Request	115
Issues with an Associated Pull Request or an Associated Commit	2146

Table 1: Number of Collected Bug Reports and Fixes by Type

bug fixes found in commits created without pull requests. If a Jira issue number is found in a commit message, that commit is also associated with the bug report as a fix. Therefore, even bug fixes without pull requests are also detected. This results in an additional 115 bugs with fixes being added to the data set that would have otherwise been missed.

The bug collection process is finalized by loading all of the collected bug reports with associated fixes into the central database with the common bug schema. A final summary of the collected issues and their counts can be seen in table 1. After the database of bugs is ready and prepared, the analysis procedure follows.

3.2 Bug Analysis

The bug analysis step is performed by manually investigating and categorizing a randomly selected sample of 100 bugs from 2146 of the bugs collected. This relatively small sample is chosen due to the time limitations of the research and this is an area for improvement discussed in section 9 on future work. The process is done in pairs of two researchers to minimize the effect of personal bias on the categorization. Differences in the categorization are discussed and a consensus is reached on how the bug should be categorized. The sample analysis is further broken down into 5 iterations of 20 bugs to allow for iteration on the categorization of the bugs as new samples are analyzed.

The categorization is specifically geared towards answering the research questions posed in section 1.2. The process itself is executed by examining the bug report in Jira manually to categorize the properties of the bug. Additionally, by investigating the pull request or the commit that fixes the bug the categorization for the fix and the behavior of the bug can be further refined. Below, the categories used for the analysis are listed and grouped by the research sub-question they tackle.

3.2.1 Symptoms

The symptom is the feature of the bug that indicates the way the system is misbehaving to convince the user that the system is experiencing unwanted behavior.

The categories are: unexpected runtime behavior (**URB**), configuration does not parse as expected (**URBCDNP**), target misconfiguration (**URBTM**), misleading report (**MR**), unexpected dependency behavior error (**UDBE**), performance issue (**PI**), crash (**C**), feature not functional (**CFNF**), execution crash (**CEC**), configuration parsing crash (**CEC**), environment related error (**CERE**).

3.2.2 Root Causes

The root sources of the unwanted behavior that cause the system to misbehave.

The categories are: error handling & reporting bugs (**EHRB**), misconfiguration in the codebase (**MC**), misconfigured default values (**MCDV**), misconfigured dependencies (**MCDP**), target machine operations (**TMO**), incorrect file system operations (**TMOFS**), target machine dependency issues (**TMOD**), fetch target machine facts failure (**TMOFTMF**), parsing issue (**TMOPI**), instruction translation error (**TMOITE**), controller machine operations (**CMO**), executor problems (**CMOEP**), connection problems (**CMOCONP**), parsing issue (**CMOPI**).

3.2.3 Impact

The disruptiveness level and the consequences of the bug to the end-user. The level accounts for how common and drastic the effect of the bug is. The consequence describes the user-facing behavior of the bug.

Level The categories are: **low** - the system works well besides rare edge cases, **medium** - the system fails to work when executing a common use case, **high** - the system fails to work completely or crashes.

Consequence The categories are: security hazard (**SH**), performance degradation (**PD**), logs reporting failure (**LOGRF**), target configuration failed (**TCF**), target configuration crash (**TCFC**), target configuration inaccurate (**TCIA**), target configuration incomplete (**TCIN**), confusing user experience (**CUX**).

3.2.4 Fixes

The code and conceptual system changes represent fixes made to resolve the bug. The code fix is the concrete change to the source code. The conceptual fix is a higher-level feature that describes the fix in terms of which system component it changes and how.

Code Fix The categories are: change on data declaration initialization (**CDDI**), change on assignment statements (**CAS**), add class (**AC**), remove class (**RC**), change class (**CC**), add method (**AM**), remove method (**RM**), change method (**CM**), change loop statements (**CLS**), change branch statements (**CBS**), change return statement (**CRS**), invoke method (**IM**).

Conceptual Fix The categories are: fix execution component (**FEC**), fix parser component (**FPC**), fix connectivity component (**FCC**), expand execution feature (**EEF**), expand parser feature (**EPF**), expand connectivity feature (**ECF**), change dependencies (**CDEP**), change system structure (**CSS**), change configuration (**CCONF**), display diagnostic message (**DDM**).

3.2.5 System-Dependency

System-dependency describes the bug's tendency to only manifest in a specific system configuration or environment.

The categories are: **true** - the bug only appears in a specific system configuration, **false** - the bug appears regardless of the system configuration.

3.2.6 Triggers

The original cause of the bug and how the bug is reproduced. The cause represents the initial trigger that introduced the bug. The reproduction trigger shows how a user can trigger the bug.

Cause The categories are: logic errors (**LE**), algorithmic errors (**AE**), configuration errors (**CE**), programming errors (**PE**).

