

**Industry Paper****Verification of Shell Script Behavior by Comparing Execution Log**Hitoshi Kiryu<sup>†</sup>, Satoshi Suda<sup>‡</sup>, Shinpei Ogata\*, and Kozo Okano\*<sup>†</sup>Graduate School of Engineering, Shinshu University, Japan  
21w2025g@shinshu-u.ac.jp<sup>‡</sup>Advanced Technology R&D Center of Mitsubishi Electric  
Suda.Satoshi@ay.mitsubishielectric.co.jp\*Faculty of Engineering, Shinshu University, Japan  
{ogata, okano}@cs.shinshu-u.ac.jp

**Abstract** - Scripts written in shell languages including Bash are widely used to automate tasks on UNIX families such as Linux. These scripts, especially those used for automating tasks, are often used continuously even after the operating environment such as OS is updated. The behavior of the shell scripts may change due to OS updates which also usually include commands' upgrades. For this reason, developers must in advance know the changes of the commands used in the script and also check each script in the OS, for every time of the releases. It also produces the cost of debugging scripts and the associated commands. This paper proposes a method to verify if the behavior of a script does not change between two different OS versions. It also detects the cause of the difference. An automated tool for the proposed method is also presented. The proposed method embeds commands which generate execution logs into the scripts and executes those scripts on two different OS versions. The tool compares the generated log files from each OS, and if the behavior is changed, it presents the commands that is cause of differences to the developers. As a result of evaluation of the proposed method, we confirmed that the proposed method can verify the different behavior and detect the commands that cause the difference for a simple example. In addition, the result shows that it was not possible to detect the cause of commands which behavior changes in the area that does not appear in the standard output. In order to detect these commands as the cause, it is necessary to collect logs from a different approach than the standard output.

**Keywords:** Behavior verification, Debug, Execution log, Shell script, Bash

**1 INTRODUCTION****1.1 Background**

Bash [1] is a typical shell language that runs on UNIX OSes such as Linux. It is a POSIX-compliant Bourne Shell language with extended history and aliasing features, and it is mainly adopted as default login shell in many UNIX OSes. These scripts written in shells are widely used in corporate business systems

for automating tasks. The behavior of shell scripts written in the Bash may change due to OS updates and associated command upgrades. For Instance, because of commands upgrades, it is possible to lose access to environment variables, which are a set of variables used to configure the shell and various commands. It is explained in Section 3. Therefore, developers must understand the commands which are executed in the scripts and the changes in the OS specifications when the OS is upgraded. It produces a lot of costs for debugging scripts and the associated commands.

In this paper, we propose a method to support developers in solving problems that occur when updating the OS by verifying whether the behavior of Bash scripts is equivalent between two different version OSes and detecting the cause of the inequality. Although there are many distributions of Linux, in this paper, we focus on CentOS, which has been used for business systems in companies.

**1.2 Related Work**

Various studies [2]–[5] have been conducted to localize bugs related to these problems. These studies proposed methods to identify statements that cause bugs by using bug reports, trace information, visualization. These studies aim to fix and identify bugs, while our purpose is to identify the causes of the differences in behavior.

A similar role to Bash scripts is played by Dockerfile that describes a series of procedures to build containers. In the field of micro-services architecture, services based on Docker are promising. The services are defined by Docker files, and the files sometimes contain complex scripts. The analysis of Dockerfile has been studied particularly from the viewpoint of developer support. For instance, Kaisei Hanayama and his co-authors [6] have suggested a method to create a code-completion tool for writing Dockerfiles is proposed by using machine learning of Dockerfiles on Github. In another study [7], a survey of Self-Admitted Technical Debt (SATD) in Dockerfile was conducted for an actual DockerHub project. From the survey, patterns of SATD in Dockerfile and their proportion are revealed.

In addition, CBMC [8] and JBMC [9] which is based on CBMC are software model checking tools for verifying the be-

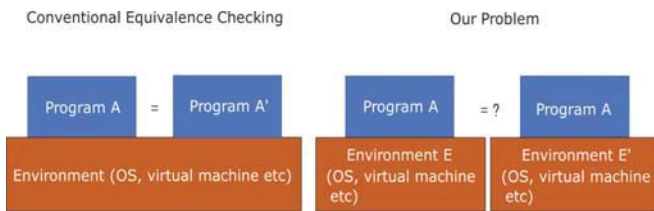


Figure 1: Differences from our previous study

havior of programs. These tools take a program and assertions as input and use bounded model checking [10] to verify that the assertions are valid. Although it is possible in theory to verify the behavior using these tools, they are not tools for directly checking changes of behavior, which is the focus of this paper, since they are for verifying whether the behavior is as specified.

