Bash in the Wild: Language Usage, Code Smells, and Bugs

YIWEN DONG*, University of Waterloo, Canada
ZHEYANG LI*, University of Waterloo, Canada
YONGQIANG TIAN, University of Waterloo, Canada
CHENGNIAN SUN†, University of Waterloo, Canada
MICHAEL W. GODFREY, University of Waterloo, Canada
MEIYAPPAN NAGAPPAN, University of Waterloo, Canada

The Bourne-again shell (Bash) is a prevalent scripting language for orchestrating shell commands and managing resources in Unix-like environments. It is one of the mainstream shell dialects that is available on most GNU Linux systems. However, the unique syntax and semantics of Bash could easily lead to unintended behaviors if carelessly used. Prior studies primarily focused on improving reliability of Bash scripts or facilitating writing Bash scripts; there is yet no empirical study on the characteristics of Bash programs written in reality, *e.g.*, frequently used language features, common code smells and bugs.

In this paper, we perform a large-scale empirical study of Bash usage, based on analyses 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 scripts. We also find that the most common problem areas concern quoting, resource management, command options, permissions, and error handling. We envision that these findings can be beneficial for learning Bash and future research that aims to improve shell and command-line productivity and reliability.

CCS Concepts: • **Software and its engineering** → *Language features*; • **General and reference** → **Empirical studies**.

Additional Key Words and Phrases: empirical studies, shell scripts, bash, language features, code smells, bugs

ACM Reference Format:
Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. 2022. Bash in the Wild: Language Usage, Code Smells, and Bugs. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2022), 22 pages. https://doi.org/10.1145/3517193

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

*Both authors contributed equally to the paper.
†Corresponding author.

Authors' addresses: Yiwen Dong, y225dong@uwaterloo.ca, University of Waterloo, Canada; Zheyang Li, z942li@uwaterloo.ca, University of Waterloo, Canada; Yongqiang Tian, y258tian@uwaterloo.ca, University of Waterloo, Canada; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, Canada; Michael W. Godfrey, migod@uwaterloo.ca, University of Waterloo, Canada; Meiyappan Nagappan, mei.nagappan@uwaterloo.ca, University of Waterloo, Canada.

1049-331X/2022/1-ART1 $15.00
https://doi.org/10.1145/3517193

interactions of 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 top ten most popular languages in Github, based on the number of unique contributors [16]. However, unlike other popular, general-purpose programming languages such as C, Java, Scala, Rust, JavaScript and Python, shell languages are domain-specific, designed to facilitate the interactions between humans and operating systems. Shell languages can be difficult to learn and are well known for having unintuitive syntax and semantics [10, 17]. For example, if a variable has a string value which contains whitespaces (whitespaces are the default internal field separators, known as `IFS` [1]), the retrieval of the variable value would be split by the `IFS` and becomes multiple words. In Figure 1a, the variable `x` does not contain any whitespace and the script will correctly execute because no default word splitting will be applied to the variable `x`. However, in Figure 1b, 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 can be interpreted as the comparison between two strings: `if [ "this is a sentence" = "..." ]`. Even so, do not be fooled into thinking quoting once is enough. Suppose in Figure 1c the script checks the output from running a script called `test.bash` inside a directory called `test folder`. The outer quote surrounding `"$(...)"` keeps the output from `test.bash` script from being expanded as seen in Figure 1b. However, without using double quotes also for `"$x"`, the Bash shell behaves unexpectedly as `x` expands into the command `test` with the argument `folder/test.bash` rather than running the file we expected.

```
#!/bin/bash
x="this-is-a-sentence"
if [ $x = "..." ]; # No syntax error
then
  ...

fi
```

(a) Example of no word splitting

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

(b) Example of word splitting

```
#!/bin/bash
x="test folder/test.bash"
-- if [ "$( $x )" = "..." ];   # Silently fails
++ if [ "$( "$x" )" = "..." ]; # Correct usage
then
  ...
fi
```

(c) Example of word splitting in subshell

Fig. 1. Example of syntax error as a result of word splitting when there are space characters in the string assigned to a variable in a Bash script.

Due to Bash's sensitivity to whitespaces, extra care has to be taken for using variables, brackets *etc.*, something that is normally simple in regular general-purpose programming languages. These two examples of word splitting above show one of the many subtleties in Bash and other shell languages. The remainder of this paper will discuss other characteristics/subtleties of Bash. For example, 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 difficult to follow and understand once multiple expansions are compounded with each other. These are only the tip of the iceberg that showcase the unintuitive side of Bash. Bash syntax and semantics can easily lead to unintended behaviors if developers are not aware of the nuances, and all these characteristics can easily make shell scripting error-prone and difficult to maintain.

To combat the error-proneness nature of shell scripts, there has been an increasing interest in improving code quality of shell scripts by using various tools. For instance, ShellCheck [21], an open-source static analysis tool in Github with 22,000+ stars, is designed to find subtle syntactic or simple 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 [10] that converts Bash scripts to Python scripts, NL2Bash [26] and NLC2CMD challenge (English to Bash) in the project CLAI [2] from IBM 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 shell scripts to or from different languages that are easier to maintain and debug.