Reproduction The categories are: command line interface commands (**CLIC**), environment setup (**ENVS**), faulty dependency usage (**FDEPU**), operating system specific execution (**OSSE**), test case (**TC**), specific invocation (**SI**), target machine control execution (**SITMCE**), internal module invocation (**SIIMI**), custom module invocation (**SICMI**), target machine related parsing (**SITMRP**), config or runbook parsing (**SICRP**).

3.3 Bug Analysis Tooling

Manual bug analysis is an extremely time-consuming effort. Since analyzing bug reports and their associated fixes is the core of the bug study from which conclusions about the system are drawn, it is important to make sure that the process of the analysis is streamlined and sped up. This way, larger samples can be analyzed in more detail by more people with less time invested.

To achieve these goals in the bug study and to make the analysis process far more efficient additional bug analysis tooling was developed in the form of an interactive web application that enables the researchers to analyze bugs and their fixes in a more streamlined way. The web application takes in a sample of a bug report and bug fix links. The researcher can then iteratively analyze the sample by using helpful automation that removes the need to manually open links and speeds up the categorization process by exposing an array of selectable inputs for each category. An example of the usage of the tooling can be found in appendix A.2.

The tooling directly contributed to the increase of efficiency in the bug analysis process. Therefore, a larger sample of the bugs was analyzed compared to what would have been possible without it. Additionally, this tooling could be extended to support other types of bug analysis studies. The initial version of the tool used in this study is open-sourced and available to be used by other researchers.

4 Bug Study Results

The bug analysis results section discusses the main patterns observed when categorizing the selected sample of bugs. Each research question posed is answered by studying the patterns in the distribution of the chosen categories. The categorizations used are a combination of the results of multiple researchers to mitigate the effect of personal bias in the categorization.

4.1 Symptoms

The most common category for the symptoms is the unexpected runtime behavior with 44% of the bug sample being categorized like that. The unexpected runtime behavior category is further broken down into runtime configuration parsing issues being 23% of the sample and target misconfiguration being 7% of the sample. The second-largest symptom category is the misleading report which applies to 14% of the sample with issues related to errors and error messages. Outright crashes of the system are comparatively rare with only 9% of bugs being categorized like that. Performance issues and unexpected dependency behavior only had 2% and 1% of bugs, respectively. The symptoms distribution is visualized in figure 1.

Since Puppet is written in Ruby, which is compiled and has a strong type system, outright crashes are not the leading symptom of bugs because most illegal operations are caught in the compilation

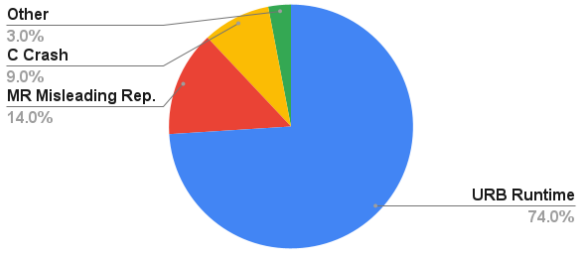

Figure 1: Symptoms Distribution

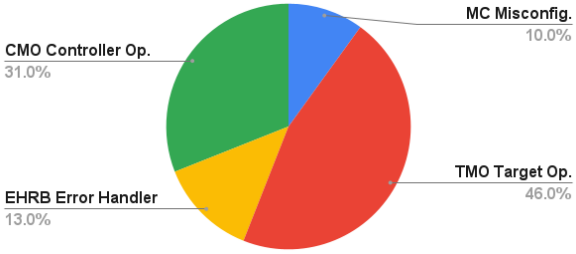

Figure 2: Root Causes Distribution

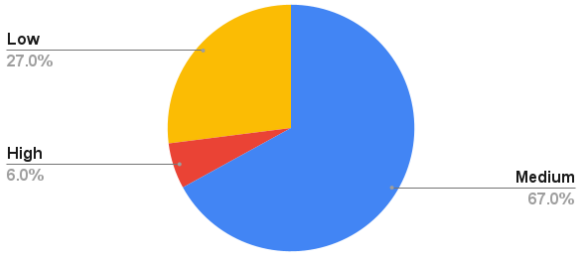

Figure 3: Impact Level Distribution

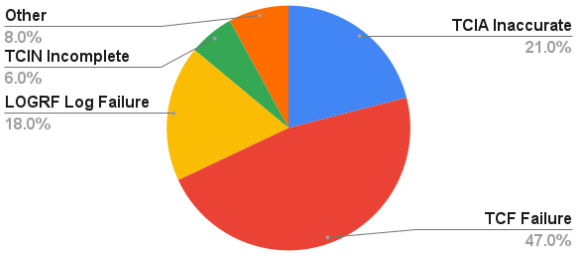

Figure 4: Impact Consequence Distribution

and testing stages. Unexpected runtime behaviors can, however, get past these stages if there is not a specific test case in the test suite triggering that bug. This explains the dominant distribution of this category. Therefore, to meaningfully reduce the number of bugs found at runtime methods like fuzz testing could potentially be applied [13, 14].