The author’s group has been researching behavioral equivalence verification of programs [11]–[13]. In these studies, we verified the differences in the behavior of modified programs as shown on the left side of Fig. 1. The problem that we will address in this paper is the difference in the behavior in changes in the environment, as shown on the right side of Fig. 1. Therefore, the methods used in these studies cannot be applied to this problem.

The authors did not find any research on the behavioral equivalence of Bash when the environment changes. Therefore, it would be useful for developers to create a tool to verify whether a script behaves equally across operating systems. Bash has a debugging tool [14], but these tools do not work with inter-pipeline output and scripts called within a script.

### 1.3 The Approach

In the proposed method, we first embed commands which generate execution log in script files to be verified. The log generating commands (Loggers) are programs that log standard output, output between pipelines, and variable assignments. Scripts with Loggers embedded are executed to generate and retrieve the logs. This process is performed on each of the two target operating systems built on VirtualBox [14], and the obtained logs are compared to detect differences in behavior. If the logs match, it is assumed that there is no change in the behavior. If the logs do not match, we present where the logs differ as the commands which is cause of the difference (hereinafter called “cause command”) to the developer. The information to be presented is executed command, execution log, and stack trace information of the function or script.

We conducted three evaluations of the proposed method. In the first evaluation, we intentionally created commands with different behaviors and experimented to detect the difference in behaviors and to identify the cause. In the second evaluation, we experimented to see if the proposed method can detect the difference in the behavior of a script containing the command `sudo`, whose behavior was changed by updating the command

in the past and to identify the cause of the change. In the third evaluation, we applied the proposed method to a script, which access given URL via a proxy server imitatively downloading files in secured network.

As a result of our evaluation experiments, we confirmed that our proposed method is helpful for developers in verifying whether behaviors changed. We also found that it is possible to detect differences in the behavior of commands with small side-effects, e.g. commands that only calculate and return arguments. However, it is difficult to directly identify the cause of the differences in commands with side-effects, such as commands that affect the environment variables.

In the following sections, Section 2 describes the techniques related to this research, such as bug localization and lexical and syntactic analysis. In Section 3 we describe the motivating example, and in Section 4 we describe the proposed method. In Section 5 we present the results in the evaluation experiments and in Section 6 we discuss the results in the experiments. Finally, we conclude in Section 8.

## 2 PRELIMINARIES

### 2.1 Bug Localization

Various methods are being researched to identify software bugs. In the paper [15], a method to identify statements with bugs from the program spectrum which is execution records of success or failure of software test cases is studied. It identifies the statement as the most suspicious statement that includes bugs. The suspicion of statement is calculated from program spectrum information using newly proposed metrics. Another research [16] proposes a method to support identifying defects by visualizing data transitions and execution flow with dynamic analysis from Java source code. This research targets to detect functional defects in logic and visualizes their details the detailed processing flow and dependencies of variables to assist users in understanding. In evaluation experiments on the tool, it was confirmed that the use of the tool can reduce the time required to locate defects.

### 2.2 Model Checking

Model checking is a method to verify that the behavior of a system satisfies the specification. A model checking tool judges whether or not a specification is valid for a state transition model that represents the behavior of a system. In many model checking tools, specifications are described by temporal logic such as CLT and LTL. There is a study [17] that applies model checking to real system development. In this study, a model checking tool UPPAAL [18] was used for the development of medical infusion pumps to formalize the models and specifications, verify the safety of the models and generate code from the verified models. Also, since the description of the state transition model and the specification is essential for model checking, the research [19] has been conducted to support the description of the model and



Figure 2: Outline of lexer and parser

the specification for engineers who do not have such knowledge. Automatic generation of checking models from the state transition table of the system and the table which summarizes the actions which occur in each state.

### 2.3 Lexical and Syntactic Analyzer

Lexical and syntactic analysis is a series of parsing processes in programming languages, as shown in Fig. 2. A series of pipelined syntactic analyzers check that the token sequence given by the lexical analysis satisfies the defined grammar. We use PLY (Python Lex-Yacc) [20] as a lexical and syntactic analyzer to embed Loggers into Bash scripts given as input. PLY is an implementation of Lex, a lexical analyzer for Python, and Yacc which is a syntactic analyzer. In the lexical analysis part, regular expressions are used to define the string-to-token conversion rules, and in the syntactic analysis part, the LALR(1) grammar [21] is used to define the syntax rules. A syntax tree is constructed from the given rules and parsed for a given sentence.