However, these existing efforts do not answer a critical question to improve the shell quality, that is, what are the fundamental facts and characteristics of shell language usage in reality? 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 code smells or faults in shell scripts. Without understanding such questions, it will be difficult for researchers to improve the quality of shell scripts. The answers to 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 paper, we strive to answer these questions above by conducting a large-scale 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 common code smells?
RQ3 What are the most common bugs that arise in Bash scripts as Bash scripts evolve?

With the above RQs, we aim to shed some light on 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 that having insight of practical Bash usage would be beneficial for new Bash practitioners and for future research in command-line tooling, shell productivity and reliability improvements. From the 1.3 million Bash scripts collected, we found the most commonly used utilities are file, path, and directory related. Yet from ShellCheck, code smells with regards to quoting, word splitting *etc.* are the most commonly reported. Through manual analysis of the bug fixing commits, we determined that quoting, along with file, path, and directory management were both the top sources of error in Bash. We made recommendations to Bash practitioners and future tool developers for ways to reduce the number of bugs in Bash scripts.

This paper makes the following 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) To ensure reproducibility and for benefit of the community, we have made all our data and code publicly available at http://doi.org/10.5281/zenodo.5732299 [25].

2 BACKGROUND

2.1 History of Bash

The term "shell" refers to a command-line interface and a type of scripting language 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 [29]. There were several utilities provided by the shell 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 developed by Stephen Bourne at AT&T Bell Labs in 1977 [23]. 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 [5]. 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 in January 2019, celebrating 30 years of active development [13].

Despite Bash having decades of history and being prevalent in Unix-like systems, it is an extension to the Bourne shell and it inherits all the shell syntax and semantics for the sake of backward compatibility, which are unfortunately unintuitive. For instance, commands in shell scripts are sensitive to whitespaces as seen in Figure 1. This incurs extra development challenges to developers and may easily lead to bugs.

There have been prior studies addressing the usability issues of Bash as mentioned in Section 1. There is also an anecdotal list of common Bash mistakes compiled by users in the Bash community called Bash Pitfalls [18]. We discuss the related work in the Section 7. To the best of our knowledge, there is no any large-scale empirical research studying the usage of Bash language.

2.2 Bash Language Features

Shell programs usually serve as the glue code in the UNIX-like environments to control the execution of external programs in an interpretation manner. Compared to traditional general-purpose high-level programming languages such as C++ and Python that have rich language features and libraries, Bash and other shell languages have a limited set of features that is domain specific and mostly designed for tasks in the shell environment.

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 [14].

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 program in Bash, `$0` that refers the name of the current Bash script, `$*` and `$@` that expand to all the positional parameters of the command-line arguments used in invoking the current script.

Pitfall Example. We showed an example of a common parameter expansion bug in the introduction (Figure 1).

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

Pitfall Example. Error handling with piped commands is tricky as only the exit code of the last command in a pipeline is preserved by default. Even if $\texttt{cmd}_2$ failed, the script could still continue with the developer completely oblivious to the failure.

2.2.3 Redirection. Similar to the pipeline feature, redirection (*i.e.*, <, >, >>) redirects the input and output of a command and it often deals with UNIX file handles such as standard input and standard output. 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 standard input to the file `file`, effectively writing the standard output of `cmd` to the given file.

Pitfall Example. Redirection is error prone as the order of redirection matter. To redirect both standard output and standard error to an file, standard output must be redirected first before redirecting standard error to standard output like so `cmd >file 2>&1`. Otherwise standard output will be redirected to `file` but standard error will be redirected to the original standard output not `file`.

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

Pitfall Example. We showed an example of a command substitution bug in the introduction (Figure 1c).

2.2.5 Filename Expansion. Commonly referred to as *globbing*, filename expansion allows wild card characters (*e.g.* *,?) to be used to match all files that fit a certain pattern. The pattern `*.txt` matches all files in the directory that ends in `.txt`. For loops often use filename expansion to process all files that match a certain pattern (*e.g.* `for file in *.txt ; do`).

Pitfall Example. Filename expansion while useful, can be dangerous because in the case where the pattern does not match any files, the variable `file` in the above for loop example will be set to the string `*.txt` which is not a file that exists. The for loop will be run once rather than skipped over in the case no file matches the expansion pattern.

3 DATA COLLECTION

The overall collection process is illustrated in Figure 2. We first collected the Bash scripts and the relevant metadata such as commit history and messages using Github API. Then for each script, we leveraged the IntelliJ Shell parser and ShellCheck to conduct various analyses. To produce the parse trees, we created a standalone, headless application using the IntelliJ Shell parser that takes in a Bash script and returns a parse tree. Details on the parse tree are discussed in Section 4.2. We

also ran ShellCheck which takes in a Bash script and produces a report with a list of issues found by ShellCheck. Details on the ShellCheck reports are discussed in Section 5.2.

