An Empirical Study on Bash Language Usage in Github

by

Zheyang Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics (MMath)
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

I hereby declare that I authored or co-authored all the materials included in this thesis: see Statement of Contribution below. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

This project is a joint work with two other PhD students from the SWAG lab: Yiwen Dong and Yongqiang Tian. All algorithms were designed and implemented by the author, Zheyang Li. Yiwen helped with brainstorming research ideas and the manual inspection in the last research question. Both Yiwen and Yongqiang helped with giving suggestion to the writing of the thesis.

Abstract

The Bourne-again shell (Bash) is a prevalent scripting language for orchestrating shell commands and managing resources in Unix-like environments. At the time of writing, it is one of the mainstream shell dialects that is available on most GNU Linux systems. However, the unique syntax and semantic of shell languages could easily lead to unintended behaviors if carelessly used. Prior studies primarily focused on replacing Bash with different languages and there is not much empirical evidence studying the usage of the Bash itself in practice.

In this study, we perform a wide-ranging empirical study of Bash usage, based on an analysis over one million open-source Bash scripts found in Github repositories. We identify and discuss which features and utilities of Bash are most often used. Using static analysis, we find that Bash scripts are often error prone, and the error-proneness has a moderately positive correlation with the size of the script. We also find that the most common problem areas concern quoting, resource management, command options, permissions, and error handling. We envision that the findings of this study can be beneficial for learning Bash and future studies that aim to improve shell and command-line productivity and reliability. In addition, we provide a large dataset of Bash script source code, parse trees and code smell reports of each collected Bash script to facilitate future research in Bash language.

Acknowledgements

I would like to thank my supervisor Chengnian Sun for their guidance and support on this study. I would also like to thank Yiwen Dong and Yongqiang Tian who helped me brainstorm ideas and proofread the writing.

I would like to thank Professor Godfrey and Professor Nagappan for reviewing my thesis and providing feedback.

Lastly I would like to thank my family for their love and support.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

A shell is a program that serves as a command-line interface to an operating system such as Unix; the ability to create scripts using a shell language allows developers to effectively orchestrate the interactions of other command-line tools and to manage system resources. Because of their power and utility shell languages such as `Sh`, `Bash`, and `Zsh` are among the most popular languages in common use; they are among the ten most popular languages in Github, based on the number of unique contributors [12]. However, unlike other popular programming languages such as JavaScript and Python, shell languages are domain specific, designed for interacting with the UNIX environments. Shell languages can be difficult to learn and are well known for having unintuitive syntax and semantics. For example, if a string variable contains whitespace (one of the default input field separators, known as IFS), the retrieval of the variable value would be split by the IFS and becomes multiple words. In Listing 1, the variable `x` does not contain whitespace and the script will correctly execute because no default word splitting will be applied to the variable `x`. However, in Listing 2, word splitting would be applied to variable `x` because of the change from dashes to whitespace characters. It would cause `$x` to be substituted with the literal values `this is a sentence`. Thus, the predicate will become `if [ this is a sentence = "..." ]`, which causes a syntax error when the if statement is executed. The way to avoid such behavior is to double quote the variable `"$x"`, so that the predicate will be interpreted as the comparison between two strings: `if [ "this is a sentence" = "..." ]`.

Listing 1.1: Example of no word splitting

```
#!/bin/bash
x="this-is-a-sentence"
if [ $x = "..." ];
then
```

```
# No error in the if statement...
fi
```

Listing 1.2: Example of word splitting

```
#!/bin/bash
x="this is a sentence"
--if [ $x = "..." ];
++if [ "$x" = "..." ];
then
# Syntax error in the if statement...
fi
```

These two examples of word splitting show one of the many subtleties in shell languages and generally such characteristics can make shell scripting error-prone and hard to maintain.

To combat the error-prone nature of shell scripts, there has been an increasing interests in improving its quality by using static analysis tools. For instance, ShellCheck [17], an open-source static analysis tool in Github with more than 22,000 stars, is designed to find subtle syntactic or semantic issues in shell scripts, supporting a variety of shell dialects. Its popularity implies the high demand in improving the quality of shell scripts.

Further, there are studies such as Bash2py [6] that converts Bash scripts to Python scripts, NL2Bash [19] and NLC2CMD challenge (English to Bash) in the IBM project CLAI [1] that aim to improve shell productivity by utilizing techniques from NLP and machine learning. These studies introduce a way to circumvent the shortcomings of writing shell scripts by converting it to or from different languages that are easier to maintain and debug.

However, these existing efforts do not solve a critical question to improve shell usability: What are the fundamental facts and characteristics of shell language usage? It is currently unclear how and how well developers are using shell language features and utilities in practice. It is also unclear what are the common faults in shell scripts. Without understanding such questions, it will be difficult for researchers to improve the quality of shell scripts. The answers of these questions will give us a better understanding of the current state of shell language usage. It will help developers in writing better shell scripts and help future studies in improving shell and command-line productivity and reliability.

In this study, we aim to address the lack of understanding in the usage of shell language by conducting an empirical study on the usage of shell scripts. More specifically, we focused on Bash, one of the mainstream shells and the default one in many Unix-like systems. We

leveraged the abundant source code from the open-source community by gathering and statically analyzing over one million Bash scripts in Github. Then we followed up with a manual inspection to study the evolution of Bash scripts. We manually inspected 200 bug-fixing commits of Bash scripts and performed thematic-analysis to identify common bugs that developers encounter during the Bash script development.

Overall, we attempted to answer the following research questions:

- **RQ1**: What are the commonly used language features and utilities in Bash scripts?
- **RQ2**: How frequently do code smells occur in Bash scripts? What are the most common code smells?
- **RQ3**: What are the most common bugs that arise in Bash scripts as they evolve?

With the above RQs, we aim to shed some light into the usage of Bash language features and utilities, the Bash script quality and code smells, and the characteristics of common bugs found in the evolution of Bash scripts. We believe having insight of practical Bash usage would be beneficial for new Bash practitioners and future research in command-line tooling and shell productivity and reliability improvements.

Our study has the following major contributions.

1. We empirically identified the common language features and utilities used in Bash scripts.
2. We empirically showed that Bash script sizes and the number of code smells have a moderately positive correlation.
3. We empirically identified the common themes of code smells and bugs in open-source Bash scripts.
4. We curated a large dataset of Bash script source code, their parse trees and ShellCheck code smell reports for future research in Bash language.
5. To ensure reproducibility and to benefit the community, we have made all our data and code publicly available. [1]

[1]Source code and data can be found here: http://bit.ly/37PLx3n

Chapter 2

Background

2.1 History

The term "shell" refers to a command-line interface and a type of scripting languages designed for orchestrating tools and managing resources in the Unix-like environment. It first began with Ken Thompson in Bell Labs in 1971 when Unix was first introduced [21]. There were utilities such as *glob* for wildcard expansion and pattern matching, `if` command to execute statements conditionally, simple redirection, command separator with "`;`" and background processes with "`&`".

The *Bourne shell*, known as sh in the V7 UNIX, was made by Stephen Bourne at AT&T Bell Labs in 1977. The Bourne shell brought the concept of control flows, loops, and variables into shell scripts and it became the inspiration for many later derivative shells. The *Bourne-again shell*, commonly known as Bash, is one such derivative intended to replace the Bourne shell. As a GNU project written in 1988 by Brian Fox, Bash has become one of the mainstream shells and it is the standard shell included in many GNU Linux distributions such as Ubuntu and Fedora. The latest version of Bash, Bash 5.0, was released about 2 years before this writing in January 2019, celebrating 30 years of active development [9].