## 3 THE MOTIVATION EXAMPLE

Listing 1: foo.bash

```
1 VAR=' baz '
2 export VAR
3 sudo bash bar.bash
```

Listing 2: bar.bash

```
1 echo ${VAR}
```

The two Bash scripts in the Listings 1 and 2, foo.bash and bar.bash, are example scripts whose behavior changes depending on OS versions. In CentOS5 and later versions, the behavior of the sudo command in foo.bash has changed due to the upgrade of the command. Versions that are older than CentOS5 output the string “baz,” while in CentOS5 and later versions not output a blank line. This is because, in 1.6 and earlier versions of the sudo, the command was able to preserve environment variables when executed by sudo, however, in 1.7 and later versions, it is necessary to specify the ‘-E’ option to preserve environment variables. The environment variable is not referred due to the change of the command specification of the newer version. This change causes the difference in behavior between versions older than CentOS5 using version 1.6 bash and later versions of CentOS.

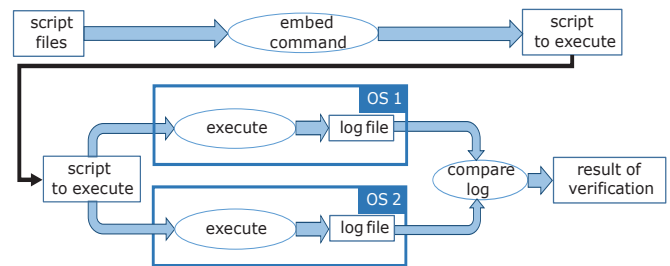


Figure 3: Outline of the proposed method

## 4 THE PROPOSED MOTHOD

The proposed method generates execution logs of Bash scripts and judges the difference in behavior by comparing the logs. The execution log contains the standard output, the output between pipelines, and the variable assignments along with the executed commands. The stack trace information of the command is also stored to make it easier to identify the cause.

### 4.1 Outline of the Proposed Method

The schematic diagram of the proposed method is shown in Fig. 3. The input and output of the proposed method are shown below.

- Input : Script file to be verified
- Output: verification results (the behavior is different and which command behaves differently)

The procedure of the proposed method is as follows.

(1) It embeds the Logger into the Bash script given as input. The Logger refers to a program that logs the standard output, output between pipelines, and variable assignment. The details of each program are described below.

(2) It executes script embedded commands on each of the two operating systems.

(3) It compares the obtained logs and judges whether the behavior is different. If the behavior is different, the command that causes the difference and its log information is presented.

We have created a tool that automatically executes the above procedure.

### 4.2 Execution Log

The format of the execution log is shown in Listing 3.

Listing 3: Abstraction of the log

```
1 <log identifier>:<commands>, line: <
   lineno>, stack: <stack trace>
2 <output log>
3 :<log identifier>
```

1. <log identifier> : “assignment” if the log is an assignment, “command” if the log is a command execution

2. <commands> : Executed commands
3. <line> : line number of executed command
4. <stack trace> : Stack trace information of the script and function when the command is executed.
5. <output log> : Assignment log and standard output log.

### 4.3 Log Generating Command(Logger)

The following four Loggers are embedded in the script. The behavior of each is shown below.

- Standard output log command : Generate standard output log
- Assignment Log command : Generate Variable Assignment Log
- Stack push command : Record Stack trace information
- Stack pop command : Record Stack trace information

#### 4.3.1 Standard Output Log Command

This command logs the standard output and the output between pipelines. It also takes two arguments, a command to be executed and its line and logs the string to be executed. The standard output log command is “stdout\_logger” in Listing 5. This logging command can generate the standard output log by pipelining this command to a line that does the normal standard output as shown in the examples in Listings 4 and 5.

Listing 4: Before embedding example for standard output

```
1 echo hello
```

Listing 5: After embedding example for standard output

```
1 echo hello | stdout_logger 'echo
  hello' 1
```

The log shown in Listing 6 is generated by executing the script shown in Listing 5.

Listing 6: Log example for standard output

```
1 command:echo hello, line: 1, stack:
2 hello
3 :command
```

Also, the output between pipelines can be logged like the standard output by embedding it as Listings 7 and 8.

Listing 7: Before embedding commands for pipeline

```
1 commandA | commandB
```

Listing 8: After embedding commands for pipeline

```
1 commandA | stdout_logger 'commandA' |
  commandB
```

#### 4.3.2 Assignment Log Command

In this step, we generate log of assignments to variables. The assignment log command is “assign\_logger” in Listing 10. It takes the assignment command to be executed, the variable name, and the value of the variable as arguments and records them in the execution log. When the assignment command is taken as an argument, single quotes are escaped to avoid the expansion of variables. The Listings 9 and 10 show an example of before and after embedding the log.

Listing 9: Before embedding commands

```
1 VAR=' baz '
```

Listing 10: After embedding commands

```
1 VAR=' baz '
2 assign_logger 'var='\'' baz '\'' ' VAR "
  $VAR" 1
```

Running the script ex.bash that executes Listing 10 will generate the log shown in Listing 11.