4.2 Root Causes

Nearly 50% of all root causes were determined to be part of the target machine operations category. Target machine operations can be further broken down into roughly equal parts in terms of the number of cases by target file system operations, instruction translation, and parsing issues. Target machines are more commonly called agents when discussed in the context of Puppet. The second-largest category is the controller machine operations with 30% of the bugs falling into it. Controller machine operations include parsing, connection, and other components which run on the Puppet server. The two smallest categories sharing roughly 10% of the sample each are misconfigurations like incorrect default values and faulty error handling and reporting code. The root causes distribution is displayed in figure 2.

From the distribution of the root causes of the bugs, we can determine the components of the configuration management system that are most likely to contain bugs. This distribution can guide software developers to spend additional resources to ensure the correctness of the behavior in these components by expanding test suites that cover the most common target machine and controller machine operations. This targeted investment would have the greatest return of investment in terms of bugs caught versus time invested.

4.3 Impact

The impact is broken down into two parts. The level represents the severity of the bug, where low means the system works well beside some edge case, medium means that some feature of the system is not working correctly, and high means that a core part of the whole system is completely dysfunctional. The consequences part of the impact represents the impact that is observed from the user's perspective. The impact level and consequences distributions are visualized in figures 3 and 4.

4.3.1 Level

The most common category for the impact level is medium with 67% of the sample. This means that most of the bugs analyzed cause a non-critical feature of the system to behave incorrectly. For example, the Puppet cron job configuration rule would be incorrectly translated to the target machine's cron configuration file. The next largest impact level is low with 27% of the bugs. These are mostly issues that do not hinder the user's workflow in a major way. Most commonly these are misleading error messages or slight misconfigurations. High-impact bugs are rare with only 6% of bugs from the sample. This means that outright system crashes and complete configuration failures are rare.

4.3.2 Consequences

The target configuration failure is the most common consequence of Puppet bugs with 42% of the sample. This matches up with the trend seen in the symptoms and root causes categories of bugs related to a certain type of failed target configuration being most common. The second most common category is the target configuration inaccurate containing 21% of the bugs. The issues in this category result in an often successful but not strictly correct configuration. The third most common category is log reporting failure with 18% of the bugs. This category contains bugs related to incomplete, inaccurate, missing logs or error reporting by the system. The target configuration incomplete and target configuration crash were found in 6% and 5% of the bugs, respectively. These categories group issues that cause early configuration terminations or certain unset missed parameters in the configuration. The confusing user experience, performance degradation, and security hazard categories are the smallest ones being applied for 4%, 2%, and 2% of the bugs in the sample, respectively.

While log reporting failures can be seen as affecting the user experience, the confusing user experience category is reserved for bugs related to confusingly handling user actions or input. All other bugs related to logging failures are assigned to the log reporting category even if they could be seen as affecting the user experience too.

4.4 Fixes

Bug fixes are categorized into two parts. The first one is the type of code fix applied and the second one is a higher-level conceptual fix that was made to the system. The code and conceptual fix distributions are visualized in figures 5 and 6.

4.4.1 Code

The most common code change was classified as a method change with 55% of the bugs falling under this category. A method change can involve multiple assignments, conditional, and other statements to change the logic of a method in a way that fixes the underlying bug. The second most common category was adding a new method that performed some additional functionality to fix the bug with 15% of the bugs having this type of fix. All of the other categories only reached single-digit percentages. The