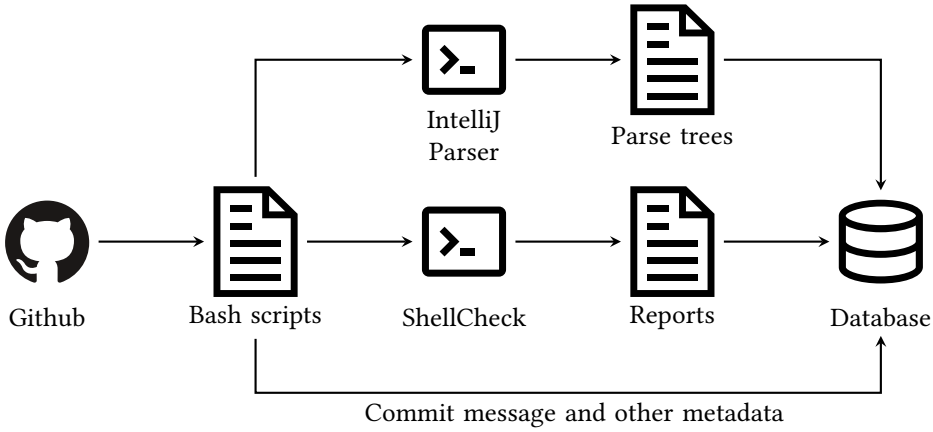

Fig. 2. Data Collection Process

In this study, we collected two corpora of Bash scripts: a general corpus and a top 1K corpus.

General Dataset. We first collected a corpus of 1,352,768 distinct Bash scripts from 516,659 public Github repositories to understand the general Bash usage.

Top 1K Dataset. To further investigate whether the more popular scripts have different characteristics than the general Bash scripts, we collected an extra corpus of 14,387 Bash scripts from the top 1,000 public shell repositories ranked by Github repository stars. The Github star is a measurement of developers' interests. We used the same assumption made in Ray *et al.*'s study [3] that star count is an indication of popularity and we adopted such scheme to identify popular Bash scripts. The Bash scripts were extracted from each repository using the same code search process as discussed in Section 3.1 *without specifying size limits*.

During the Bash script collection process, any entry that appeared in both Bash corpora was only kept in the smaller corpus, ensuring that no duplicate elements exist across the two datasets. Bash scripts with the same SHA hashes were considered as one in the following evaluations and were analyzed once to reduce bias in the results.

3.1 Bash Script Code Search (Github API)

To collect the Bash scripts, we used 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 files 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, *e.g.*, "`#!/bin/bash`".

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

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 declined after the query qualifier of file size was set to 10,000 bytes and above. 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. In the end, we collected 1,352,768 Bash scripts in total, which constitutes a sufficiently large corpus for our studies.

4 RQ1: WHAT ARE THE COMMONLY USED LANGUAGE FEATURES AND UTILITIES IN BASH SCRIPTS?

4.1 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 usage 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 for new Bash practitioners. For example, command line repair/conversion tools such as Bash2Py [10] and NL2Bash [26] can prioritize working on the common utilities to improve their efficiency. Beginners of Bash can prioritize the language features and utilities when learning Bash.

4.2 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 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. That is, if a file contains multiple uses of a language feature or utility, we only count that file once.

Language Features. To investigate the usage of language features, we focused on the major language features listed in Table 1. These language features were extracted from the Bash 5.0 manual [14].

Utilities. A large number 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 [33] and the utilities from the GNU coreutils package [15]. Bash builtins are innate functionalities that are implemented inside Bash while utilities from GNU coreutils are external programs. The former one includes 57 builtin utilities, such as `echo`, `cd` and `set`, and the latter one consists of 102 external utilities, including `rm`, `mkdir` and `cp`. There are three common utilities shared by the two groups, which

Table 1. Bash Language Features

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

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.

Listing 1. Example of a Bash script

```
#!/bin/bash
cat /etc/passwd | sort
```

```
script
├── shebang (#!bin/bash)
└── pipeline (cat /etc/passwd | sort)
    ├── simple_command (cat /etc/passwd)
    │   ├── generic_command_directive (cat)
    │   └── word (/etc/passwd)
    ├── |
    └── generic_command_directive (sort)
```

Fig. 3. The Parse Tree of the Bash Script in Listing 1.

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 illustrate, Listing 1 contains a simple Bash script which prints out sorted lines from the file `/etc/passwd`. Figure 3 shows the parse tree for Listing 1. As seen in the parse tree, the example Bash script is made up of a shebang, and a pipeline joining two commands (`cat` with the argument `/etc/passwd`, and `sort`).