Listing 11: Log example for assignment

```
1 assignment:VAR=' baz ', line: 1, stack:
2 VAR= baz
3 :assignment
```

#### 4.3.3 Stack Push, Stack Pop Command

In order to identify the execution path of commands, command to stack push and pop records stack trace information into a text file. At the start of the script or function, their name is pushed. At the end of that, the pushed name is popped. In addition, the call command string is pushed just before the script or function call, and the pushed string is popped after the calling. Listings 14 and 15 shows the result of embedding for ex.bash and test.bash shown in Listings 12 and 13.

Listing 12: ex.bash before embedding commands

```
1 test.bash 'test'
```

Listing 13: test.bash before embedding commands

```
1 echo ${1}
```

Listing 14: ex.bash after embedding commands

```
1 push_stack ex.bash
2 push_stack 'test.bash '\'' test '\'' '
3 test.bash 'test'
4 pop_stack
5 pop_stack
```

Listing 15: test.bash after embedding commands

```

1 push_stack test.bash
2 echo ${1} | stdout_logger 'echo ${1}'
  1
3 pop_stack

```

Listing 14 generate the log shown in Listing 16.

Listing 16: Log example for stack and push command

```

1 command:echo ${1}, line: 1, stack: ex
  .bash->test.bash 'test'->test.bash
2 test
3 :command

```

The log “ex.bash->test.bash ‘test’->test.bash” which is the stack trace information following “stack:” in the log, indicates that the log was generated in test.bash called by test.bash ‘test’ from within ex.bash.

#### 4.4 Embedding Commands

Embed Loggers into scripts given as input. The tool embeds each command in the following cases.

- Start of the script or function : Stack Push Command
- End of the script or function : Stack Pop Command
- The line just before the call of script or function : Stack Push Command
- The line just after the call of script or function : Stack Pop Command
- Pipeline : Standard Output Log Command
- Variable Assignment : Assignment Log command

#### 4.5 Executing Scripts

Scripts which is embedded Loggers are executed. In running the script, copy it to OS built using VirtualBox and run. Verification can be done within a single machine.

#### 4.6 Comparing Logs

The tool compare 2 execution logs generated by running. Execution log contains following information.

- Executed command
- Stack trace information
- Log identifier of standard output or assignment
- Log of standard output or assignment

In case that no difference between the logs is detected, the tool judge that behavior is consistent. If not so, the tool judge that behavior is changed and display different logs as causes.

#### 4.7 Implementation

We create the tool based on the proposed method. The tool gets scripts as input, embed commands, run embedded scripts, compare logs, and display results automatically.

The Loggers were embedded using a lexical and syntactic analyzer written in Python Lex-Yacc. It embeds Loggers if the script matches defined grammars. For example, if assignment operator “=” appears in a single command, it is recognized as an assignment and the assignment log command is embedded. The four Loggers and the log comparison program were implemented using C++.

Target Oses of verification are built on VirtualBox on Windows. Scripts given as input are embedded Loggers on Windows and copied to target Oses, then the execution logs are provided by running the scripts on each OS. The logs are copied to the Host OS and verified. By executing the script on target Oses using SSH from the Host OS, this method is executed automatically.

### 5 EVALUATION EXPERIMENT

In order to evaluate the proposed method, we conducted evaluation experiments for the following three scripts to see if it is possible to detect and identify the cause of different behaviors.

1. A script that executes commands created to behave differently between operating systems.
2. A script containing the sudo command described in Section 3
3. A script that imitates a situation of building environments via proxy

The environment and OS used for these evaluation experiments are as follows.

- Host OS : Windows10 Pro
- VirtualBox ver 5.2.18 r124319
- Guest OS 1: CentOS 4.6
- Guest OS 2: CentOS 8.2.2004

Both experiments followed steps below.

1. Embed commands into scripts given as input on the Host OS.
2. Copy the Scripts which is embedded commands to the target Oses.
3. Execute the scripts on each target Oses and copy generated logs to the Host OS.
4. Compare the obtained logs and get verification results.

## 5.1 Experiment 1

We prepared a command “sample” and a simple script to execute the command for each OS. The command “sample” takes two integer arguments and outputs the result of addition on CentOS4.6 and multiplication on CentOS8.2. The proposed method is applied to the scripts we created and conducted experiments to evaluate whether the tool can detect a difference in behavior between scripts with a different function, and whether “sample” can be identified as the causative command.

The script used for the experiment, “ex.bash,” is shown in Listing 17.

Listing 17: ex.bash

```
1 result=$(sample 2 3)
2 echo ${result}
```

Script after embedding the commands into ex.bash is shown in Listing 18.