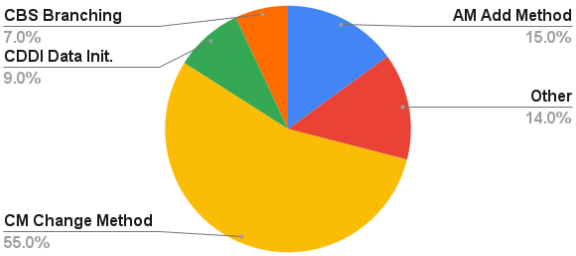

Figure 5: Code Fix Distribution

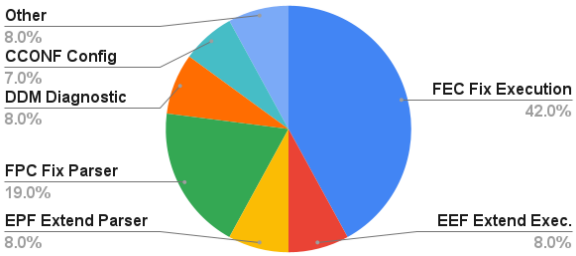

Figure 6: Conceptual Fix Distribution

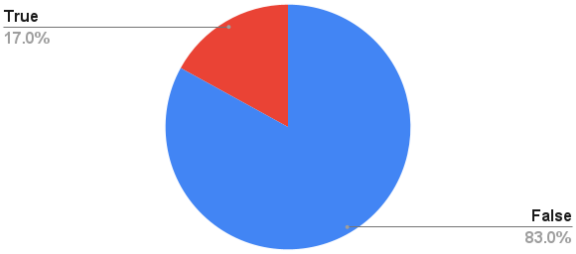

Figure 7: System-Dependency Distribution

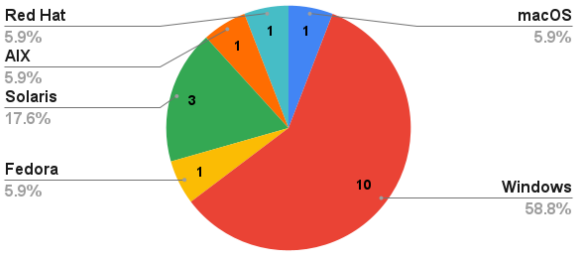

Figure 8: System-Dependency Operating System Distribution

general trends for the code bug fixes were that they were small and contained to a single component of the system.

4.4.2 Conceptual

The most common conceptual fix was the fixing of the execution component in Puppet with 44% of the sampled bugs fitting in this category. Additionally, 8% of the bugs were fixed by extending the functionality of the execution component. This means that most bugs were related to the component which executes the configuration operations on the target machine. The second most common component that was fixed and extended was the parser with 19% and 8%, respectively. The rarest categories that are all in single-digit percentages ordered from most to least common are displaying diagnostic messages, changing the configuration, fixing the connectivity component, changing system structure, and changing dependencies.

4.5 System-Dependency

The overall trend in the random sample of bugs analyzed is that bugs in Puppet are mostly system-independent. 17 bugs out of the 100 studied were found to only be present in a specific operating system. The most problematic operating system was found to be Windows with 10 system-dependent bugs assigned to it. The bugs are commonly related to specific issues with the differences in the behavior of the Windows system, for example, in its file system or background services implementation. Other operating systems that were found to have bugs specific to them were Oracle Solaris with 3 bugs, Apple macOS, Fedora Linux, IBM AIX, and Red Hat Enterprise Linux all with 1 bug each. The distributions of system-dependency features are displayed in figures 7 and 8.

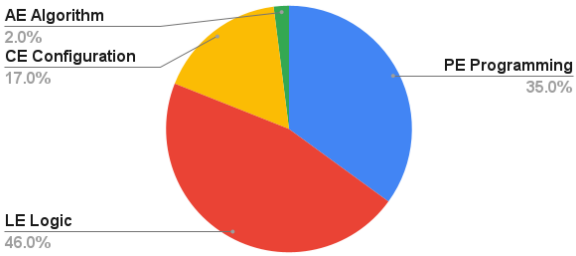

Figure 9: Trigger Cause Distribution

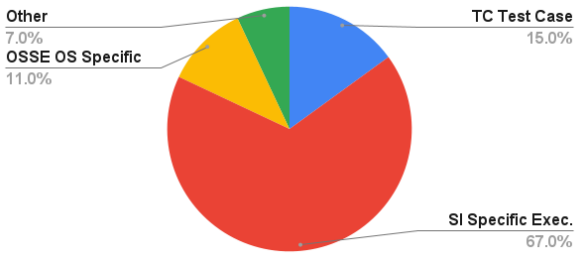

Figure 10: Trigger Reproduction Distribution

The relatively small number of system-dependent bugs found in the Puppet sample compared to other configuration management systems' samples could be a result of the Puppet system being well designed and having high unit test coverage. The way puppet is structured is that it has common components for internal logic which use system-agnostic abstractions. Each unique system Puppet supports has a well-tested abstraction implementation which is then used in the common logic modules. This way common modules only depend on a higher level abstraction and the implementation and behavior details of different systems are handled and aligned in the systems' abstraction layer. For example, interactions with the file system are not direct but are routed through a simple interface that every unique system has a standalone implementation for.

4.6 Triggers

Triggers are categorized in two parts. The first part is the trigger or the reason for the bug's existence, and the second is how to reproduce the bug. The distributions of different trigger causes and reproductions are seen in figures 9 and 10.

4.6.1 Causes

The leading causes of bugs are logic errors with 46% of the bugs falling in this category. These are usually small logical mistakes in the way a condition is written or a case is handled. The fixes to these bugs are similarly small and usually bundled with an accompanying test case that triggers the logic bug. This pattern of generally small bug fixes matches the competent programmer assumption [15]. The next most common category is a programming error with 35%. Programming errors are larger oversights by the implementer of some feature where the programmer misses or implements some part of the system incorrectly. The next most common causes of bugs are misconfigurations with 17% where the application or other defaults are set inappropriately. The least common category is algorithmic errors with 2% of the bug sample where the logic of a bigger feature is fundamentally incorrect.