4.3 Results

4.3.1 Language Features. Figure 4 plots the usage of each of the selected Bash language features from the general dataset and the top 1k dataset. Each bar shows the probability of a feature being used in an arbitrary file in the corpora. The x-axis labels in Figure 4 are the language features and they are sorted in descending order based on their popularity (i.e., the percentage of files that

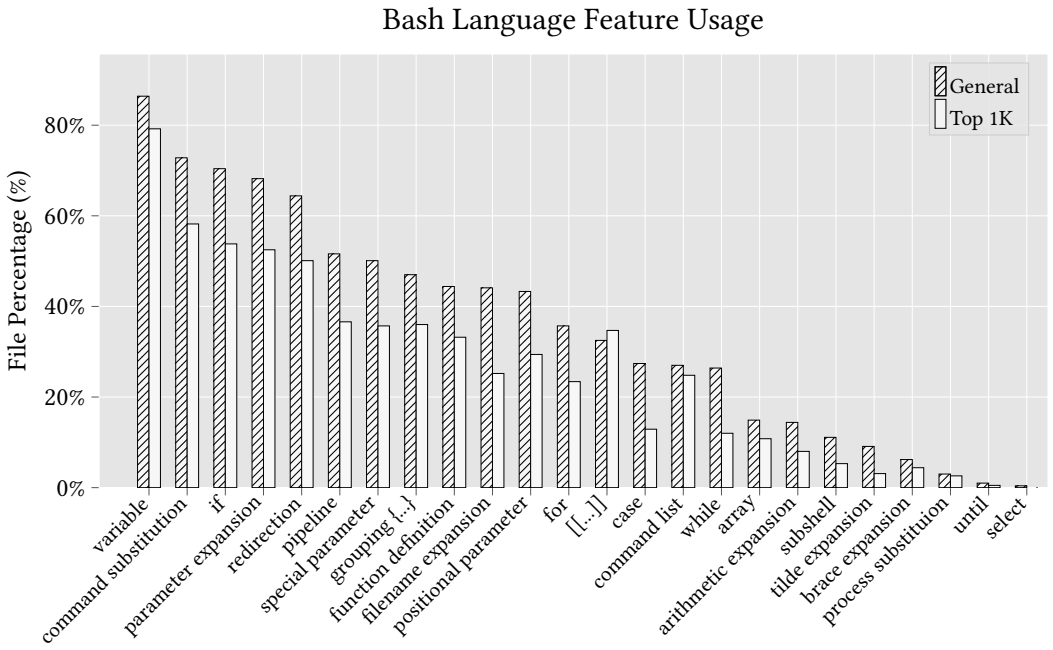

Fig. 4. The Percentage of Files Containing Each Bash Language Feature.

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. Conditional and loop expressions are both fundamental building blocks that manage the control flow of a program. Based on Figure 4, `If` statements have a 70%/58% file occurrence percentage in both datasets while `for` loops and `while` loops only have less than 40% and 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 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, 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 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 `[[...]]` of compound command.

Table 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

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 are the exceptions. They have a much similar usage in both datasets than the usage of other language features, as seen in Figure 2.

4.3.2 Utilities. Table 3 and Table 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 bold font.

Observation 4 —File, path and directory related utilities are more commonly used. Table 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 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 four 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 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.

Conclusions. We ranked the most commonly used Bash features and utilities from 1,352,768 real world Bash scripts. We found that loops and arrays are not commonly used indicating that Bash scripts follow a more linear fashion with only conditional branches. We also observe that file, path, and directory related utilities are more common.

Suggestion to static analysis tools - Static analysis tools should support top language features, builtins and GNU core utilities. Both NL2Bash [26] and Bash2py [10] neglected to analyze redirection even though this Bash feature was used in over 60% of the Bash scripts in our general dataset. In addition, while Bash2py translates many of the Bash builtin commands to their Python equivalents, most of the GNU core utilities were neglected. Translating commonly used GNU core utilities may help with readability and thus maintainability of the converted code.

Table 3. Top 30 Bash Builtin Usage

	General		Top 1k	
	builtin	file(%)	builtin	file(%)
1	echo	77.7	echo	54.8
2	[	55.9	[	37.2
3	exit	52.7	exit	33.3
4	cd	38.1	set	28.8
5	set	25.5	cd	20.2
6	pwd	23.2	source	19.5
7	export	21.1	export	14.1
8	source	18.1	local	13.1
9	shift	14.8	.	11.0
10	read	13.7	pwd	8.9
11	local	12.2	return	8.7
12	.	11.8	shift	7.6
13	return	11.6	printf	7.2
14	printf	10.2	read	7.1
15	eval	9.2	exec	5.0
16	break	7.8	break	4.3
17	test	7.4	eval	4.2
18	exec	6.8	declare	4.2
19	unset	6.5	trap	3.7
20	trap	5.7	unset	3.3
21	declare	4.4	command	2.8
22	pushd	4.4	continue	2.7
23	getopts	4.2	test	2.5
24	popd	4.2	pushd	2.3
25	continue	4.0	popd	2.1
26	**alias**	3.2	kill	1.7
27	kill	2.9	**hash**	1.6
28	**let**	2.7	**type**	1.6
29	command	2.5	getopts	1.5
30	wait	2.5	wait	1.3