Listing 18: ex.bash after embedding command

```
1 push_stack ex.bash
2 result=$( sample 2 3 | stdout_logger
   'sample 2 3' 1 )
3 assign_logger 'result=$( sample 2 3
   )' result "$result" 1
4 echo ${result} | stdout_logger 'echo
   ${result}' 2
5 pop_stack
```

Obtained logs by executing the above script on each target OSes are shown in Listings 19 and 20.

Listing 19: Log on CentOS4.6 in Experiment 1

```
1 command: sample 2 3, line: 1, stack:
   ex.bash
2 5
3 :command
4
5 assignment:result=$( sample 2 3 ),
   line: 1, stack: ex.bash
6 result=5
7 :assignment
8
9 command: echo ${result}, line: 2,
   stack: ex.bash
10 5
11 :command
```

Listing 20: Log on CentOS8.2 in Experiment 1

```
1 command: sample 2 3, line: 1, stack:
   ex.bash
2 6
3 :command
```

```
$. /bin/check_diff.exe logs/centos4.log logs/centos8.log
log is different
command:
  sample 2 3, line: 1, stack: ex.bash
in logs/centos4.log
5
in logs/centos8.log
6

log is different
assignment:
  result=$( sample 2 3 ), line: 1, stack: ex.bash
in logs/centos4.log
  result=5
in logs/centos8.log
  result=6

log is different
command:
  echo ${result}, line: 2, stack: ex.bash
in logs/centos4.log
5
in logs/centos8.log
6

The logs are different
```

Figure 4: Result of Experiment 1

```
4
5 assignment:result=$( sample 2 3 ),
   line: 1, stack: ex.bash
6 result=6
7 :assignment
8
9 command: echo ${result}, line: 2,
   stack: ex.bash
10 6
11 :command
```

The results of the comparison of the two logs are shown in Fig. 4.

Different logs are suggested. According to the results in Fig. 4, differences in the script behavior were detected. The cause of the different behavior of the script “ex.bash” is the command “sample,” and the first presented log shows the command “sample 2 3.” Therefore, the difference in behavior was detected and the command that caused the difference was identified.

## 5.2 Experiment 2

The proposed method is applied to scripts shown in Listings 1 and 2, which are indicated in Section 3 and conducted experiments to evaluate whether the tool can detect differences in the behavior of the scripts and identify the command “sudo” as the cause of the difference.

The Bash scripts after embedding the commands into the scripts are shown in Listings 21 and 22.

Listing 21: foo.bash after embedding command

```
1 push_stack foo.bash
2 VAR=' baz '
3 assign_logger 'VAR='\'' baz '\'' ' VAR "
   $VAR" 1
```

```

4 export VAR | stdout_logger 'export
  VAR' 2
5 push_stack 'sudo bash ./bar.bash'
6 sudo bash ./bar.bash
7 pop_stack
8 pop_stack

```

Listing 22: bar.bash after embedding command

```

1 push_stack bar.bash
2 echo ${VAR} | stdout_logger 'echo ${
  VAR}' 1
3 pop_stack

```

logs generated by executing the scripts shown in Listings 21 and 22 on each target OSes are shown in Listings 23 and 24.

Listing 23: Log on CentOS4.6 in Experiment 2

```

1 assignment:VAR='baz', line: 1, stack:
  foo.bash
2 VAR=baz
3 :assignment
4
5 command:export VAR, line: 2, stack:
  foo.bash
6 :command
7
8 command:echo ${VAR}, line: 1, stack:
  foo.bash->sudo bash ./bar.bash->
  bar.bash
9 baz
10 :command

```

Listing 24: Log on CentOS8.2 in Experiment 2

```

1 assignment:VAR='baz', line: 1, stack:
  foo.bash
2 VAR=baz
3 :assignment
4
5 command:export VAR, line: 2, stack:
  foo.bash
6 :command
7
8 command:echo ${VAR}, line: 1, stack:
  foo.bash->sudo bash ./bar.bash->
  bar.bash
9
10 :command

```

There is a difference in the 9th line of each log: CentOS4.6 outputs “baz,” but CentOS8.2 outputs an empty string. This is the same as the result mentioned in Section 3.

The results of the comparison of the logs are shown in Fig. 6. From the results in Fig. 6, the difference in behavior is detected

```

$ ./bin/check_diff.exe logs/centos4.log logs/centos8.log
log is different
command:
  echo ${VAR}, line: 1, stack: foo.bash->sudo bash ./bar.bash->bar.bash
in logs/centos4.log
  baz
in logs/centos8.log
The logs are different

```

Figure 5: Result of Experiment 2

by comparison of the execution logs. However, the command “echo \$VAR” suggested as a cause is not true cause as described in Section 3. The true cause command “sudo” wasn’t identified as a cause of the difference in the behavior of the script.