Despite Bash having decades of history it is an extension to the Bourne shell and it inherits all the shell syntax and semantic that are less-intuitive for backward compatibility purposes. For instance, commands in shell scripts are sensitive to white spaces. This incurs extra development overhead to programmers and it is difficult to debug. In addition, many Bash features involve expansions such as variable expansion, filename expansion, and tilde

expansion. However, the expansion model that specifies the expansion order can be hard to follow and understand once multiple expansions are compounded with each other. These are only the tips of the iceberg that showcase the unintuitive side of Bash. Bash syntax and semantic can easily lead to unintended behaviors if developers are not aware of the nuances.

There have been prior studies addressing the usability issues of Bash as mentioned in chapter 1. There is also an anecdotal list of common Bash mistakes compiled by users in the Bash community called Bash Pitfalls [14]. However, at the time of this writing, we are not aware of any large-scale empirical research studying the usage of Bash language.

2.2 Bash Language Features

Before the inception of UNIX and shell, there was the Job Control Language(JCL), a language designed by IBM for "glueing" mainframe programs in OS360. Similarly, shell languages are similar in that fashion. They are command languages that usually serve as the glue code in the UNIX environments to control the execution of external programs in an interpreted manner. Compared to traditional general-purpose high-level programming languages such C++ and Python that have rich language features and libraries, shell languages have a limited set of features that are domain specific and mostly designed for tasks in the shell environments.

This section briefly introduces some Bash features to facilitate the understanding of the study. More details on the language features can be found in the Bash manual [10].

2.2.1 Parameter Expansion & Special Parameters

Parameter expansion is used to expand the value of a variable. The most basic form is `${var}` where the expansion returns the value of the variable `var`. There are many other parameter expansion rules that perform a variety of substitutions to the variable. For example `${var:=word}` would return the value of `var` if it is not empty, otherwise it assigns the expansion of `word` to `var` and returns the newly assigned value.

There are several constant variables that have special meanings. To name a few, there are `$?` that refers to the exit status of the most recent executed command in Bash, `$0` that refers to the name of the Bash scripts, `$*` and `$@` that expand to all the positional parameters of the command-line arguments used in invoking the scripts.

2.2.2 Pipeline

Pipeline (`|`) is a shell language feature that creates a chain of commands. It is in the form of

$$\texttt{cmd}_1 \texttt{ | } \texttt{cmd}_2 \texttt{ | } \cdots \texttt{ | } \texttt{cmd}_n$$

where the standard output of a command is passed as standard input to the next command in the pipeline.

2.2.3 Redirection

Similar to the pipeline feature, redirection (`>`) redirects the standard input and output of a command and it often deals with UNIX file handles such as standard input, output and error. There are a variety of redirection rules in Bash but the basic redirection rule is in the form of `cmd > file` where the output of `cmd` is redirected as the input to the file `file`, effectively writing the output of `cmd` to the given file.

2.2.4 Command Substitution

Command substitution is in the form of `$(cmd)` where the command `cmd` is run in a separate shell environment and the return value of command substitution is the standard output of the command. For example, `TMP_DIR=$(mktemp -d)` runs the command `mktemp` in a subshell to create a temporary directory and assigns the temporary directory name to the variable `TMP_DIR`.

2.3 Summary

Shell is a both a command-line interface and a type of scripting languages used in the UNIX environments. In this chapter, we have introduced the history of shell and Bash, as well as showing some Bash features.

Chapter 3

Dataset

In this study, we first collected a corpus of approximately 1.3 million Bash scripts from around 510,000 public Github repositories to understand the general Bash usage. To further investigate whether the more popular scripts have different characteristics than the general Bash scripts, we collected an extra corpus of approximately 15,000 Bash scripts from the top 1,000 public shell repositories ranked by Github repository stars. The Github star is a measurement of developer interest. We used the same assumption made in Ray et al.'s study [3] that it is an indication of popularity and we adopted such scheme to identify popular Bash scripts. During the Bash script collection, any entry that appeared in both Bash corpora was only kept in the smaller corpus, ensuring that no duplicate elements exist across the two corpora.

We will refer to the dataset from the general repositories as the **general** dataset and the dataset from the top 1,000 repositories as the **top 1k** dataset.

The overall collection process is illustrated in Figure 3.1. We first collected the Bash scripts and their relevant metadata such as commit history and messages using Github API. For each script, we leveraged IntelliJ's shell parser and ShellCheck to generate parse trees and code smell reports respectively.

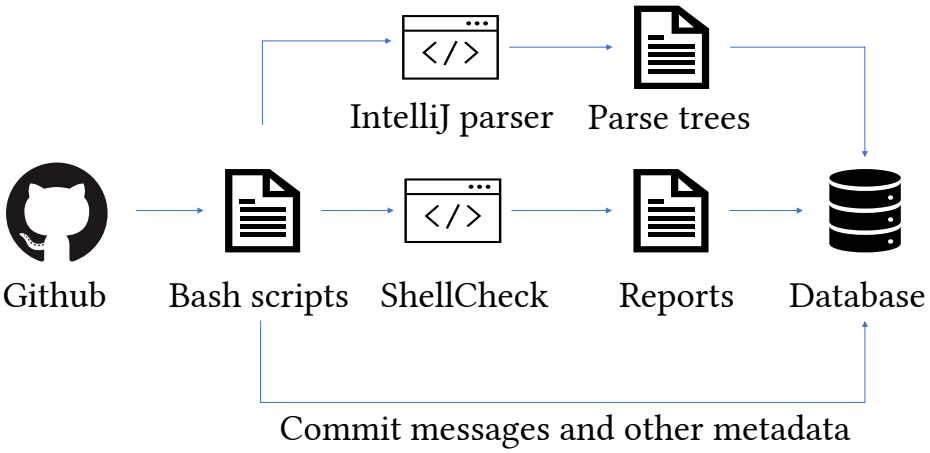

Figure 3.1: Data Collection Process

3.1 Bash Script Collection (Github API)

The General Dataset

We collected approximately 1.3 million Bash scripts using the Github Code Search API[1] with the following parameters:

1. Keyword: **bash**
2. Qualifier 1: **in:file**
3. Qualifier 2: **language:shell**
4. Qualifier 3: **size:n...n+1** where n ranges from 0 to 49999

Github uses the tool Linguist[2] to detect the language of source code file and labels all varieties of shell scripts (e.g., Zsh, Ksh, Bash) as "shell". To ensure that we only have Bash scripts, we used the keyword "bash" in the API and performed filtering locally based on the Bash shebang [3] with the following regular expression:

[1] https://docs.github.com/en/rest/reference/search#search-code
[2] https://github.com/github/linguist
[3] shebang is the first line of code in a shell script that specifies the interpreter

```
^#!.*[\/\\\s](bash)\b
```

To understand the regular expression, it can be broken down into several parts:

- `^#!`: This means the matching string has to start with the characters `#!`.
- `.*`: This means that there can be any number of characters following `#!`.
- `[\/\\\s]`: This is a character group that matches either the `\`, `/`, or any whitespace characters. This is used to cover all kinds of path ending that could lead to the Bash binary.
- `(bash)` : This matches the exact word "bash' in the shebang.
- `\b`: This is a metacharacter that indicates the boundary of the matching string.

The Github API itself also has several limitations:

1. Only the default branch is considered.
2. Only files smaller than 384 KB are searchable.
3. You must always include at least one search term when searching source code.
4. Query returns a maximum of 1,000 results.

Due to the fourth limitation of Github API that it can only return one thousand results per query, we constructed 25,000 queries with each only looking for files within specific file size range in byte(s). In the end, we used the following list of byte ranges: 0-1, 2-3, ..., 49,997-49,998, 49,999-50,000. Github Code Search can return files with size up to 384KB, but we realized during the data collection process that the amount of shell scripts returned per query quickly declines after the query qualifier of file size was set to 10,000 bytes and beyond. We decided to stop our queries at the byte range of 49,999-50,000 bytes to collect a good amount of samples within a reasonable time frame.