Table 4. Top 30 GNU Core Utility Usage

	General		Top 1k	
	utility	file(%)	utility	file(%)
1	rm	33.1	cat	19.8
2	mkdir	30.1	mkdir	18.6
3	cat	28.7	rm	17.7
4	dirname	23.2	dirname	16.6
5	cp	23.0	cp	9.4
6	date	16.6	true	7.5
7	sleep	12.0	cut	7.5
8	mv	11.9	basename	7.2
9	basename	11.6	chmod	7.0
10	ls	11.5	sleep	6.9
11	cut	10.3	head	4.6
12	chmod	9.9	sort	4.4
13	tr	9.1	mv	4.4
14	true	7.4	tr	4.3
15	touch	6.6	touch	4.2
16	head	6.6	date	4.0
17	wc	6.4	mktemp	3.9
18	ln	6.0	uname	3.9
19	uname	5.5	ln	3.8
20	tee	5.4	chown	3.6
21	sort	5.4	readlink	3.4
22	tail	5.0	wc	3.2
23	readlink	4.6	ls	2.9
24	mktemp	3.2	tail	2.7
25	chown	2.9	id	1.8
26	hostname	2.7	tee	1.7
27	seq	2.5	seq	1.6
28	id	2.5	hostname	1.2
29	**whoami**	1.5	uniq	0.9
30	uniq	1.5	**install**	0.6

5 RQ2: HOW FREQUENTLY DO CODE SMELLS OCCUR IN BASH SCRIPTS? WHAT ARE THE COMMON CODE SMELLS?

5.1 Motivation

As mentioned in Section 1, there exist studies such as Bash2Py [10], NL2Bash [26] and NLC2CMD [2] (English to Bash) that attempt to convert Bash scripts to Python scripts or convert commands described in natural languages 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 in reality and whether or not having syntactic or semantic issues is a common characteristic of Bash scripts.

5.2 Approach

To answer the question, we used ShellCheck reports as the measurement of script quality. ShellCheck is an open-source, state-of-the-practice static analysis tool that can catch many syntactic and semantic issues hidden in shell scripts. 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. The following is a short example of a generated ShellCheck report for Listing 1b:

Listing 2. ShellCheck Report for Listing 1b

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

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.

5.3 Results

Prevalence of code smells. The distribution of report severity from both the general and the top 1k dataset is listed in Table 6. 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 are more prevalent in the general dataset than in the top 1k dataset. As Table 6 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 6 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 five most common reports from our Bash script corpora as shown in Table 5. 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 5 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 revolve around the categories of quoting, word splitting, array, error handling and return value.

Table 5. 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.

Correlation between the size of Bash script size and number of code smells. Figure 5 is a plot that visualizes the correlation between the size of Bash script and the average number of 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 [6] to show the correlation as well as the density.

Observation 8 — The size of Bash script and the number of code smells have a moderately positive correlation.. As shown in Figure 5, 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:

$$p_{X,Y} = \frac{Cov(X, Y)}{\sigma_X \sigma_Y} \tag{1}$$

Table 6. 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%)

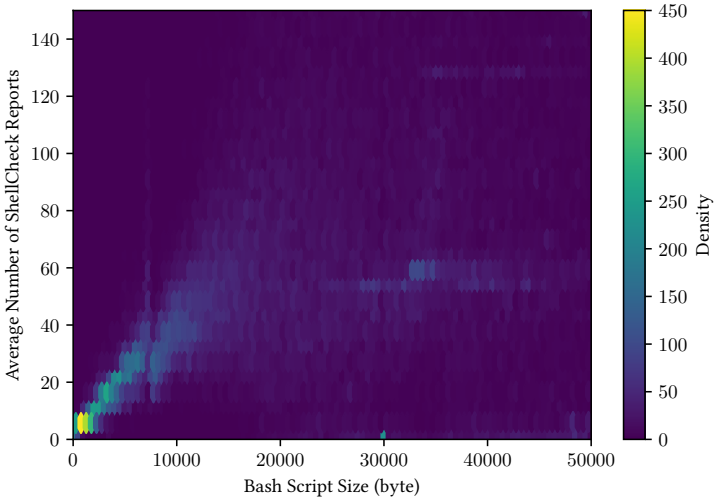

Fig. 5. Correlation between the Size of Bash Scripts and the Average Number of Code Smells. The brighter parts represent higher density for the number of code smells at a certain script size. A moderate positive correlation can be seen from the lighter area extending upward as the Bash script size increases.

$Cov(X, Y)$ 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.

Conclusions. Bash is particularly prone to code smells for development, even among top Bash repositories as observed using ShellCheck. We found that although feature usage is similar between general and top 1k dataset, the general dataset contains more code smells.

