Understanding Execution Environment of File-Manipulation Scripts by Extracting Pre-Conditions

Rodney Rodriguez and Xiaoyin Wang
Department of Computer Science
University of Texas at San Antonio
rodney.rodriguez@my.utsa.edu, xiaoyin.wang@utsa.edu

***Abstract*—File manipulation scripts are widely used in software projects to operate the file system at run time. Due to the emergence of DevOps practices in software industry, developers also use longer and more complicated file manipulations in their continuous integration and deployment scripts to automate software build, testing, and deployment in different environment configurations. A major challenge on understanding these scripts is that they make lots of implicit assumptions on the file system they are executed on. Such assumptions are rarely documented and often do not hold when a script is moved to another execution environment. In this paper, we propose a static-analysis-based technique that statically infer the directory tree pre-condition of the file system required to execute a file manipulation script. We evaluated our analysis on 58 docker files and the experiment shows that our technique is able to generate directory tree pre-conditions on real world scripts efficiently.**

I. Introduction

File systems are widely used for data storage in computer systems. To automatically manipulate files, various software projects use file manipulation scripts combining basic operations such as `touch`, `cp`, and `rm` provided by the operating system. Furthermore, the emerging DevOps practice in software industry requires the developers to fully automate the software building, testing, and deployment process, which leads to more complicated file manipulations in build scripts (e.g., gradle scripts, makefiles), deployment scripts (e.g., docker files), and continuous integration scripts (e.g., gitlab.yml, travis.yml).

A large portion of runtime errors in file manipulation scripts are related to path existence properties of the underlying file system. As a basis for estimation, we studied the open bug report repositories [1] of software projects. In Github which stores 391,231 closed bug reports in total for all Shell script projects, searching for the key phrase "file not found" and "file already exists" returns 33,466 results (performed on March 22^{nd}, 2019). Furthermore, path-existence-related failures are typically severe because they will directly cause termination of script execution which is often followed by a software crash.

Although file manipulation scripts are relatively short, they are still difficult to understand and run because developers often make assumptions on the existing paths in the file system over which they do not have full control. Once the system is modified by other scripts, and/or the script is executed on a different machine, the assumptions may no longer hold and the script will suffer from path-existence-related errors. To reduce path-existence-related errors, in this paper, we present a novel static-analysis-based technique to infer path-existence pre-conditions of file-manipulation scripts. In particular, given a piece of file manipulation script, our approach calculates the pre-condition for each control flow point in the script. The pre-condition at the beginning of the script can be then checked against a file system's file structure to determine whether the script can be executed on the file system without causing path-existence-related errors.

The first step of our technique is to design an abstract domain that summarizes the path-existence states of a file system. Despite the extensive research efforts [2] [3] on summarizing memory states including modeling both variable values [4] [5] and heap shapes [6] [7] [8], there have been few techniques developed to summarize file-system states, which brings two special challenges. First of all, unlike memory locations which are usually referred to by variable names and field names hard-coded in the source code, file system paths are often string values generated at run time by concatenating string variables. Second, although a file system state on path existence can be presented as a path tree, unlike the tree or graph domains used in heap shape analysis where edges can be labeled with predefined field names/indexes, the possible edge labels (folder and file names) in a directory tree are generated by the program at run time and thus are usually unbounded.

To overcome these two challenges, our key insight is to summarize the run-time state of a file system as a set of string values, each of which represents an existing path in the file system. Thus the state of the file system is presented as all the currently existing paths, and the abstract domain of the file-system state in static analysis can be defined as a string set that contains all possibly existing paths in the file system at a program point. To handle infinite paths / path sets due to loops and regular-expression-based path presentations (e.g., "rm foo/bar*"), we further use an automaton to represent the set of all possible paths. Such an automaton is referred to as a File-System-State (FSS) automaton, and the transfer functions of file operations such as `cp` and `rm` can be modeled as finite state transducers that transform one FSS automaton to another.

To sum up, this paper makes the following main contributions.

- An intermediate language FMIL that captures path-related semantics of file manipulation scripts.
- A static analysis FiFA that infers all possible directory tree states of running a given file manipulation script. Within FiFA, we use FSS automaton as abstract domains and finite state tranducers to define transfer functions.
- An evaluation on 58 dockerfiles which shows that FiFA is able to generate preconditions within reasonable amount of time.