The Top 1,000 Dataset

Additionally, we collected approximately 15,000 Bash scripts from the top 1,000 shell repositories in Github ranked by stars.

To identify the top 1,000 shell repositories in Github, we constructed the Github search query using the Github Repository Search API with the following parameters:

1. Qualifier 1: **language:shell**
2. Qualifier 2: **sort=stars**
3. Qualifier 3: **order=desc**
4. Qualifier 4: **per_page=100**

With the above parameters, the Github query returns 100 results per page and we stepped through 10 pages of results to collect the top 1,000 shell repositories ranked by stars. Then we searched locally based on the Bash shebang to identify any Bash scripts in each repository. To ensure there is no overlap, we removed all duplicates in the general dataset.

3.2 IntelliJ Shell Parser and Parse Trees

Listing 3.1: Example of a Bash script

```
#!/bin/bash
echo *.mp3 | grep song
```

IntelliJ IDEA is a popular open-source IDE. It provides an API layer called Program Structure Interface (PSI) that allows developers to create parser plugins for different types of files. IntelliJ's shell parser is one of the many available parser plugins and it supports parsing shell scripts.

We created a Java program that calls the shell parser on each of the collected Bash script. The parse tree output was saved to the database. Listing 3.1 and Listing 3.2 give an example of what a parse tree would look like for a simple Bash script.

Listing 3.2: The Parse Tree of the Bash Script in Listing 3.1

```
[
  {
    "text": "#!/bin/bash\n",
    "element": "shebang"
  },
  {
    "children": [{
      "children": [{
```

```
      "children": [
      {
        "children": [
        {
          "children": [...],
          "text": "echo",
          "element":
          "GENERIC_COMMAND_DIRECTIVE"
        },
        {
          "children": [...],
          "text": "*.mp3",
          "element": "LITERAL"
        }
        ],
        "text": "echo *.mp3",
        "element": "SIMPLE_COMMAND"
      },
      {
        "text": "|",
        "element": "|"
      },
      {
        "children": [
        {
          "children": [...],
          "text": "grep",
          "element":
          "GENERIC_COMMAND_DIRECTIVE"
        },
        {
          "children": [...],
          "text": "song",
          "element": "LITERAL"
        }
        ],
        "text": "grep song",
        "element": "SIMPLE_COMMAND"
      }
      ],
      "text": "echo *.mp3 | grep song",
      "element": "PIPELINE"
    }],
    "text": "echo *.mp3 | grep song",
    "element": "PIPELINE_COMMAND"
```

```
    }],
    "text": "echo *.mp3 | grep song",
    "element": "COMMANDS_LIST"
  }
]
```

3.3 ShellCheck and Code Smell Reports

ShellCheck is a popular open-source static analysis tool for finding subtle issues in shell scripts and it has become the most starred Haskell project in Github. Given a shell script, it can generate a report for each detected issue. For each report, it comes with a severity rating, the location of the issue and an explanation message. The severity rating is classified into four categories: *error, warning, info, style*. The following is a short example of a generated SheckCheck report in JSON format:

Listing 3.3: Example of a ShellCheck Report

```
[
  {
    "file":"...",
    "line":3,
    "endLine":3,
    "column":6,
    "endColumn":6,
    "level":"info",
    "code":2086,
    "message":"Double quote to prevent globbing and
    word splitting."
  }
]
```

Listing 3.4: ShellCheck command

```
shellcheck -f json -s bash -x <file_path>
```

The dataset includes the ShellCheck reports in JSON format for the most recent commit snapshot of each collected Bash script. The ShellCheck reports were generated using ShellCheck version 0.7.0 with the command in Listing 3.4.

3.4 Commit History

Besides all the generated content such as the parse trees and ShellCheck reports, the dataset also includes commit history information on the main branch for each collected Bash script. More specially, the commit metadata includes commit message, commit SHA and commit date.

Listing 3.5: Cloning command

```
git clone --single-branch --no-tag --filter=blob:limit=50k
<clone_url> <clone_dest>
```

For each Bash script, we first cloned their corresponding Github repository using the command in Listing 3.5. Within the cloned repository, we iterated through all commits of a given Bash script and collected the relevant commit metadata.

3.5 Database

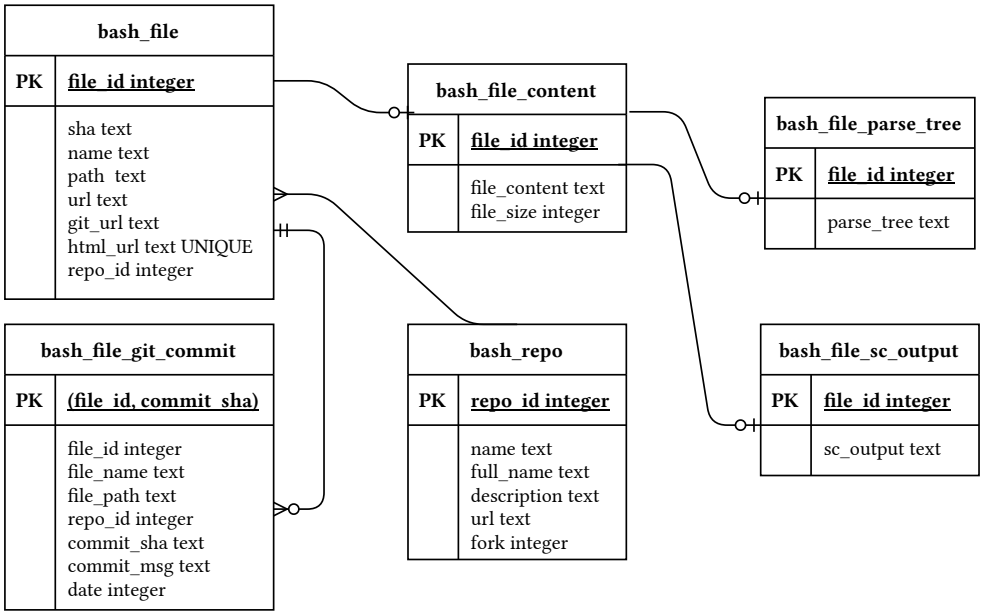

Figure 3.2: Database Schema

Figure 3.2 presents the schema of the database. All the metadata of each Bash script are stored in `bash_file` table. We identified the distinct Bash script based on their file

content SHA. The content of the distinct Bash scripts are stored in `bash_file_content` table. The parse trees and ShellCheck code smell reports are stored as JSON strings in the tables `bash_file_parse_tree` and `bash_file_sc_output` respectively. The table `bash_repo` stores the repositories information and the table `bash_file_git_commit` stores the commit history of each Bash script.

3.6 Data Availability

To ensure reproducibility, all the collected data and scripts mentioned in this thesis are made publicly available and can be found here: https://bit.ly/37PLx3n

Chapter 4

Results

This chapter covers the motivation, approach and the results for each of the research question mentioned.

4.1 RQ1: What are the commonly used language features and utilities in Bash scripts?

4.1.1 Motivation

Two important aspects of a programming language are the available language features and libraries. In this case, Bash is a scripting language that has its own language features and it often serves as the glue code for many utilities in the Unix-like environment. To examine the usability of Bash language, we want to first investigate and identify the core usage of Bash language features and utilities. We believe that such information would be useful for future research in Bash tooling or new Bash practitioners.

For example, command line repair or conversion tools such asBash2Py [6] and NL2Bash [19] can prioritize working on the common language features and utilities to improve its efficiency.