Suggestion to Bash practitioners - Practice defensive programming by checking exit status of commands and using quotes where appropriate. Developers should try to reduce code smells by explicitly checking the exit status after each command to ensure correct execution as Bash lacks error handling constructs found in so many mainstream languages (*e.g.* C++, Java, Lisp, Python). Quotes ensure that word splitting would not occur unexpectedly causing one argument to split into multiple. We gave one such example in Figure 1.

Even so, Bash will remain error-prone as common sources for bugs such as quoting and splitting, are also features used by developers. More sophisticated tools need to be proposed in order to find Bash bugs rather than relying on hard coded heuristics. We investigate real world Bash bugs in the following research question.

6 RQ3: WHAT ARE THE MOST COMMON BUGS THAT ARISE IN BASH SCRIPTS AS THEY EVOLVE?

6.1 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 were found and fixed during the script development. To further understand the characteristics of real-world bugs in Bash scripts, it would be useful to look into the evolution of Bash scripts and identify common bugs which the developers were aware of and efforts were spent into fixing. In doing so, we will have a more holistic view of

bugs found in Bash scripts and evaluate whether common bugfixes can be feasibly captured using static analysis tools such as ShellCheck.

6.2 Approach

To investigate the common bugs that manifest in 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 was used on commit messages, similar to the method used in other bug studies [3, 11, 22, 28, 31]. 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.*

A total of 200 bug-fixing commits were randomly sampled. We settled on 200 because the number was similar to existing bug categorization studies [3, 11, 22, 28, 31] to reflect the types of real world bugs in Bash projects. Among the 200 commits, the first 100 ones were randomly sampled from the top five Bash projects based on their commit history where twenty commits were randomly sampled from each project. Table 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[4] is a command line tool for managing Ruby application environment; devstack[5] is a set of scripts that facilitates the deployment of OpenStack cloud; RetroPie-Setup[6] is a collection of shell scripts that help set up Ubuntu on Raspberry Pi and PC; dokku[7] is a tool to manage the lifecycle of applications; LinuxGSM[8] 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.

Table 7. Sampled Bash Github Projects

Projects	Commits	Stars	Contributors	Development 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

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

Manual Inspection. To manually inspect each bug-fixing commit, three authors who were proficient at programming in Bash were asked to do 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

[4]https://github.com/rvm/rvm
[5]https://github.com/openstack/devstack
[6]https://github.com/RetroPie/RetroPie-Setups
[7]https://github.com/dokku/dokku
[8]https://github.com/GameServerManagers/LinuxGSM

The inspectors individually reviewed twenty bug fixing commits per week. Their responses to the above questions were recorded for each bug. After each week, the responses were collected and each bug was discussed. All conflicts were resolved during the discussion to ensure consistency and quality. Eventually we reached a consensus on the above three questions for each bug.

Bug Categories. We predefined four bug categories of interest for inspection. In the end, all manually inspected bugs fitted into one of these predefined four categories.

6.2.1 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 3 below shows an example of Bash syntactic bug.

Listing 3. Example of Bash syntactic bug

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

6.2.2 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 below shows an example of Bash semantic bug.

Listing 4. Example of Bash semantic bug

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

6.2.3 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 5 below shows an example of Application semantic bug.

Listing 5. Example of Application semantic bug

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

6.2.4 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.

Thematic Analysis. We used open card sorting [36] to group the common causes of Bash semantic bugs identified during manual inspection. During the sort, each bug was added to either an existing group or into a new group based on its cause. The groups were split and joined during the card sort as we saw fit. We originally planned to analyze common themes from application semantic bugs as well. However, it turned out that the majority of application semantic bugs were generic and domain specific. We summarized our results in the following section.

6.3 Results

By inspecting 200 randomly sampled bug-fixing commits, we were able to identify 57 Bash semantic bugs, 122 application semantic bugs, 3 Bash syntactic bugs and 32 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 8. Furthermore, all the disagreements were settled easily. Overall, we were able to identify several common themes in Bash semantics bugs in our samples.

Table 8. Bugfix Category Distribution

	Bash Semantics	Application Semantics	Bash Syntax	False Positive	Total
General	22	61	2	21	106
Projects	35	61	1	11	108
Total	57	122	3	32	214

Bash Bug Themes. One of the objectives in the study is to identify common Bash bugs. A quick inspection of Table 8 tells us that only two bug fixes 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:

6.3.1 Quoting (9/57 = 15.7%). As suggested by the collected ShellCheck reports in Section 5.3 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.

6.3.2 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 bug fixes 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.

6.3.3 Command Options (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 desired behaviors. Based on our inspected bug-fixing samples,

command option is also a common source of bugs. 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 options 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 3 and Table 4 include the popular builtins and GNU core utilities found in the collected Bash scripts. Static analysis can make use of mandb[9] that contains the information of system command options, or creates its command option database to check invalid option usage for common commands.

6.3.4 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 revolve around missing command permission and unexpected change of resource permissions. ShellCheck currently is not able to catch permission bugs.