4.6.2 Reproduction

By far the largest category for reproducing a bug in Puppet is a specific invocation with 67% of the bugs falling under this category and its subcategories. Specific invocation means executing the features of the Puppet system given a specific set of inputs that triggers the bug in it. Usually, this is done by creating a Puppet configuration such that the expectations of the user and the actual system behavior mismatch. Operating system-specific and environment-specific execution sits at 13% of the sample, which roughly lines up with the percentage of system-dependent bugs found in Puppet. Bugs purely triggered by test cases in internal modules make up 15% of the bug sample. Finally, command-line interface usage to

trigger a bug stands at 5%, which can be explained by the relative simplicity of the Puppet client as the bulk of the operational logic is contained in the server and agent components.

5 Responsible Research

Conducting this research in a responsible and reproducible way is a priority to ensure that the findings are a positive contribution to the broader research community. This study follows the main guiding principles outlined in the Netherlands Code of Conduct for Research Integrity [16]. This section discusses the ethical implications as well as the steps taken to ensure that the research is guided by the reproducibility and transparency principles.

To address the concern of proper reproducibility in scientific research [17], additional steps are taken to provide simple and accessible ways to independently verify the findings of the research. The methodology applied is documented as much as possible to provide visibility and clarity into how the data was collected, analyzed, and how conclusions were drawn from it. Any data collection automation or analysis tooling used is open-sourced and publicly accessible for anyone to use for the reproduction and independent verification of the results. The complete data set that was studied is also available for the wider research community to be studied further. Therefore, the reproducibility of this research is high provided that the same method outlined is followed.

To ensure that the research is conducted transparently, a concerted effort is made to highlight both positive and negative results [18]. For example, issues discovered during the data collection process are documented. Potential issues related to the effect of personal bias in the categorization of the bugs are acknowledged. To prevent this bias in the analysis of the data, the analysis stage was conducted multiple times with multiple researchers conducting the categorization and cross-examining, comparing the results, and finally coming to a consensus conclusion, which is then presented in the paper.

6 Discussion

The results of the bug analysis display a clear trend of the types of bugs that are most common in the Puppet configuration management system. Bugs in Puppet generally have the symptom of unexpected runtime behavior. The most common root causes are incorrect target machine operations. The impact level generally is medium with the consequences of target configuration failure. The fixes are usually small method changes in the execution component. Most bugs are generally system-independent. The trigger cause in most cases is incorrect logic and the bugs can usually be reproduced by configuring a specific invocation for the problematic module.

The findings imply that most bugs are in the specific execution components and not the common logic of the system. This means that additional tooling and automation would improve the quality of the execution components and would bring the highest return on investment for Puppet. This could be implemented as an automated checks that require additional test coverage when changing the execution component or linters that check for logic errors. Additionally, approaches for automated fault detection and validation explored in previous research could be extended to incorporate the findings of this study [19, 20]. Finally, methods like fuzz testing could be applied to great effect to discover mishandled execution and parsing failures in the system before the users discover them [13, 14, 21].

6.1 Threats to Validity

In the interest of performing transparent research, the following section mentions potential threats to the validity of the research. These are areas that future research could aim to improve upon.

While the bug collection process of gathering the whole history from the Puppet issue tracker is a simple approach, it might result in some inaccuracies because it relies on the maintainers of Puppet to categorize all bug reports as the "Bug" issue type. Further validation steps could be taken to reduce the number of false-positives and false-negatives in the collected bug sample.

The analysis process has certain inherent drawbacks because of its setup. For example, the analysis is performed manually which is highly time-consuming and therefore resulted in a relatively small sample size that was analyzed. Due to the small sample size, the results might have been slightly skewed. An analysis of a larger sample of bugs would result in higher confidence for the trends observed. Additionally, since categorization can be subjective personal bias might affect the results. This drawback can be mitigated by aggregating the categorizations performed by multiple diverse researchers.

7 Related Work

While there is little literature specifically studying the bugs in configuration management systems, there are publications of research analyzing different types of systems that influenced the setup of this study. Additionally, there are interesting works in the automated discovery of faults in Puppet programs and verification of Puppet configurations. Both of these areas and notable related publications are discussed in more detail in this section.

The basis for the methodology of this bug study is inspired by the research done on typing-related bugs in JVM compilers [1]. The study explored the symptoms, root causes, triggers, and fixes of a random sample of 320 bugs. The sample size is roughly 3 times larger compared to the one analyzed in the Puppet bugs study. A larger sample size gives more confidence about the underlying categorization distribution observed. Another key difference between the studies is the categorization of the bug fix, as the Puppet bug study splits the bug fix category into a code and conceptual fix, which offers a more fine-grained picture of the different types of fixes that are made. The main findings of the JVM bug study are that most bugs manifest as unexpected compile-time errors and that the root causes are usually due to misimplemented type systems.

Other bug studies that are relevant in this context studied numerical bug characteristics [22] and concurrency bugs [2]. These studies applied a comparable bug collection methodology by scraping bugs and fixes from various open sources like Stack Overflow and Github. For comparison, the bug collection sources for the Puppet study were Jira for the bug reports and Github for the bug fixes. Both studies also presented the analysis of comparable sample sizes of 269 and 186, respectively.