4.1.2 Approach

We approached this question by first defining the scope of language features and utilities, and then generating a parse tree for each collected Bash script from Github. Instead of

Table 4.1: Bash Language Features

pipelines:	`|`, `|&`
lists of commands:	`;`, `&`, `||`, `&&`
compound commands:	`until`, `while`, `for`, `if`, `case`, `select`, `[[...]]`
grouping:	subshell, { `commands` }
function:	⟨function definition⟩
variable:	⟨variable definition⟩
shell parameters:	positional, special, e.g., , `$0`, `$?`
expansion:	brace, tilde, parameter, arithmetic, filename
redirection:	`>`, `>>`, Heredoc, Here-string...
substitutions:	command, process
array:	`array=(value1, value2, ...)`
alias:	`alias name="Hello, world!"`

computing the raw frequency of feature and utility usage from each file, we computed the relative frequency of files that contain certain features and utilities to prevent large or repetitive files from skewing the results.

Language Features. To investigate the usage of language features, we focused on the major language features listed in Table 4.1. These language features were extracted from the Bash 5.0 manual [10].

Utilities. A large numbers of command line utilities exist in the Unix-like systems. To make the study feasible, we limited the scope of utilities by only considering the builtins [24] and the utilities from the GNU coreutils package [11]. Bash builtins are innate functionalities that are implemented inside Bash while utilities from GNU coreutils are external programs. The former includes 57 builtin utilities, such as `echo`, `cd` and `set`, and the latter consists of 102 external utilities, including `rm`, `mkdir` and `cp`. There are three common utilities shared by the two groups, which are `echo`, `test` and `pwd`. In total our study takes 156 ($= 57 + 102 - 3$) utilities. By combining both groups, we believe that it would cover most of the operations needed to develop Bash scripts.

Parse Trees. Once the scope of language features and utilities was defined, we used the IntelliJ shell parser to generate a parse tree for each Bash script and analyzed the language feature and utility usage extracted from the parse tree nodes. To give an example, Listing 3.2 is a parse tree of the Bash script example in Listing 3.1.

By traversing each parse tree, we counted the number of times each language feature and utility that appears. To avoid the scenario of a particular file skewing the usage of certain language features, we counted the number of files that contain the features rather than the raw counts of the features and utilities.

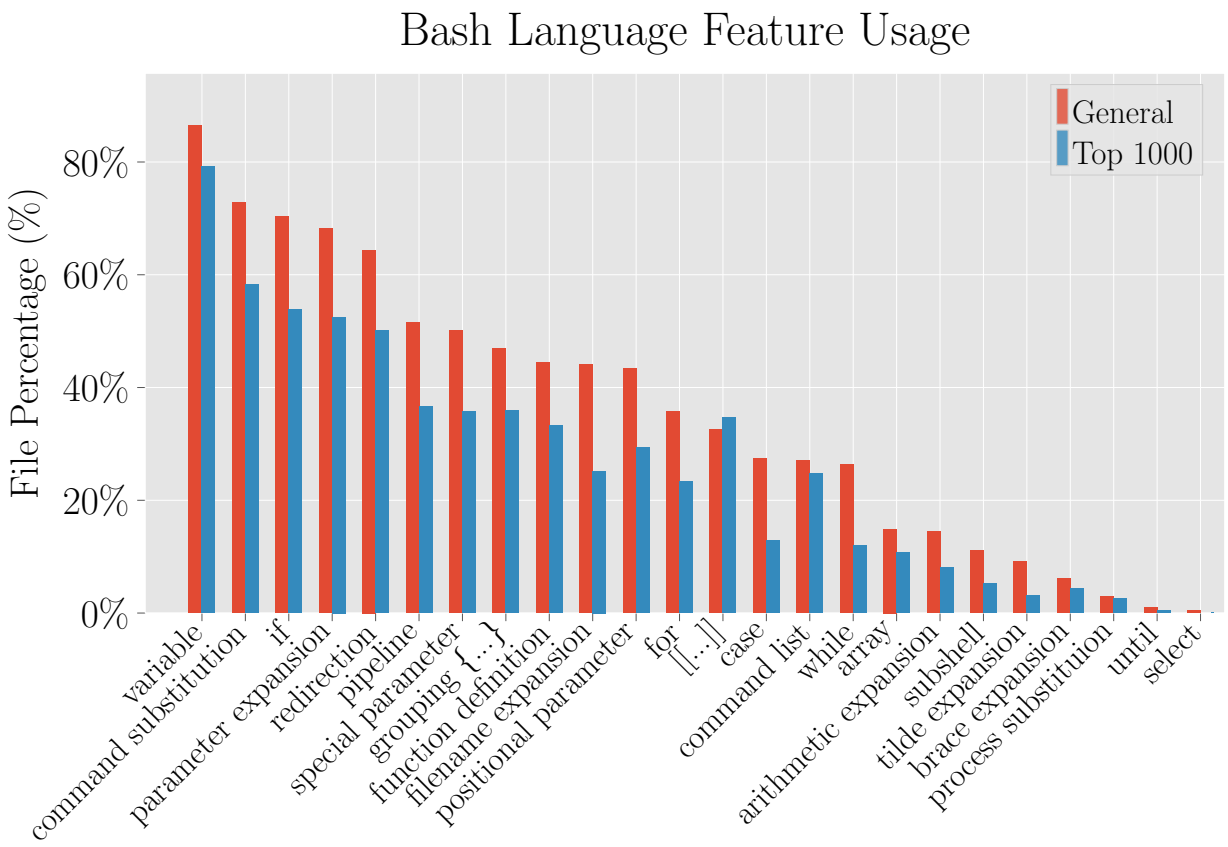

Figure 4.1: The Percentage of Files Containing Each Bash Language Feature.

Majority of the nodes in the parse tree can be directly mapped to one of the Bash language features. However, the few exceptions are pipeline, filename and tilde expansions: Pipeline is a common parent node in the grammar defined by IntelliJ and it is possible that a pipeline contains only one simple command; filename and tilde expansions are only handled after the parsing stage and they do not get expanded in the parse trees. To address the pipeline issue, we only included the pipeline language feature when the pipeline tree node has two or more children in the parse tree, indicating the pipeline contains multiple commands with either `|` or `|&`. To address the issue of filename and tilde expansions, we further parsed the value of terminal nodes in the parse trees and searched for `*`, `?` and `~`.

4.1.3 Results

Language Features

Figure 4.1 plots the usage of each of the selected Bash language features from the general dataset in red and the top 1k dataset in blue. Each bar shows the probability of a feature being used given an arbitrary Bash script. The x-axis labels in Figure 4.1 are the language features and they are sorted in descending order based on their popularity (i.e., the percentage of files that contains such feature) from the general dataset. The following are a few interesting observations based on the results.

Observation 1 — Loop is less commonly used than conditional in Bash scripts. Conditional and loop expressions are both fundamental building blocks that manage the control flow of a program. Based on Figure 4.1, `if` statement has a 70%/58% file occurrence percentage in both datasets while `for` loop and `while` loop only have less than 40%/30% file occurrences respectively. It suggests that Bash developers do not usually work on repetitive tasks and the Bash scripts follow a more linear fashion with only conditional branches.

Observation 2 — Array is not commonly used in Bash scripts. In the original Bourne shell, arrays are not supported. In contrast, one of the additional builtin features from Bash is the support of array. Despite being a builtin feature and supposedly a useful data structure in many other programming languages, a quick inspection on Figure 4.1 suggests that array usage is relatively minimal with around 10%/15% occurrences in the collected Bash scripts.

Observation 3 — The popularity of Bash language features are similar across the general dataset and the top 1k dataset. Although the x-axis labels are sorted based on popularity from the general dataset in red, the language feature usage in the top 1k dataset follows a similar descending order. To measure the differences of the language feature usage among the two datasets, we computed the average differences in file occurrence percentage for each feature. It resulted in each language feature having an average of only 5.6% more usage in the general dataset than the top 1k dataset. Additionally, we inspected and listed the ten most commonly used features from both datasets in Table 4.2. We can observe that the top ten features are almost the same, with the exception that the general dataset has more filename expansions while the top 1k dataset has more Bash conditionals `[[...]]`.