6.3.5 Error Handling (6/57 = 10.5%). In Section 5.3, one of the major themes in the collected ShellCheck reports is error handling and our inspection on the 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 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 limited in general.

7 RELATED WORK

As of the time of writing this paper, we are not aware of any large-scale empirical studies in Bash usage similar to ours. There are a few studies and open-source tooling that revolve around debugging and testing Bash scripts. Mazurak *et al.* [30] developed a static analyzer ABASH to identify common security vulnerabilities in Bash scripts. Our work complements this by identifying error-prone aspects of bash. D'Antoni *et al.* [9] focused on the usability of command-line and developed a rule-based tooling called NoFAQ to automatically correct problematic commands. Similar to Mazurak [30], Holen [21] developed a pattern-based shell linter called ShellCheck in Haskell that catches popular syntax and semantic issues in Bash scripts and it has gained its popularity over time. There is also another open-source tooling named BAT (Bash Automated Testing)[10] that facilitates the testing of Bash scripts. Our work can facilitate development of tools like these to better target real-world problems Bash practitioners face such as permissions and error handing.

Besides Bash tooling and studies, there have been studies that focus on the language feature usage in other programming languages. Dyer *et al.* [12] generated the abstract syntax trees (AST) over 31,000 Java projects and empirically analyzed their language feature usage. Similarly, Lämmel *et al.* [24] also employed an AST-based approach and empirically studied the API footprint and coverage in open-source Java repositories. Collberg *et al.* [8] conducted a large-scale static analysis of Java bytecode from 1,132 java jar-file and collected various metrics regarding the Java

[9]https://man7.org/linux/man-pages/man8/mandb.8.html
[10]https://github.com/sstephenson/bats

feature usage. In addition to Java, Hills *et al.* [20] 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. Compared to them, our study is the first one focusing on Bash language. Our observation noted that there is very little array usage, yet file, path, and directory related commands are very frequently used.

There are a few empirical studies that examined the bugs in different programming languages and the bugs in real-world projects. For examples, previous study analyzed the bugs found in other programming languages such as Rust [31], C and C++ [4]. The same effort in probabilistic language systems led to developments of better fuzzing techniques [11]. Besides, popular projects such as the Linux kernel [7], file systems [27], compilers [32], deep learning systems [34] and others [19, 22, 35], enjoyed attention from researchers providing characterization of bug fixes. Our work falls in this research direction. We focus on Bash script, whose features and bug patterns have not been empirically studied in large scale. We aim to draw researcher attention to Bash language and address the common issues faced by software engineers.

8 THREATS TO VALIDITY

8.1 Internal

8.1.1 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 (22,000+ stars in Github). It has a long development history (from 2012) and is active in development. It is also integrated in some commercial IDE product (*e.g.* IntelliJ Idea). We believe it is a reliable ground truth for Bash script measurement. On the other hand, there is no other good static analysis tools for Bash, a fact which our paper aims to bring to the attention of our software engineering and programming language research community.

8.1.2 Bash script analysis. Although we have collected a good amount of Bash scripts, there could be certain scripts that were automatically generated and followed certain templates or patterns. A large amount of auto-generated Bash scripts could skew and affect the analysis of language feature and utility usage. During our manual inspection, we did not find any auto-generated Bash scripts and we believe the chance of having many auto-generated Bash scripts in our collected samples is minimal.

Additionally, 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.

Lastly, we conducted a manual inspection in our study. To address the threats in manual inspection (*e.g.* bias), all inspectors are familiar with Bash and all inspection results were cross-referenced to minimize human error.

8.2 External

8.2.1 Data collection. 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 for the general dataset and we do not know the actual file size distribution of Bash scripts in the real world. To alleviate

the threat, we collected a broad spectrum of Bash scripts whose file sizes range from 1 to 50,000 bytes.

Our top 1K dataset on the other hand did not suffer from this issue as none of the repositories had over 1,000 Bash files. The results from the potentially biased general dataset and the top 1K dataset in RQ1 and RQ2 were similar, thus this bias may not be a significant source of error. Furthermore, our manual analysis for RQ3 was conducted using the top 1K dataset and does not suffer from the same bias as we are able to collect all the Bash scripts in those repositories.

9 CONCLUSIONS

The Bourne-again shell, commonly known as Bash, is one of the mainstream shells available in many Unix-like systems. In this paper, we presented the first large-scale empirical study on Bash language usage.

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 quite 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 this empirical study, we believe that our results 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 the shell has decades of history and is one of the ten most popular programming languages in Github, our study found Bash scripts to be generally error-prone. We recommend Bash practitioners to program defensively by checking command exit statuses and by quoting variables to prevent unintended word splitting. Further, we suggest Bash practitioners to add checks for the existence of static resources in order to reduce the bugginess of their Bash scripts.

Results show that current static analysis tools are insufficient at analyzing real world Bash scripts and finding Bash bugs. We hope that the Bash usage insights will guide future tools on supporting popular Bash features and utilities. We recommend developers for static analysis tools to address detecting common Bash bugs such as invalid flags. We envision that the insights from our empirical results will be helpful for future research in the improvement of shell scripting, command-line productivity and reliability.