The study of change type in bug fixing code [3] formed the basis for the code fix category of the bug analysis process. The main difference between the studies is that the final categorization applied for the bug fix in the Puppet study contained both a code fix subcategory that used a similar taxonomy to the change type study and a novel conceptual fix category that expressed a higher-level perspective on which component of the system was being fixed and how.

Finally, studies exploring novel automated ways of fault detection [19] and configuration verification [20] in Puppet served as the inspiration for what sophisticated automated could be created and studied next, given the results discovered in this study. The practical fault detection in puppet programs study showed that by analyzing the Puppet program and its execution system call trace, it was possible

to uncover 92 previously undiscovered issues in 33 Puppet modules [19]. Additionally, the Rehearsal Puppet configuration verification tool is able to determine the idempotency[2] of Puppet configurations which can then assist in detecting bugs [20]. These are exciting results that give a glimpse into how applying automated fault detection tooling combined with the bug distributions uncovered in this study can improve the quality of configuration management systems.

8 Conclusions

This research analyzed the symptoms, root causes, impact, fixes, system-dependency, and triggers of bugs found in the Puppet configuration management system. An analysis of a collected random sample of 100 bugs showed that the most common symptom of bugs in Puppet is unexpected runtime behavior. Generally, the root causes are incorrect target machine operations. The impact for most bugs tends to be of medium severity and results in failed target configuration. The fixes are generally small method code changes to the execution component of the system. The bugs are rarely system-dependent and are mostly triggered by specifically invoking a module containing a logic error.

These findings can guide further research in configuration management systems and radical quality improvements with the creation and adoption of automated toolings like fault detection [19], validation [20], fuzz testing suites [13, 14], and linters that would target the most bug-prone modules of the system that are the execution and parsing components.

9 Future Work

The results of this study could be further verified by performing additional analysis of a larger bug sample. A more fine-grained analysis of the largest bug categories could also reveal additional takeaways about large classes of bugs. Furthermore, perspectives from other researchers would remediate the effect of the personal bias that might have affected the categorization of the bug sample. Finally, the correlations between categories pictured in figures 12, 13, and 14 could be analyzed more in-depth.

There are exciting areas for future research that could build upon the results discovered in this study. For example, the creation of automated testing suites and linters that could help improve the most bug-prone areas of configuration management systems. Additionally, the results of this study are going to be placed in the context of other configuration management systems and shared with the wider research community as a submission to the 45th International Conference on Software Engineering [7] as discussed in section 1.3.

10 Acknowledgements

Thank you to Mattia Bonfanti, Bryan He, Matas Rastenis, Thodoris Sotiropoulos, Diomidis Spinellis, and Rapole Krupauskaite for reviewing this research and providing feedback.

References

[1] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. Well-typed programs can go wrong: a study of

[2] The property of a configuration to yield the same result even after multiple successive applications.

typing-related bugs in JVM compilers. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[2] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–32, 2020.

[3] Yangyang Zhao, Hareton Leung, Yibiao Yang, Yuming Zhou, and Baowen Xu. Towards an understanding of change types in bug fixing code. *Information and Software Technology*, 86:37–53, 2017.

[4] Wayne Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley, 1986.

[5] Puppet. Code repository for the server automation framework and application, 2022.

[6] Puppet. Infrastructure automation and delivery, 2022.

[7] ICSE 2023 Technical Track, 2022.

[8] Puppet. JIRA tickets, 2022.

[9] Puppet. Pull requests, 2022.

[10] CHAOSS. Send Sir Perceval on a quest to retrieve and gather data from software repositories., 2022.

[11] Mykolas Krupauskas, Mattia Bonfanti, Bryan He, and Matas Rastenis. A study of bugs found in configuration management systems artifacts, 06 2022.

[12] Puppet. How to contribute, 2021.

[13] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[14] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[15] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, F. G. Sayward, and School of Information and Computer Science, Georgia Institute of Technology. Mutation Analysis. *Technical report GIT-ICS-79/08*, pages 4–4, 1979.

[16] KNAW, NFU, NWO, TO2-federatie, Vereniging Hogescholen, and VSNU. Netherlands code of conduct for research integrity. *DANS*, 2018.

[17] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, 2016.

[18] Devang Mehta. Highlight negative results to improve science. *Nature*, 2019.