While the language feature usage from the top 1k dataset follows a similar descending order, the usage of Bash conditional `[[...]]` of compound command and command list

Table 4.2: Common Bash Language Features

Rank	General	Top 1k
1	variable	variable
2	command substitution	command substitution
3	if	if
4	parameter expansion	parameter expansion
5	redirection	redirection
6	pipeline	pipeline
7	special parameter	grouping ...
8	grouping {...}	special parameter
9	function definition	[[...]]
10	filename expansion	function definition

are the exceptions. They have a much similar usage in both datasets than the usage of other language features.

Utilities.

Table 4.3 and Table 4.4 show the usage of top 30 Bash builtins and GNU core utilities from both the general and the top 1k datasets. Entries that only appear in one dataset but not the other are highlighted in red.

Observation 4 —**File, path and directory related utilities are more commonly used in Bash scripts.** Table 4.4 shows that the five most commonly used utilities from the GNU coreutil package in both datasets are the same. `rm` deletes a file or directory; `mkdir` creates a directory; `cat` reads the content of a file; `dirname` takes a path and removes the trailing "/" component in the path; `cp` copies a file or a directory to a destination. All of these are utilities that manage files, path and directories.

Observation 5 — **Bash utility usage is relatively similar across the general dataset and the top 1k dataset.** In terms of Bash builtins, the differences in the top 30 popular builtins between the two datasets are minimal. Table 4.3 shows that `alias` and `let` only appeared in the top 30 list from the general dataset while `type` and `hash` only appeared in the top 30 list from the top 1k datasets. Out of these 4 builtins, the highest utility usage is 3% from `alias`, which is much less substantial compared to the rest of the builtin usage. Similarly, the top 30 popular GNU core utilities from both datasets almost

Table 4.3: Top 30 Bash Builtin Usage

General		Top 1k	
builtin	file(%)	builtin	file(%)
echo	77.7	echo	54.8
[	55.9	[	37.2
exit	52.7	exit	33.3
cd	38.1	set	28.8
set	25.5	cd	20.2
pwd	23.2	source	19.5
export	21.1	export	14.1
source	18.1	local	13.1
shift	14.8	.	11.0
read	13.7	pwd	8.9
local	12.2	return	8.7
.	11.8	shift	7.6
return	11.6	printf	7.2
printf	10.2	read	7.1
eval	9.2	exec	5.0
break	7.8	break	4.3
test	7.4	eval	4.2
exec	6.8	declare	4.2
unset	6.5	trap	3.7
trap	5.7	unset	3.3
declare	4.4	command	2.8
pushd	4.4	continue	2.7
getopts	4.2	test	2.5
popd	4.2	pushd	2.3
continue	4.0	popd	2.1
alias	3.2	kill	1.7
kill	2.9	hash	1.6
let	2.7	type	1.6
command	2.5	getopts	1.5
wait	2.5	wait	1.3

Table 4.4: Top 30 GNU Core Utility Usage

General		Top 1k	
utility	file(%)	utility	file(%)
rm	33.1	cat	19.8
mkdir	30.1	mkdir	18.6
cat	28.7	rm	17.7
dirname	23.2	dirname	16.6
cp	23.0	cp	9.4
date	16.6	true	7.5
sleep	12.0	cut	7.5
mv	11.9	basename	7.2
basename	11.6	chmod	7.0
ls	11.5	sleep	6.9
cut	10.3	head	4.6
chmod	9.9	sort	4.4
tr	9.1	mv	4.4
true	7.4	tr	4.3
touch	6.6	touch	4.2
head	6.6	date	4.0
wc	6.4	mktemp	3.9
ln	6.0	uname	3.9
uname	5.5	ln	3.8
tee	5.4	chown	3.6
sort	5.4	readlink	3.4
tail	5.0	wc	3.2
readlink	4.6	ls	2.9
mktemp	3.2	tail	2.7
chown	2.9	id	1.8
hostname	2.7	tee	1.7
seq	2.5	seq	1.6
id	2.5	hostname	1.2
whoami	1.5	uniq	0.9
uniq	1.5	install	0.6

contain the same 30 entries, with the exception of `whoami` and `install`. Overall, the Bash scripts in both datasets exhibit very similar usage of utilities.

4.2 RQ2: How frequently do code smells occur in Bash scripts? What are the most common code smells?

4.2.1 Motivation

As mentioned in Section 1, there exist studies such as Bash2Py [6], NL2Bash [19] and NLC2CMD [1] (English to Bash) that attempt to convert Bash scripts to Python scripts or convert commands from natural language to Bash commands. One of the practicalities of these studies is that it avoids the overhead of developing and maintaining Bash scripts due to its less intuitive syntax and semantics. We want to investigate and gain insights into the general quality of Bash scripts and whether or not having syntactic or semantic issues is a common characteristic.

4.2.2 Approach

To answer the question, we used ShellCheck reports as the measurement of Bash script quality.

As mentioned previously in section 3.3, ShellCheck is an open-source static analysis tool that can catch many syntactic and semantic issues hidden in shell scripts. It first started in 2012 and it has been gaining popularity over time.

Given a shell script, it generates a report for each detected issue. For each report, it comes with a severity rating, the location of the issue and an explanation message. The severity rating is classified into four categories: *error, warning, info, style*. Style related reports are considered free of mistakes as we do not care about stylistic improvement in this study. Listing 3.3 is a short example of a generated SheckCheck report.

One problem we found in using ShellCheck was that the severity rating is not clearly defined among error, warning and info, and there is no available information from its official documentation as far as we are concerned. To mitigate this issue, we applied ShellCheck to each script and treated all reports regarding error, warning and info as potential code

smells. Overall, we ran ShellCheck on the most recent commit snapshot of each collected Bash script. Analysis on the ShellCheck reports was conducted to identify the distribution of report severity and common issues reported.

4.2.3 Results

Prevalence of code smells. The distribution of report severity from both the general and the top 1k dataset is listed in Table 4.5. There are two groups of severity rating that are considered. The first group counts the percentage of Bash scripts that have at least one of error, warning and info reports; the second group counts the percentage of issue-free Bash scripts that only has style reports or empty reports.

Observation 6 — Code smells are prevalent in Bash scripts and it is more prevalent in the general dataset than those in the top 1k dataset. As Table 4.5 demonstrates, the overall data suggests that code smells are prevalent in Bash scripts. More specifically, only 19.1% of Bash scripts in the general dataset have only style or no ShellCheck reports. The data suggest that code smells or potential bugs are quite common in the general population of Bash scripts. In contrast, 46.5% of Bash scripts that are from the top 1k dataset are free from any error, warning or info reports. The results are to be expected because they are more likely to be maintained by multiple developers and potentially have gone through many reviews or static analysis such as ShellCheck before getting committed to their Github repositories.

Themes of code smells. Table 4.5 shows that it is quite common for Bash scripts to have code smells or mistakes that go unrecognized, even if the script is popular and potentially well maintained by multiple developers. In order to understand the details, we pooled together all reports of error, warning and info severity and showed the 5 most common reports from our Bash script corpora in Table 4.6. We further categorized and labeled each distinct report to find common characteristics among them. The bold entries in the tables indicate that they appear in both the general and the top 1k datasets.

Observation 7 — Quoting, word splitting, error handling, array and return value are the common themes of code smells. Table 4.6 shows that the top 5 ShellCheck reports of the two datasets are similar. Four out of five types of reports are the same. By looking at the bold entries of the table, we can see that the common code smells categories are quoting, word splitting, array, error handling and return value.