The rest of this paper is organized as follows. Section II is going to present a motivating example to illustrate the requirement of FiFA and just the techniques we develop. We will introduce the details of FiFA in Section IV, and present our evaluation setup and results in Section V. After that, we discuss related works in Section VI. Finally, we conclude in Section VII.

II. Example

In this section, we present an example to illustrate the problem we are solving. Below is a real-world dockerfile from project Zalenium[1]. The original dockerfile has 434 lines so we cannot put the whole file here and can provide only two code snippets. The two snippets show two RUN commands (running its argument as a shell command) with if conditions. The first snippet checks whether the option `kubernetesSlimVersion` is set to true, and if it is not set, the dockerfile will download and set up the docker libraries. The second snippet setup the testing bots when they are enabled in the configuration. Both snippets create and remove some folders / files in the file system, and make assumptions of the file system. For example, the first snippet assumes that “docker/” exists, and the second snippet assumes that “tmp/” exists.

```
RUN if [ "${kubernetesSlimVersion}" = "false" ]; then \
     set -x \
     && DOCKER_VERSION="17.12.0-ce" \
     && curl -fSL "https://${DOCKER_BUCKET}/linux/static/..." \
          -o docker.tgz \
     && tar -xzvf docker.tgz \
     && mv docker/docker /usr/bin/docker-${DOCKER_VERSION} \
     && rm -rf docker/ && rm docker.tgz \
     && docker-${DOCKER_VERSION} --version | grep "${DOCKER_VERSION}
          "; \
   else \
     echo "Skipping adding Docker because of kubernetes slim 
          mode"; \
   fi
...
RUN if [ "${testingBotEnabled}" = "true" ]; then \
   cd /tmp \
   && wget -nv "${TB_TUNNEL_URL}" \
   && mv testingbot-tunnel.jar /usr/local/bin \
   && java -jar /usr/local/bin/testingbot-tunnel.jar --version; \
   else echo "Testing Bot Disabled"; \
   fi
```

There are many such configuration-guarded file manipulation pieces, making it difficult to test all of them and find out all the possible pre-conditions of the file system state before running the script.

[1]https://github.com/zalando/zalenium

III. Intermediate Language

File manipulation scripts can be written in many different programming languages, such as shell script, docker files, yml files, etc. These programming languages are also usually dynamic and flexible so that they allow code of other programming languages to be inserted as a part of its code. For example, it is standard practice to have embedded shell script snippets in Docker files and yml files. Furthermore, software configuration and deployment systems evolve very fast, so do the file-manipulation scripts they use. In the past decade, we witnessed the emerging of many new types of file manipulation scripts such as gradle scripts for gradle build tools, travis.yml for TravisCI continuous integration, and Docker files for Docker. Based on the above observations, we developed our technique on an intermediate language so that various current and future file-manipulation scripts can be benefit from our technique.

Since our technique uses its own intermediate language and we want to analyze Docker files in our evaluation, we need a way to convert the Docker files to data that our framework can understand. To do this we used a Dockerfile parser to look for Docker commands that could potentially modify the file system[2]. Commands such as COPY, ADD, and RUN can affect the Docker container’s file system. The COPY and ADD commands are fairly straight forward and could be translated to our framework’s copy command. The RUN command is the most challenging one to transform because it could be any shell command within the container. It is possible to make almost any modification to the file system using the RUN command. There were so many possibilities and due to time constraints we had to only consider the most common RUN commands. As per Docker’s Dockerfile documentation, the RUN command is run in a shell, which by default is ‘/bin/sh -c‘ on Linux[3]. In an effort to handle most RUN file cases, we implemented a shell parser to parse all the RUN commands within our Dockerfile data set[4]. We searched through all the RUN commands and counted each shell command occurrence. We decided to support the top 25 shell commands with the highest number of occurrences that may modify the file system.