To ensure reproducibility and facilitate future research on Bash scripts, we have made both our collected data set and analysis scripts publicly available at

http://doi.org/10.5281/zenodo.5732299 [25]

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their insightful comments. This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant.

REFERENCES

[1] [n.d.]. *Advanced Bash-Scripting Guide.* https://web.archive.org/web/20210602215627/https://tldp.org/LDP/abs/html/internalvariables.html

[2] Mayank Agarwal, Jorge J. Barroso, Tathagata Chakraborti, Eli M. Dow, Kshitij Fadnis, Borja Godoy, Madhavan Pallan, and Kartik Talamadupula. 2020. Project CLAI: Instrumenting the Command Line as a New Environment for AI Agents. arXiv:2002.00762 [cs.HC]

[3] Ray Baishakhi, Posnett Daryl, Devanbu Premkumar, and Filkov Vladimir. 2017. A Large-Scale Study of Programming Languages and Code Quality in GitHub. *Commun. ACM* 60, 10 (Sept. 2017), 91–100. https://doi.org/10.1145/3126905

[4] Pamela Bhattacharya and Iulian Neamtiu. 2011. Assessing programming language impact on development and maintenance: a study on c and c++. In *2011 33rd International Conference on Software Engineering (ICSE)*. 171–180. https://doi.org/10.1145/1985793.1985817

[5] Stephen R Bourne. 1978. *An introduction to the UNIX shell.* Bell Laboratories. Computing Science.

[6] Daniel B Carr, Richard J Littlefield, WL Nicholson, and JS Littlefield. 1987. Scatterplot matrix techniques for large N. *J. Amer. Statist. Assoc.* 82, 398 (1987), 424–436.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 73–88. https://doi.org/10.1145/502034.502042

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

[9] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs from Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 582–592. https://doi.org/10.1145/3106237.3106241

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

[11] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 574–586. https://doi.org/10.1145/3236024.3236057

[12] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. 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* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[13] Free Software Foundation. 2020. Bash. Retrieved Feb 2, 2021 from https://www.gnu.org/software/bash/

[14] Free Software Foundation. 2020. GNU Bash manual. Retrieved Feb 15, 2021 from https://www.gnu.org/software/bash/manual/

[15] Free Software Foundation. 2020. GNU core utilities. Retrieved Feb 15, 2021 from https://www.gnu.org/software/coreutils/

[16] Github. 2020. The 2020 state of the octoverse. Retrieved Feb 2, 2021 from https://octoverse.github.com/

[17] Michael Greenberg and Austin J. Blatt. 2019. Executable Formal Semantics for the POSIX Shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371111

[18] Greg. 2021. Bash Pitfalls. Retrieved Feb 23, 2021 from https://mywiki.wooledge.org/BashPitfalls/

[19] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. 2015. What Change History Tells Us about Thread Synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 426–438. https://doi.org/10.1145/2786805.2786815

[20] Mark Hills, Paul Klint, and Jurgen Vinju. 2013. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 325–335. https://doi.org/10.1145/2483760.2483786

[21] Vidar Holen. 2021. ShellCheck. Retrieved Feb 2, 2021 from https://www.shellcheck.net/

[22] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[23] M. Jones. 2011. Evolution of shells in Linux. https://web.archive.org/web/20210411144653/https://developer.ibm.com/technologies/linux/tutorials/l-linux-shells/. Accessed: 2021-04-11.

[24] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-Scale, AST-Based API-Usage Analysis of Open-Source Java Projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (TaiChung, Taiwan) *(SAC '11)*. Association for Computing Machinery, New York, NY, USA, 1317–1324. https://doi.org/10.1145/1982185.1982471

[25] Zheyang Li, Yiwen Dong, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. 2022. *Bash in the Wild: Language Usage, Code Smells, and Bugs*. https://doi.org/10.5281/zenodo.5732299

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

[27] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A Study of Linux File System Evolution. *ACM Trans. Storage* 10, 1, Article 3 (Jan. 2014), 32 pages. https://doi.org/10.1145/2560012

[28] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[29] John R. Mashey. 1976. Using a Command Language as a High-Level Programming Language. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) *(ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 169–176.

[30] Karl Mazurak and Steve Zdancewic. 2007. ABASH: Finding Bugs in Bash Scripts. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security* (San Diego, California, USA) *(PLAS '07)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/1255329.1255347

[31] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[32] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

[33] Ubuntu. 2019. bash-builtins. Retrieved Feb 15, 2021 from http://manpages.ubuntu.com/manpages/bionic/man7/bash-builtins.7.html

[34] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[35] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 913–923. https://doi.org/10.1109/ICSE.2015.101

[36] Thomas Zimmermann. 2016. Card-sorting: From text to themes. In *Perspectives on data science for software engineering*. Elsevier, 137–141.