Correlation between the size of Bash script size and number of code smells. Figure 4.2 shows the correlation between the size of Bash script and the average number of

Table 4.5: Percentage of Bash Scripts with ShellCheck Reports

Severity Rating	General	Top 1k
Error/Warning/Info	1,071,330 (80.9%)	7,707 (53.5%)
Style/None	252,934 (19.1%)	6,699 (46.5%)

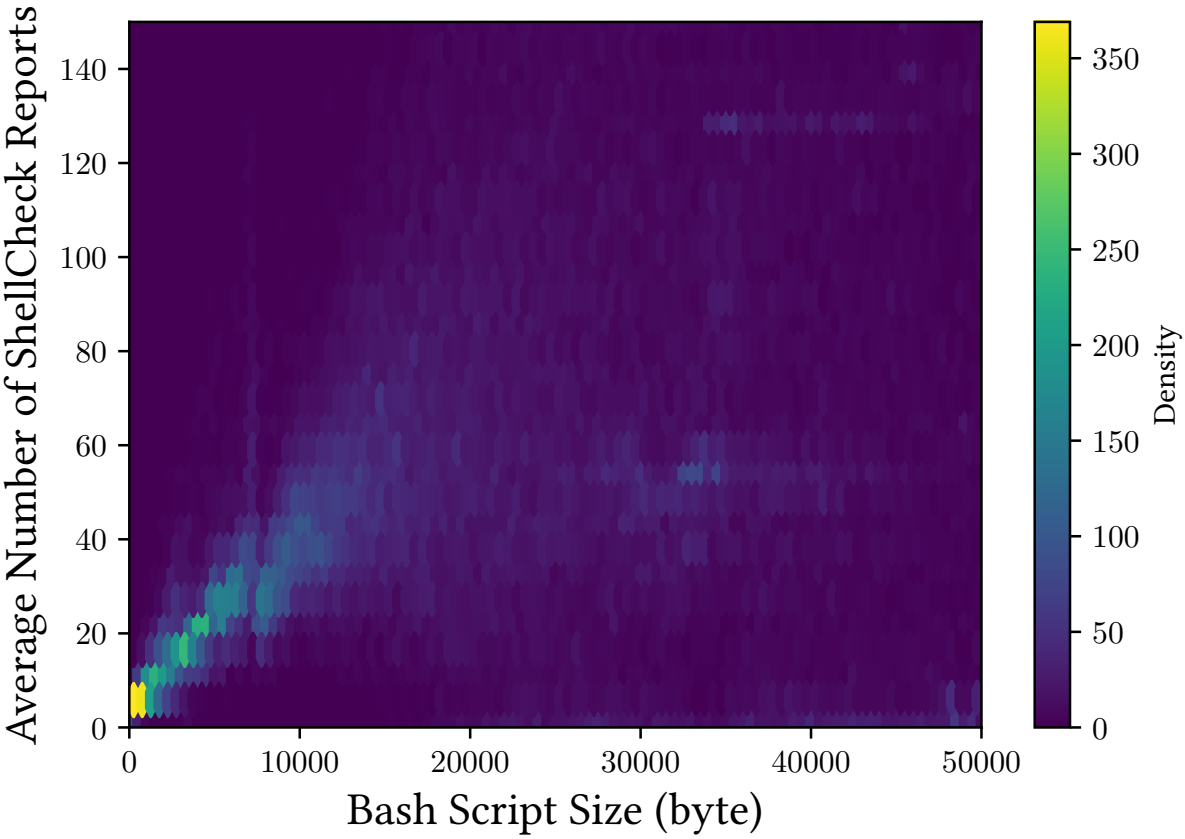

Figure 4.2: Correlation between the Size of Bash Scripts and the Average Number of Code Smells

code smells (i.e., ShellCheck reports) per script size. The x-axis is the size of Bash script in byte and the y-axis is the corresponding average number of code smells. Due to the large size of data points, we applied hexagon binning to show the correlation as well as the

Table 4.6: Top 5 ShellCheck Reports

(a) Top 5 ShellCheck Reports from the General Dataset

Code	Category	Files(%)	Message
SC2164	Error handling	21.5%	Use cd ... —— exit in case cd fails.
SC2046	Quoting, Splitting	20.9%	Quote this to prevent word splitting.
SC2162	Command option	12%	read without -r will mangle backslashes.
SC2155	Return value	8.7%	Declare and assign separately to avoid masking return values.
SC2068	Quoting, Splitting, Array	6%	Double quote array expansions to avoid re-splitting elements

(b) Top 5 ShellCheck Reports from the Top 1k Dataset

Code	Category	Files(%)	Message
SC2046	Quoting, Splitting	9.5%	Quote this to prevent word splitting.
SC2164	Error handling	8.1%	Use cd ... —— exit in case cd fails.
SC2155	Return value	5.3%	Declare and assign separately to avoid masking return values.
SC2128	Array, Expansion	3.2%	Expanding an array without an index only gives the first element.
SC2068	Quoting, Splitting, Array	2.6%	Double quote array expansions to avoid re-splitting elements.

density.

Observation 8 — **The size of Bash script and the number of code smells have a moderately positive correlation.** As shown in Figure 4.2, the number of code smells slowly increases as the script size gets larger. To quantify the correlation, we calculated the Pearson correlation coefficient between the two variables using the following formula:

$$p_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \sigma_Y} \tag{4.1}$$

$Cov(X,Y)$ is the covariance between X and Y, and σ_X and σ_Y are the standard deviation of X and Y respectively. The calculation resulted in a correlation coefficient value of 0.38, which indicates a moderately positive correlation between script sizes and the number of code smells.

4.3 RQ3: What are the most common bugs that arise in Bash shell scripts as they evolve?

4.3.1 Motivation

The results from RQ2 give us a general picture of common mistakes in Bash scripts that are hidden and overlooked. However, they do not include bugs that have been fixed during the script development. To further understand the issues in Bash script usage, it would be useful to look into the evolution of Bash scripts and identify common bugs of which the developers were aware and that were fixed during the development. In doing so, we will have a more holistic view of issues found in Bash scripts and evaluate whether common bugs can be feasibly captured using static analysis tools such as ShellCheck.

4.3.2 Approach

To investigate the common bugs that manifest during the evolution of Bash scripts and identify their characteristics, we randomly sampled bug-fixing commits and manually inspected each of the samples. In total, we studied 200 random bug-fixing commit samples.

Data collection and sampling. To identify bug-fixing commits, a keyword heuristic technique was used on commit messages, similar to the method suggested by Ray et al. [3]. We filtered and identified any commit message that contains any of the following bug-fixing related keywords: *error, bug, fix, issue, mistake, incorrect, fault, defect, flaw, bugfix.*

Due to the large size of the collected Bash script corpora, a total of 200 bug-fixing commits were randomly sampled. More specifically, the first 100 bug-fixing commits were randomly sampled from five selected Bash projects based on their commit history. 20 commits were randomly sampled from each project. Table 4.7 shows all the selected projects, all of which are active in development and have a reasonable amount of stars, contributors and development history. RVM [1] is a command line tool for managing Ruby application environment; devstack[2] is a set of scripts that facilitates the deployment of OpenStack cloud; RetroPie-Setup[3] is a collections of shell scripts that help setup Ubuntu on Raspberry Pi and PC; dokku[4] is a tool to manage the lifecycle of applications; LinuxGSM[5]

[1] https://github.com/rvm/rvm
[2] https://github.com/openstack/devstack
[3] https://github.com/RetroPie/RetroPie-Setups
[4] https://github.com/dokku/dokku
[5] https://github.com/GameServerManagers/LinuxGSM

is a command line tool that facilitates the management of game servers. The rest 100 bug-fixing commits were randomly sampled from the general dataset in which there is no overlap with the previously selected samples.

We adopted such sampling scheme in hopes of achieving a good balance between feasibility and generalization of the inspection.