IV. Approach

A. Abstraction Domain

In our technique, we use automaton as the abstract domain. The automaton represents all possibly existing file paths in the file system at a program point. For example, for the following piece of code, the automaton at the end of the program is shown in Figure 1

```
mkdir 'tmp'
if (other) {
    touch 'tmp\abc'
}else {
    touch 'def'
}
```

[2]https://github.com/asottile/dockerfile
[3]https://docs.docker.com/engine/reference/builder/#run
[4]https://github.com/mvdan/sh

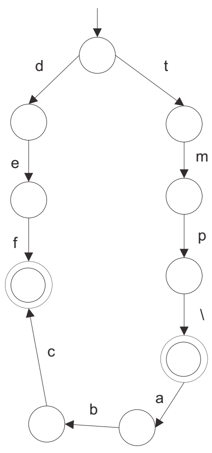

Fig. 1: An Exemplar FSS Automaton

In figure 1, we can see that, FiFA uses a post fix "/" to indicate that a path value is a folder. So FiFA can detect errors when a file is being referred to as a folder and vice versa.

B. Transfer Functions

To infer the pre-condition of a file-manipulation script, we need to run our analysis backward from the end of the script. The abstract domain is initialized as an empty automaton because no directory tree assumption is required at the end of a script. Whenever our analysis passed a file manipulation operation, it needs to apply the transfer function to the abstract domain. Note that we maintain two abstraction domains, one for positive pre-conditions (indicating that certain paths need to be existing, denoted as D), and the other for negative pre-conditions (indicating that certain paths must not be existing, denoted as N).

Since the basic operations such as union are well defined for automaton abstract domain, we just introduce our transfer functions for the basic file operations: `touch`, `mkdir`, `cp`, and `rm`. The `mv` operation can be presented as a `cp` operation and a `rm` operation, so we just translate it to two more basic operations. In particular, transfer functions for the five basic operations are presented below.

$$\begin{aligned} touch\ x : N &= N \cup automaton(x) \\ D &= D \cup parentdir(x) \end{aligned} \tag{1}$$

$$\begin{aligned} mkdir\ x : N &= N \cup automaton(x + \text{"/"}) \\ D &= D \cup parentdir(x) \end{aligned} \tag{2}$$

$$\begin{aligned} cp\ x\ y : N &= N \cup automaton(y + basename(x)) \\ D &= D \cup automaton(x) \end{aligned} \tag{3}$$

$$rm\ x : D = D \cup automaton(x) \tag{4}$$

In the functions, we use D and N to denote the positive and negative abstract domain before (on the right hand side) and after (on the left hand side) the operation. We use function $automaton(x)$ to denote the automaton generated by string analysis for path variable x. It should be noted that we do not have a transfer function for `cd` because it does not change the file system state, but only change the current directory. We handle `cd` by moving CD flags (indicating Current Directory) on the states of the FSS automaton. When `cd x` is encountered, we will calculate the intersection between FSS automaton D and $automaton(x)$, and put flags on the states in D which are paired with an acceptance state of $automaton(x)$ during the intersection process, and remove CD flags from all other states. For `rm` and `rmr`, we will do the automaton difference only if x is a constant string, to make sure our transfer functions are conservative. Otherwise since $automaton(x)$ is an over-estimation of x's possible values, the difference between D and $automaton(x)$ can be an under-estimation.

We use function $basename(x)$ to denote the automaton generated by extracting the base filename of a path variable x. Here we can see that because x is an automaton, the function $basename$ can only be implemented as a transducer, which is presented in Figure 2. In the finite state transducer, we use eps to denote ϵ, and texttu0001-uffff to denote the whole character set Σ, and $*$ as an output indicates that the output of the transducer at the specific transition will be the same as the input.

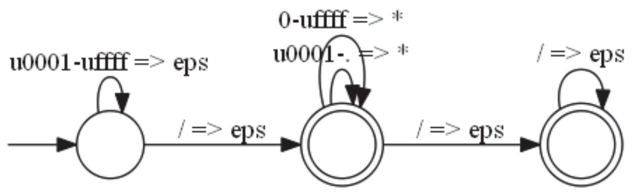

Fig. 2: FST to extract the basename

We use function $parentdir(x)$ to denote the automaton generated by extracting the parent directory of a path variable x. For example, the parent directory of a regular expression `a/b/c|a/d` would be `a/b/|a/`. The transducer to extract parent directory from a path variable x is presented in Figure 3.