[19] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. Practical fault detection in puppet programs. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[20] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[22] Anthony Di Franco, Hui Guo, and Cindy Rubio-Gonzalez. A comprehensive study of real-world numerical bug characteristics. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

A Appendix

A.1 Bug Schema

```
CREATE TABLE bugs_fixes (
        id int NOT NULL AUTO_INCREMENT,
        system varchar(100) NOT NULL,
        issue_id int NOT NULL,
        pull_request_id int NOT NULL,
        number int NOT NULL,
        title text,
        body text,
        issue_url text,
        pull_request_url text,
        created_at varchar(100) NOT NULL,
        closed_at varchar(100),
        updated_at varchar(100),
        state varchar(100),
        labels text,
        comments int,
        comments_url text,
        commits int,
        additions int,
        deletions int,
        changed_files int,
        commits_data text,
        bug_report_url varchar(250),
        bug_fix_url varchar(250),
        symptoms varchar(50),
        root_causes varchar(50),
        impact varchar(50),
        consequences varchar(50),
        fixes varchar(50),
        system_dependent varchar(50),
        triggers varchar(50),
        characteristics varchar(50),
        notes text,
        PRIMARY KEY (id)
);
```

A.2 Bug Analysis Tooling

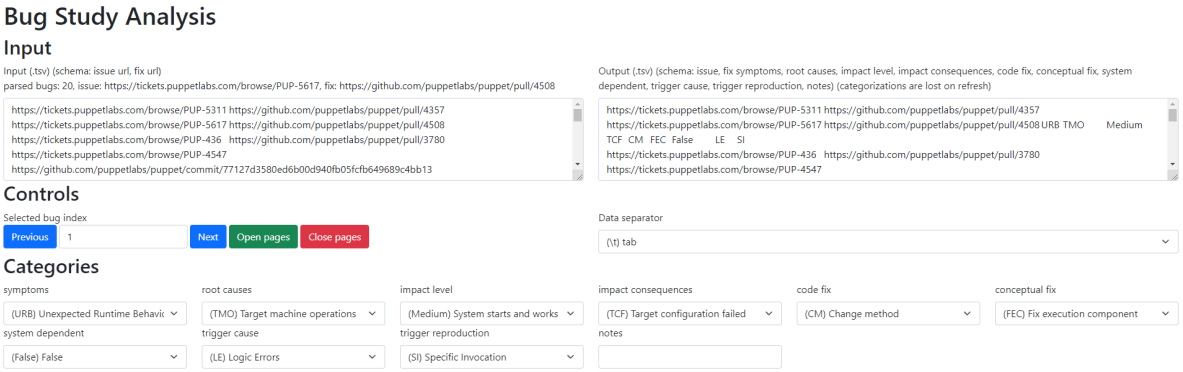

Figure 11: Interactive Web Application to Support the Bug Analysis Process

A.3 Correlations Between Categories

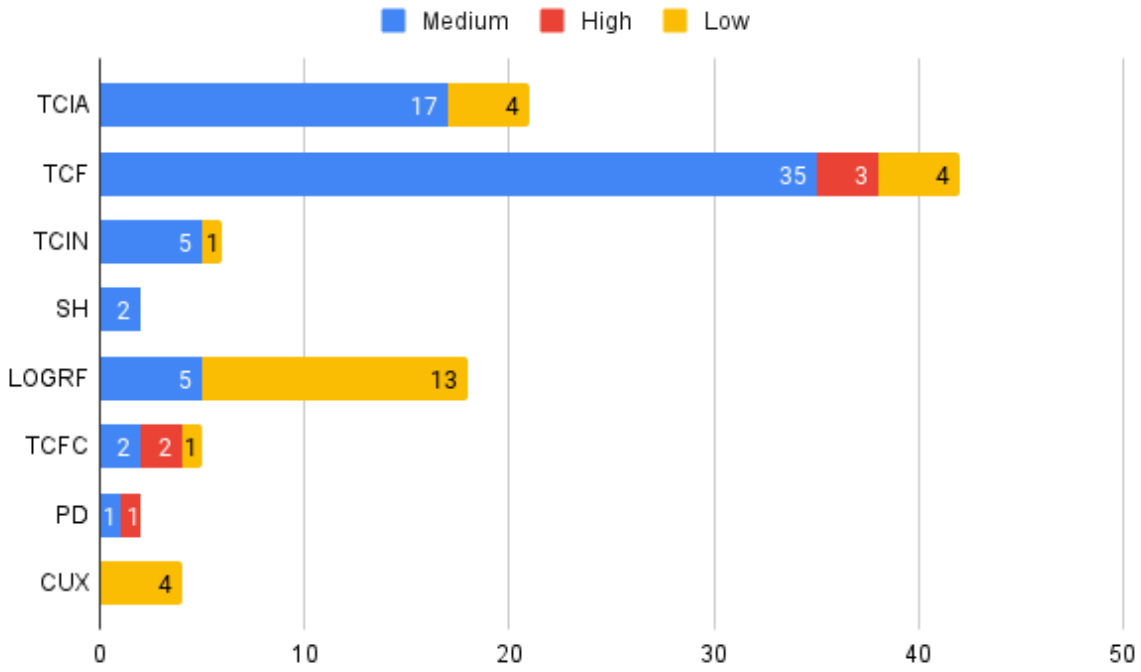

Figure 12: Correlation Between Symptoms and Root Causes

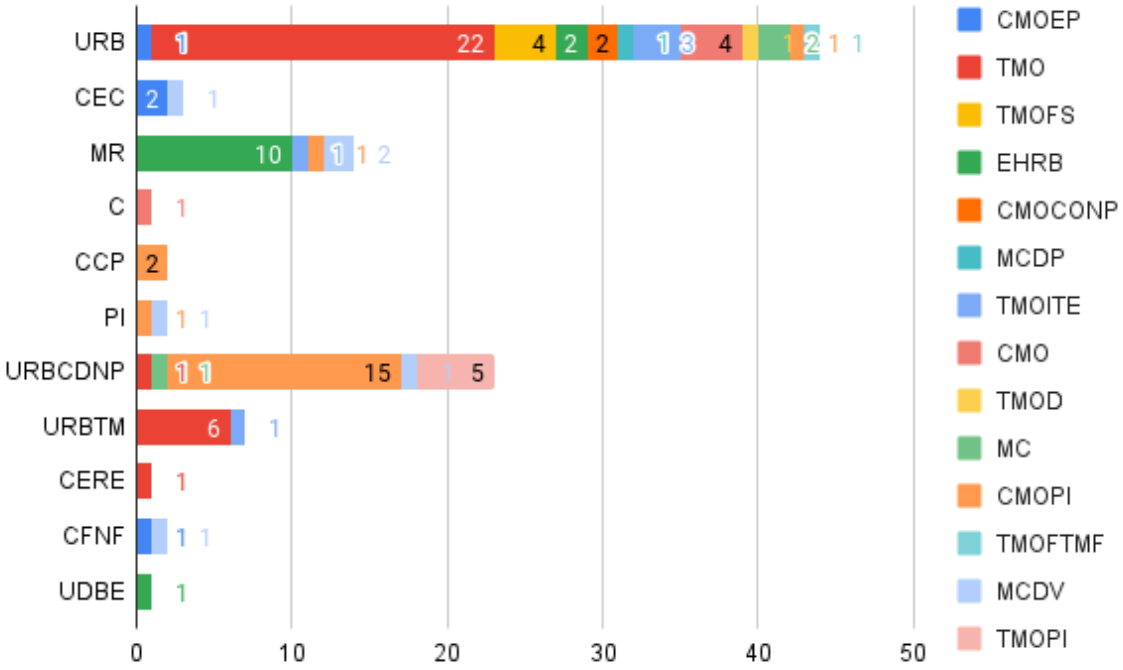

Figure 13: Correlation Between Impact Level and Impact Consequences

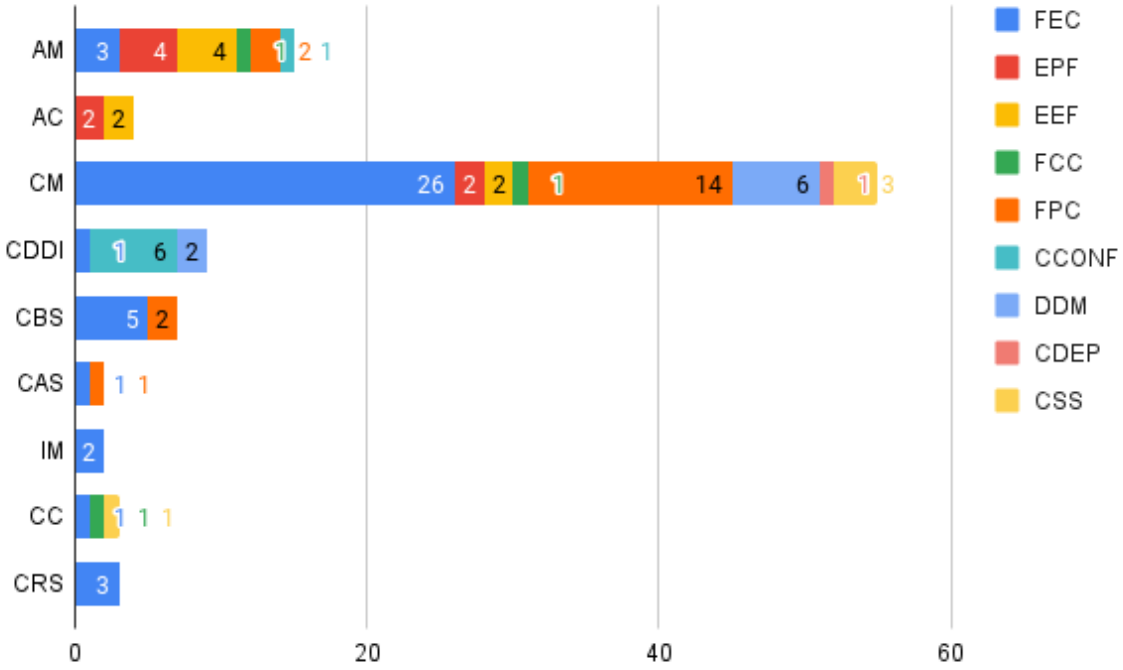

Figure 14: Correlation Between Code Fixes and Conceptual Fixes