Manual inspection. To manually inspect each bug-fixing commit, Yiwen Dong and I, who have experience with Bash, did the following tasks:

- Identify and categorize the bug(s)
- Identify how the relevant bug(s) was fixed
- Check whether it can be caught by ShellCheck if possible

We predefined a set of bug categories as a starting point based on a preliminary inspection, with the possibility of adding new categories along the inspection process. Eventually, we came up with the following four major categories.

- **Bash syntactic bug**
  Any bug that is caused by misuse of syntax is considered a syntactic bug. They are often caused by white space, indentation, newline, parentheses, etc.

Listing 4.1: Example of Bash syntatic bug

```
#!/bin/bash
x="this is a sentence."
--if [ "$x" = "..."];
++if [ "$x" = "..." ];
# missing the white space near the closing bracket
then
    # do something here
fi
```

- **Bash semantic bug**
  Bash semantic bugs are the bugs related to the misuse of Bash features and utilities. They are the focus of this inspection.

Listing 4.2: Example of Bash semantic bug

```
#!/bin/bash
--rm /some_folder
++rm -r /some_folder
# missing option -r when working with folder
```

Table 4.7: Sampled Bash Github Projects

Projects	Commits	Stars	Contributors	Dev History
RVM	11,530	4.5k	556	12 years
devstack	10,050	1.8k	640	10 years
RetroPie-Setup	6,752	9k	146	9 years
Dokku	6,529	20.7k	401	8 years
LinuxGSM	5,787	2.6k	144	8 years

- **Application semantic bug**
  Application semantic bugs are primarily related to bugs caused by application logic. Most application bugs are domain-specific but we tried to gain more insights by further providing a few sub-categories such as resource cleaning, file/path/directory management and portability. Any application semantic bug that does not fit into any sub-categories is given the generic sub-category.

Listing 4.3: Example of Application semantic bug

```
#!/bin/bash
--if [[ cmd1 && cmd2 && cmd3 ]];
++if [[ cmd1 || cmd2 || cmd3 ]];
# changes are due to application logic
then
    # do something here
fi
```

- **False positive**
  Lastly, the false positive category denotes the commit is not at all related to any bug-fixing activities (e.g., version update) or the bugs themselves are not in Bash scripts.

All inspections were done individually, then the results were combined and discussed among all the inspectors. Any conflict was discussed and resolved during the process.

4.3.3 Results

By inspecting 200 randomly sampled bug-fixing commits, we were able to identify 57 Bash semantic bugs, 127 application semantic bugs, 2 Bash syntatic bugs and 33 false positives.

Although atomic commits are considered good practices in general, certain commits included multiple bugfixes and we ended up inspecting more than 200 changes. The per group distribution is shown in Table 4.8. Before the major inspection was conducted, we initially came up with a few sub-categories under the category of application semantic bug in hopes of finding common themes in them. However, it turned out that the majority of application semantic bugs were quite generic and domain specific. Overall, we were able to identify several common themes in Bash semantics bugs in our samples.

Table 4.8: Bug Category Distribution

	Bash Semantic	Application Semantic	Bash Syntax	False Positive	Total
General	22	61	2	21	106
Projects	35	66	0	12	113
Total	57	127	2	33	219

Common Bash Bugs. One of the objectives in the study is to identify common Bash bugs. A quick inspection of Table 4.8 tells us that only two bugfixes were related to syntax and seemingly it is rare to have bugs only caused by syntax issues. In contrast, Bash semantic bugs are much more prevalent based on our inspection results. In the 57 Bash semantic bugs, the following are the common themes discovered in their root cause:

Quoting, $9/57 = 15.7\%$. As suggested by the collected ShellCheck reports in Section 4.2.3 that quoting is one of the major themes of code smells and potential causes of bugs, the results from the manual inspection also corroborate the findings from RQ2. A closer look at the inspected bug-fixing samples reveals that Bash developers were having quoting issues with expansions where globbing and word splitting would be performed without quoting, or mixing up single quotes with double quotes. Based on our bug-fixing samples, the former issue could be mostly caught by ShellCheck's extensive quoting checks where double quoting expansions is assumed to be the convention. However, ShellCheck was less effective against the latter issue in our samples. The latter issue seems not as common as the former one and thus it is harder to assume developer's intention.

File, path and directory management, $8/57 = 14.0\%$. Files, paths and directories are resources that developers frequently interact with and manage in their Bash scripts. Based on the inspected bug-fixing samples, it is another common theme of bugfixes during Bash script development. More specifically, most resource bugs revolve around the lack of existence checking of resources. It seems to be the case that developers often assume

the existence of certain static resources in their Bash scripts and the assumption does not always hold in all environments. As far as we are concerned, ShellCheck does not check the existence of resources and it is not able to detect such type of bugs.

Suggestion to Bash practitioners - Adding checks to the existence of static resources can be helpful. Although it is possible that resources are created dynamically in Bash scripts, there are still many usage of statically specified resources. Adding checks to the existence of static resources can help reduce some of the resource management bugs, in which their existences are wrongly assumed.

Command option, 6/57 = 10.5%. Commands are the core of Bash scripts as they provide the means for developers to interact with the operating systems. As part of the commands, options are essential in specifying the desire behaviors. Based on our inspected bug-fixing samples, command option is also a common source of error. These bugs are mostly related to ⓐ the usage of invalid options, ⓑ the improper usage of options. As far as we are concerned, ShellCheck currently is only able to catch very few command option bugs. The improper usage of option often depends on user intention and it is unlikely that static analysis will be able to catch such bugs. However, the usage of invalid options of common utilities can be feasibly caught with static analysis.

Suggestion to static analysis tools - Static analysis tools can incorporate command option checking to reduce the usage of invalid flags. Table 4.3 and Table 4.4 include the popular builtins and GNU core utilities found in the collected Bash scripts. Static analysis can make use of mandb[6] that contains the information of system command options, or creates its command option database to check invalid option usage for common commands.

Permission, 6/57 = 10.5%. Permission plays an important role in the Unix-like systems. Certain commands and resources can only be used when the users are given the sudo/root privilege. Based on the inspected samples, they did not have sudo/root privilege by default and the permission bugs we identified concern missing command permission and unexpected change of resource permissions. ShellCheck currently is not able to catch permission bugs.

Error handling, 6/57 = 10.5%. In Section 4.2.3, one of the major themes in the collected ShellCheck reports is error handling and our inspection on the random bug-fixing samples align well with the previous finding. Several bugs were identified to have chains of commands and developers assumed the success of each command. The general fixes we observed are either adding `|| true` to each command (i.e., `cmd1 || true`) so

[6] https://man7.org/linux/man-pages/man8/mandb.8.html

that command failure would not exit the script, or putting `&&` in between each command (i.e., `cmd1 && cmd2 && ... && cmdn`) so that the subsequent commands are run only if the previous commands have succeeded. To some degree, ShellCheck warns user about potential command failure for certain commands such as `cd`. However, it is command specific and it is quite limited in general.

4.4 Summary

In this chapter, we showed the motivation, approach and result for each of the research question. Specifically, we identified the commonly used language features and utilities, the common code smells and bugs in Bash scripts.

Chapter 5

Related Work

We are unaware of any large-scale empirical studies in Bash usage similar to ours in terms of scope. There are a few studies and open-source tools that focus on debugging and testing Bash scripts. Mazurak et al. [22] developed a static analyzer ABASH to identify common security vulnerabilities in Bash scripts. D'Antoni et al. [5] focused on the usability of command-line and developed a rule-based tooling called NoFAQ to automatically correct problematic commands. Similar to Mazurak [22], Holen [17] developed a pattern-based shell linter called ShellCheck in Haskell that catches popular syntax and semantic issues in Bash scripts and it has gained popularity over time. There is also another open-source tooling named BAT (Bash Automated Testing)[1] that facilitates the testing of Bash scripts.