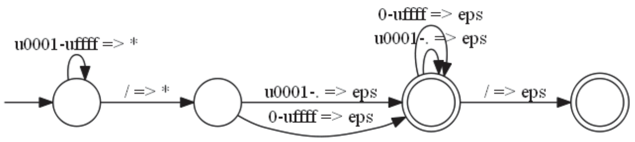

Fig. 3: FST to extract the parent folder

The definition of paths in file manipulation scripts can be various. For example, `mkdir abc` and `mkdir abc/` both create a folder with name `abc`. This may cause our automaton to have extra file path separators, resulting in double file path separators or extra file path separator at the end of a path value. To remove such extra path separators, we design the following two finite state transducers as shown in Figures 4 and 5.

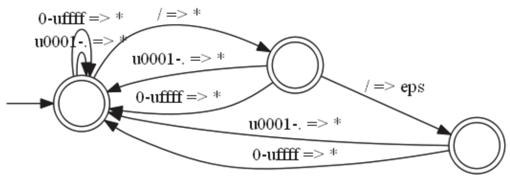

Fig. 4: FST to remove double file path separators

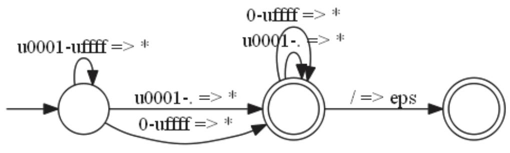

Fig. 5: FST to remove the last file path separator

V. Evaluation

A. Evaluation Setup

In our evaluation, we use a set of Docker files from a popular curated list of Docker resources and projects on Github[5]. We created a simple program to extract any projects that included any source code written using the Go programming language. We used a Dockerfile from each repository and automatically parsed them to generate an intermediate script that our framework could use to perform the analysis.

The results presented in this report were performed on a computer running Windows 10 with an Intel i7-6700K CPU and 16GB RAM. Our implementation and all evaluation subjects are available at our anonymous project website[6].

B. Evaluation Results

Our evaluation results are presented in Table I. The Columns 2-5 present the minimal, maximal, median, and average value of LOC (lines of code) of the file, execution time, and the total number of nodes in precondition automatons, respectively. From the results in the table we can see that the execution time of our technique is minimal, with a maximal execution time of 263ms, average execution time of 38ms, and median execution time of 21ms. Our created precondition averagely has 30 nodes and has 26 nodes as median. This shows that our generated pre-conditions are generally of readable size by users.

TABLE I: Analysis Results

Type	Min	Max	Median	Average
LOC	4	51	13	17
Execution Time (ms)	3	263	21	38
PreCondition	2	88	26	30

VI. Related Works

Horton and Parnin developed a technique [9] that aims to solve the dependency resolution problem by taking a runnable code snippet and installing all language-level and system-level dependencies so that we may execute the snippet without any import errors. Wolf et al. proposed an approach [10] to predict build errors from the social relationship among developers. McIntosh et al. [11] carried out an empirical study on the efforts developers spend on the building configurations of projects. Tamrawi et al. [12] proposed a symbolic-execution-based technique to analyze Make files and detect bad smells / common errors. Downs et al. [13] proposed an approach to remind developers in a development team about the building status of the project. Al-Kofahi et al. proposed an approach [14] to detect semantic changes in Make files, and later proposed an a fault localization approach [15] for Make files, which provides the suspiciousness scores of each statement in a Make file for a build error. Rehearsal [16] is a verification framework for configurations written in puppet. In particular, Rehearsal uses several static analyses to shrink the puppet abstraction models to a tractable size, and then frames determinism-checking as decidable formulas for an SMT solver. Tamrawi et al. [17] developed a symbolic execution framework, SYMAKE, for analysing make files and detect errors. SYMake first produces a symbolic dependency graph (SDG), which represents the dependencies among files during the building process. Adams et al. [18] presented a design and implementation of a reverse-engineering framework for build systems. Hassan et al. studied the feasibility of automatic software build [19], [20] and developed novel techniques to predict build results [21] and to repair build scripts [22] / docker files [23].