### 5.3 Experiment 3

The following script shown in Listing 25 updates packages with package managing command “yum” via a proxy server, which is described by environment variables including “http\_proxy.” Some enterprises and institutions often create proxy servers to protect internal networks from cyberattacks, and users in organizations access external networks via proxy servers. A tool to manage packages like “yum” in the script is necessary command for building server environment. This script imitates such a situation.

Listing 25: Example script of updating command via a proxy server

```

1 http_proxy="http
  ://192.168.56.1:3128/"
2 https_proxy="https
  ://192.168.56.1:3128/"
3 ftp_proxy="ftp://192.168.56.1:3128/"
4 export http_proxy https_proxy
  ftp_proxy
5 sudo yum update -y

```

In this Experiment, We applied the method to a script that is the essence of the script in Listing 25. The script extracts the title of the given URL “http://example.com” with the curl command via a given proxy server. In order to evaluate that the tool can detect differences in behaviors and identify cause command. This script just extracts the title of a given URL via proxy server.

Listing 26: test.bash

```

1 export http_proxy="http
  ://192.168.56.1:3128/"
2 echo "title is" $(sudo curl -sS "
  example.com" 2>&1 | grep -Po "(?<=
  title>)(.+)(<=/title>)"

```

The script embedded command is shown in Listing 27.

Listing 27: test.bash after embedding command

```

1 push_stack test.bash
2 export http_proxy="http
  ://192.168.56.1:3128/"
3 assign_logger 'http_proxy="http
  ://192.168.56.1:3128/' http_proxy
  "$http_proxy" 1
4 echo "title is" $( sudo curl -sS "
  http://example.com" 2>&1 |
  stdout_logger 'sudo curl -sS "http
  ://example.com"' 2 | grep -Po
  "(?<=title>)(.+) (?=</title>)" |
  stdout_logger 'sudo curl -sS "http
  ://example.com" 2>&1 | grep -Po
  "(?<=title>)(.+) (?=</title>)"' 2 )
  | stdout_logger 'echo "title is "
  $(sudo curl -sS "example.com"
  2>&1 | grep -Po "(?<=title
  >)(.+) (?=</title>)"') 2
5 pop_stack

```

Listings 28 and 29 show generated logs on the each OSes.

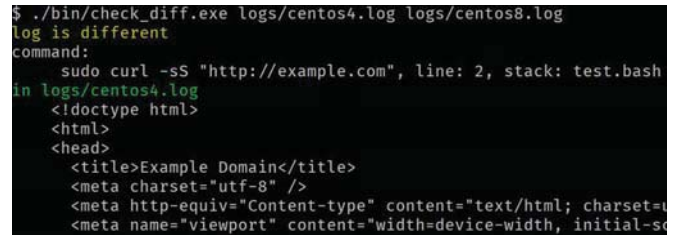
Listing 28: Log on CentOS4.6 in Experiment 3

```

1 assignment:http_proxy="http
  ://192.168.56.1:3128/", line: 1,
  stack: test.bash
2 http_proxy=http://192.168.56.1:3128/
3 :assignment
4
5 command: sudo curl -sS "http://
  example.com", line: 2, stack: test
  .bash
6 <!doctype html>
7 <html>
8 <head>
9   <title>Example Domain</title>
10   (Omission)
11 </html>
12 :command
13
14 command: sudo curl -sS "http://
  example.com" 2>&1 | grep -Po "(?<=
  title>)(.+) (?=</title>)", line: 2,
  stack: test.bash
15 Example Domain
16 :command
17
18 command: echo "title is" $(sudo curl
  -sS "example.com" 2>&1 | grep -Po
  "(?<=title>)(.+) (?=</title>)",
  line: 2, stack: test.bash
19 title is Example Domain

```

Listing 29: Log on CentOS8.2 in Experiment 3



```

$ ./bin/check_diff.exe logs/centos4.log logs/centos8.log
log is different
command:
  sudo curl -sS "http://example.com", line: 2, stack: test.bash
in logs/centos4.log
<!doctype html>
<html>
<head>
  <title>Example Domain</title>
  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=
  <meta name="viewport" content="width=device-width, initial-s

```

Figure 6: Result of Experiment 3

```

1 assignment:http_proxy="http
  ://192.168.56.1:3128/", line: 1,
  stack: test.bash
2 http_proxy=http://192.168.56.1:3128/
3 :assignment
4
5 command: sudo curl -sS "http://
  example.com", line: 2, stack: test
  .bash
6 curl : (6) Could not resolve host:
  example.com
7 :command
8
9 command: sudo curl -sS "http://
  example.com" 2>&1 | grep -Po "(?<=
  title>)(.+) (?=</title>)", line: 2,
  stack: test.bash
10 :command
11
12 command: echo "title is" $(sudo curl
  -sS "example.com" 2>&1 | grep -Po
  "(?<=title>)(.+) (?=</title>)",
  line: 2, stack: test.bash
13 title is
14 :command

```

The result is shown in Fig. 6.