In addition to Bash tooling and studies, there have been studies that focus on the language feature usage in other programming languages. Dyer et al. [7] generated the abstract syntax trees(AST) over 31,000 Java projects and empirically analyzed their language feature usage. Similarly, Lämmel et al. [18] also employed an AST-based approach and empirically studied the API footprint and coverage in open-source Java repositories. Collberg et al. [4] conducted a large-scale static analysis of Java bytecode from 1,132 java jar-file and collected various metrics regarding the Java feature usage. In addition to Java, Hills et al. [15] studied PHP language features using 19 large open-source repositories with the focus on dynamic language feature. Out of 109 PHP language features, they identified that 80% of files only use 74 language features. Similar to these studies, our work aim to understand the fundamental usage of the shell language, and this study is the first one focusing on Bash language.

[1] https://github.com/sstephenson/bats

Chapter 6

Discussion

6.1 Threats to Validity

6.1.1 Internal Validity

Static analysis tool. In this study, we leveraged the static analysis tool to measure the code quality of Bash scripts. Therefore, the performance of the static analysis tool could affect our results. To mitigate this threat, we selected ShellCheck, which is one of the most popular open-source static analysis tools for shell languages with more than 22,000 stars in Github. It has a long development history since 2012 and is active in development. It is also integrated in some commercial IDE product (e.g., IntelliJ Idea). We believe it provides a reliable ground truth for Bash script measurement.

Bash script analysis. The collected Bash scripts could have gone through ShellCheck and the code smells or bugs could have been fixed, ignored or silenced. Based on the data, we believe that such scenario is not common and it would have minimal impact on our analysis.

Additionally, we conducted a manual inspection in our study. To address the threats in manual inspection (e.g., human error), all inspectors are familiar with Bash and all inspection results were discussed and combined. All the conflicted answers were resolved during the process to minimize human error.

6.1.2 External Validity

Data collection. Regarding empirical studies, data is the new oil. In our study, we collected over one million Bash scripts in hopes of having a more representative study. However, all of our samples came from the same platform Github. Any close source and proprietary Bash scripts were not included in this study, limiting the generalization of the study to some extent. Additionally, we collected our Bash scripts in chunks using the Github API. Due to its limitations, we could only collect 1,000 files per byte range and we do not know the actual file size distribution of Bash scripts in the real world. To mitigate the threat, we collected Bash scripts whose file sizes range from 1 to 50,000 bytes in hopes of covering a wider spectrum and improving the generalization of the study.

6.2 Future Usage of Dataset

Besides answering the research questions, this study also contributes a large dataset of Bash script source code, their parse trees and code smell reports. This study used the dataset to analyze the usage of Bash language features and utilities, common code smells and common bugs during the evolution of Bash script. In addition, we believe that it can also be used for studying the Bash language model, which is not done in the study.

6.2.1 Bash Language Model

A language model is used to predict the probability of sequences of word. It is usually used in the context of natural languages. However, there have been studies focusing on the language models for programming languages. Hinder et al. [16] applies the N-gram model to programming languages and the result suggests that they are generally repetitive and predicable. In addition to using N-gram model, there are also studies that attempt to use machine learning and NLP techniques to learn language models for programming languages. Code2Vec [2] uses a similar idea from word2vec that encodes source code snippets to high dimensional vectors in order to find code similarity. CodeBert [8], based on the pre-trained natural language model BERT, fine tunes the deep neural models on source code snippets. These studies apply more advanced neural models to study traditional programming languages such as C++ and Python. Despite the fact that shell has been among the ten most popular programming languages in Github, many of the programming language model studies do not include shell languages. With the availability of this Bash script dataset, we believe that it enables researchers to study Bash language model.

Chapter 7

Conclusions

The Bourne-again shell, commonly known as Bash, is one of the mainstream shells available in many Unix-like systems. In this study, we presented a large-scale empirical study on Bash language usage in Github. By statically analyzing over one million Bash scripts in Github, we identified the commonly used Bash language features and utilities in the general and popular Bash scripts, showing that both groups share very similar usage. We then studied the occurrences of code smells in Bash scripts with ShellCheck, showing that Bash script is often error-prone and there could be many issues that go unrecognized. Only 20% of general Bash scripts in our samples are free of code smells while 50% of popular Bash scripts in our samples are free of code smells. We also looked into the common themes of code smells, showing that quoting, word splitting, error handling, array and return value are some of the common themes of code smells. Further, we showed that there is a moderately positive correlation between the size of Bash script and the number of code smells found in Bash scripts. Lastly we conducted a manual inspection on randomly sampled bug-fixing commits of Bash scripts. We discovered several common themes of bugs during the evolution Bash scripts, most of which concern quoting, resource management, command option, permission and error handling. With the empirical results from the study, we believe that they can be utilized to help Bash practitioners focus on learning the common language features and utilities while paying more attention to common code smells and bugs. Although shell has decades of history and is one of the ten most popular programming languages in Github, our study suggests that there is still room for improvement in script quality. We envision that the Bash usage insights from our empirical results will be helpful to future research in improvement of shell scripting and command-line productivity and reliability.

References

[1] Mayank Agarwal, Jorge J. Barroso, Tathagata Chakraborti, Eli M. Dow, Kshitij Fadnis, Borja Godoy, Madhavan Pallan, and Kartik Talamadupula. Project clai: Instrumenting the command line as a new environment for ai agents, 2020. arXiv:2002.00762.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290353.

[3] Ray Baishakhi, Posnett Daryl, Devanbu Premkumar, and Filkov Vladimir. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, September 2017. doi:10.1145/3126905.

[4] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007. doi:https://doi.org/10.1002/spe.776.

[5] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 582–592, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3106237.3106241.

[6] Ian J. Davis, Mike Wexler, Cheng Zhang, Richard. C. Holt, and Theresa Weber. Bash2py: A bash to python translator. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 508–511, 2015. doi:10.1109/SANER.2015.7081866.

[7] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014,

page 779–790, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2568225.2568295.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. arXiv:2002.08155.

[9] Free Software Foundation. Bash, 09 2020. URL: https://www.gnu.org/software/bash/.

[10] Free Software Foundation. Gnu bash manual, 12 2020. URL: https://www.gnu.org/software/bash/manual/.

[11] Free Software Foundation. Gnu core utilities, 12 2020. URL: https://www.gnu.org/software/coreutils/.

[12] Github. The 2020 state of the octoverse, 2020. URL: https://octoverse.github.com/.

[13] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2487085.2487132.

[14] Greg. Bash pitfalls, 02 2021. URL: https://mywiki.wooledge.org/BashPitfalls/.

[15] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of php feature usage: A static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 325–335, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2483760.2483786.

[16] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press, 2012.

[17] Vidar Holen. Shellcheck, 2021. URL: https://www.shellcheck.net/.

[18] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, page 1317–1324, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1982185.1982471.

[19] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system. In *LREC: Language Resources and Evaluation Conference*, Miyazaki, Japan, May 2018.

[20] Vadim Markovtsev and Waren Long. Public git archive: A big code dataset for all. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 34–37, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3196398.3196464.

[21] John R. Mashey. Using a command language as a high-level programming language. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 169–176, Washington, DC, USA, 1976. IEEE Computer Society Press.

[22] Karl Mazurak and Steve Zdancewic. Abash: Finding bugs in bash scripts. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, page 105–114, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1255329.1255347.

[23] V. Thakur, M. Kessentini, and T. Sharma. Qscored: An open platform for code quality ranking and visualization. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 818–821, 2020. doi:10.1109/ICSME46990.2020.00101.

[24] Ubuntu. bash-builtins, 2019. URL: http://manpages.ubuntu.com/manpages/bionic/man7/bash-builtins.7.html.