Our work is also related to string analysis [24]. Along this line of work, Wang et al. [25], [26], [27] developed techniques to trace origins of string values for software internationalization. Zhang et al., developed a string-analysis-based approach [28] to detect errors in database applications. Mostafa et al. [29] and Rodriguez et al. [30] developed NetDroid and NTApps to summarize network traffic based on string analysis.

VII. Conclusion

In this paper, we present a novel file flow analysis framework that infers pre-conditions of file-manipulation scripts. Our evaluation on more than 58 Docker files shows that our analysis can efficiently perform the analysis averagely within 38 milliseconds. In the future, we plan to extend our translator tool to handle shell environment variables, Dockerfile ENV commands, and conditional statements within RUN statements. We may also wish to add support for additional commands within the Docker RUN command.

Acknowledgments

This work is supported in part by NSF Awards NSF-1736209, NSF-1846467, and NSF-2007718.

[5]https://github.com/veggiemonk/awesome-docker
[6]https://sites.google.com/site/fifarepo/

References

[1] Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei, "Jdf: detecting duplicate bug reports in jazz," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2. IEEE, 2010, pp. 315–316.

[2] A. Miné *et al.*, "Tutorial on static inference of numeric invariants by abstract interpretation," *Foundations and Trends® in Programming Languages*, vol. 4, no. 3-4, pp. 120–372, 2017.

[3] P. Cousot, "Abstract interpretation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 324–328, 1996.

[4] D. Monniaux and L. Gonnord, "Cell morphing: From array programs to array-free horn clauses," in *International Static Analysis Symposium*. Springer, 2016, pp. 361–382.

[5] A. Cortesi and M. Zanioli, "Widening and narrowing operators for abstract interpretation," *Computer Languages, Systems & Structures*, vol. 37, no. 1, pp. 24–42, 2011.

[6] R. Wilhelm, M. Sagiv, and T. Reps, "Shape analysis," in *International Conference on Compiler Construction*. Springer, 2000, pp. 1–17.

[7] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 289–300.

[8] B. Guo, N. Vachharajani, and D. I. August, "Shape analysis with inductive recursion synthesis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 256–265.

[9] E. Horton and C. Parnin, "Dockerizeme: Automatic inference of environment dependencies for python code snippets," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, p. 328–338.

[10] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of ICSE*, 2009, pp. 1–11.

[11] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. Hassan, "An empirical study of build maintenance effort," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 141–150.

[12] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen, "Build code analysis with symbolic evaluation," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 650–660.

[13] J. Downs, B. Plimmer, and J. G. Hosking, "Ambient awareness of build status in collocated software teams," in *Proceedings of ICSE*, 2012, pp. 507–517.

[14] J. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. Nguyen, "Detecting semantic changes in makefile build code," in *Proceedings of ICSM*, 2012, pp. 150–159.

[15] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for build code errors in makefiles," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 600–601.

[16] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, p. 416–430.

[17] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, p. 650–660.

[18] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, Oct 2007, pp. 114–123.

[19] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 38–47.

[20] F. Hassan and X. Wang, "Mining readme files to support automatic building of java projects in software repositories," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 277–279.

[21] ——, "Change-aware build prediction model for stall avoidance in continuous integration," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 157–162.

[22] ——, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1078–1089.

[23] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 796–801.

[24] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18, available from `http://www.brics.dk/JSA/`.

[25] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun, "Locating need-to-translate constant strings for software internationalization," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 353–363.

[26] ——, "Locating need-to-translate constant strings in web applications," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 87–96.

[27] ——, "Transtrl: An automatic need-to-translate string locator for software internationalization," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 555–558.

[28] H. Zhang, H. B. K. Tan, L. Zhang, X. Lin, X. Wang, C. Zhang, and H. Mei, "Checking enforcement of integrity constraints in database applications based on code patterns," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2253–2264, 2011.

[29] S. Mostafa, R. Rodriguez, and X. Wang, "Netdroid: Summarizing network behavior of android apps for network code maintenance," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 165–175.

[30] R. Rodriguez, S. Mostafa, and X. Wang, "Ntapps: A network traffic analyzer of android applications," in *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, 2017, pp. 199–206.