The result shows that the tool detect changes in logs and the commands “sudo curl -sS “http://example”” are suggested as cause commands.

## 6 DISCUSSION

In the all Experiments, the difference in behavior between the script which executes commands with different behaviors is detected.

The cause command of the difference is identified in evaluation Experiment 1. In the experiment, differences in execution logs were detected in command substitution and assignment before they appear on standard output. Shell scripts that have complicated processing will almost certainly use such operators. Therefore, It is helpful for developers to find the difference before it comes out as standard output. This is also true for the pipeline which our method supports.



While, in the Experiment 2, the wrong command was suggested as the cause command. The reason is that the environment variables are not referred in the script called by the “sudo” command in CentOS8.2, and the difference in behavior is surfaced at the stage of “echo \$VAR” which performs standard output. In the case of change in the behavior of such a command which has no standard output, differences in behavior are detected in standard output or assignment. This is because the proposed method generates execution logs which focus on standard output and assignment. Therefore, it will be difficult to directly identify such commands as the cause command. Similarly, it will be difficult to precisely identify the cause commands in case of commands like “sudo,” which has a function to affect shared resources such as environment variables changed, i.e. a command with a strong side effect.

In Experiment 3, The tool identified cause command “sudo”. The command “sudo curl -sS "http://example.com"” accesses the URL via proxy server, which is described by “http\_proxy”. However, in CentOS8.2, the access to URL fails due to the command “sudo” doesn’t preserve environment variables. There are commands that refer to environment variables when performs, such as curl and tools to manage packages including yum and apt. These commands are generally used in building environment on server.

From result of the evaluations, Our proposed method can support developers in checking behavior changes in scripts.

## 7 FUTURE WORK

Defined grammars on lexical and syntax analyzers are simple. The analyzers cannot support complex grammars. Therefore, the tool needs to expand the grammars of analyzers.

Since this method only collects logs that appear in the standard output or assignment. If the behavior differs in areas that don’t appear in the standard output or assignment, e.g. signal trapping, it is not possible to verify whether the behavior changed.

The proposed method needs to evaluate the execution time and used memory for large and complex scripts such as recursive.

Thus, addressing these issues will be the main task in the future.

## 8 CONCLUSION

In this paper, we proposed a method to verify whether the behavior of shell scripts written in Bash is changed before and after OS upgrade and created a tool based on the method. The tool based on the proposed method can verify the behavior and identify the causes of the differences by comparing the execution logs of the shell scripts generated by log generating commands(Loggers). From evaluation experiments, we confirmed that our proposed method can verify differences in the behavior in standard output and assignment, and the method can support developers in verifying whether behaviors in scripts is changed before and after OS update efficiently. Furthermore, we conclude that it is difficult to precisely identify the cause commands

in case of commands which have no standard output or strong side effects.

## ACKNOWLEDGEMENT

Part of this work is supported by fund from Mitsubishi Electric Corp.

The research is also being partially conducted as Grant-in-Aid for Scientific Research A (19H01102) and C (21K11826).

## REFERENCES

- [1] “GNU Bash,” <https://www.gnu.org/software/bash/> (referred May 13, 2022).
- [2] J.Nam, S.Wang, Y.Xi, and L. Tan: “A bug finder refined by a large set of open-source projects,” *Information and Software Technology*, Vol.112, pp.164–175 (2019).
- [3] S.Kim, T.Zimmermann, K.Pan, and E.J.Whitehead Jr.: “Automatic Identification of Bug-Introducing Changes,” *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, pp.81-90 (2006).
- [4] S.Tsakiltidis, A.Miranskyy, and E.Mazzawi: “Towards Automated Performance Bug Identification in Python,” *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp.132-139 (2016).
- [5] K.Matsushita, M.Matsumoto, K.Ohno, T.Sasaki, T.Kondo, and H.Nakashima: “A Debugging Method Based on Comparison of Execution Trace,” *Symposium on Advanced Computing Systems and Infrastructures (SACSI)*, Vol.2011, pp.152-159 (2011) (in Japanese).
- [6] K.Hanayama, S.Matsumoto, and S.Kusumoto: “Humpback: Code Completion System for Dockerfiles Based on Language Models,” *In 1st Workshop on Natural Language Processing Advancements for Software Engineering(NLPaSE 2020)*, pp. 1-4 (2020).
- [7] H.Azuma, S.Matsumoto, Y.Kamei, and S.Kusumoto: “Survey of Self-Admitted Technical Debt in Container Virtualization Technology,” *IEICE technical report*, Vol.120, No.193, pp.25-30 (2020) (in Japanese)
- [8] E.Clarke, D.Kroening, and F.Lerda: “A tool for checking ANSI-c programs,” *International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2004, Lecture Notes in Computer Science*, Vol.2988, pp.168-176 (2004).
- [9] L.Cordeiro, D.Kroening, and P.Schrammel: “JBMC: Bounded Model Checking for Java Bytecode,” *International Conference on Tools and Algorithms for the Construction and Analysis of Systems(TACAS 2019)*, *Lecture Notes in Computer Science*, Vol.11429, pp.219-223 (2019).
- [10] A.Biere, A.Cimatti, E.Clarke, and Y.Zhu: “Symbolic Model Checking without BDDs,” *In Proceedings of the Workshop on Tools and Algorithms for the Construction*

- and Analysis of Systems(TACAS 1999), Lecture Notes in Computer Science, Vol.1579, pp.193-207 (1999).
- [11] K.Okano, R.Karashima, S.Harauchi, and S.Ogata: “Regression Verification for C Functions with Recursive Data Structure,” International Journal of Informatics Society, Vol.11, No.2, pp.107-115 (2019).
- [12] K.Okano, S.Harauchi, T.Sekizawa, S.Ogata, and S.Nakajima: “Consistency Checking between Java Equals and hashCode Methods Using Software Analysis Workbench,” IEICE Transactions on Information and Systems, Vol.E102, No.8, pp.1419-1422 (2019).
- [13] R.Karashima, S.Harauchi, S.Ogata, and K.Okano: “Proposal and evaluation for property verification for Java functions with recursive data structures by SAW,” Proceedings of International Workshop on Informatics 2019 (IWIN2019), pp.155-162 (2019).
- [14] “BASH Debugger,” <http://bashdb.sourceforge.net/>
- [15] “Oracle VM VirtualBox,” <https://www.virtualbox.org/> (referred May 13, 2022).
- [16] C.Oo and H.Min Oo: “Spectrum-Based Bug Localization of Real-World Java Bugs,” International Conference on Software Engineering Research, Management and Applications, pp.75-89 (2019)
- [17] T.Sato, T.Katayama, Y.Kita, H.Yamaba, K.Aburada, and Naonobu Okazaki: “Development of TFVIS (Transitions and Flow VISalization) for Java Programs,” Journal of Information Processing (JIP), Vol.59, No.4, pp.1137-1149 (2018) (in Japanese).
- [18] B.Kim, A.Ayoub, O.Sokolsky, I.Lee, P.Jones, Y.Zhang, and R.Jetley: “Safety-assured development of the GPCA infusion pump software,” 2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT), pp. 155-164, (2011).
- [19] G.Behrmann, A.David, and K.G.Larsen: “A Tutorial on Uppaal,” In Formal Methods for the Design of Real-Time Systems, Vol.3185, pp.200-237 (2004).
- [20] T.Koike: “Model Checking Support Environment based on State Transition Matrix,” SIGEMB, Vol.2008, No.116 pp.91-96 (2008) (in Japanese).
- [21] “PLY (Python Lex-Yacc) — ply 4.0 documentation,” <https://ply.readthedocs.io/en/latest/> (referred May 13, 2022).
- [22] F.DeRemer and T.Pennello: “Efficient Computation of LALR(1) Look-Ahead Sets,” ACM Transactions on Programming Languages and Systems, Vol.4, No.4, pp.615-649 (1982).

(Received: October 30, 2021)

(Accepted: February 23, 2022)



**Hitoshi Kiryu** is a graduate student of Shinshu University. His areas of interest include formal verification.



**Satoshi Suda** Satoshi received his M.E. degree in mathematical from Osaka University, Osaka, Japan, in 2016. He joined Mitsubishi Electric Corp. Currently he is a researcher of Solution Engineering Dept. at Advanced Technology R&D Center and mainly engaging in research on software development efficiency.



**Shinpei Ogata** is an Associate Professor at Shinshu University, Japan. He received his BE, ME, and PhD from Shibaura Institute of Technology in 2007, 2009, and 2012 respectively. From 2012 to 2020, he was an Assistant Professor, and since 2020, he has been an Associate Professor, in Shinshu University. He is a member of IEEE, ACM, IEICE, IPSJ, and JSSST. His current research interests include model-driven engineering for information system development.



**Kozo Okano** received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. From 2002 to 2015, he was an Associate Professor at the Graduate School of Information Science and Technology of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2020, he has been a Professor at the Department of Electrical and Computer Engineering, Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, and IPSJ.