RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis

Foyzul Hassan
University of Texas at San Antonio
USA
foyzul.hassan@utsa.edu

Rodney Rodriguez
University of Texas at San Antonio
USA
rodney.rodriguez@utsa.edu

Xiaoyin Wang
University of Texas at San Antonio
USA
xiaoyin.wang@utsa.edu

ABSTRACT

Dockerfiles are configuration files of docker images which package all dependencies of a software to enable convenient software deployment and porting. In other words, dockerfiles list all environment assumptions of a software application's build and / or execution, so they need to be frequently updated when the environment assumptions change during fast software evolution. In this paper, we propose RUDSEA, a novel approach to recommend updates of dockerfiles to developers based on analyzing changes on software environment assumptions and their impacts. Our evaluation on 1,199 real-world instruction updates shows that RUDSEA can recommend correct update locations for 78.5% of the updates, and correct code changes for 44.1% of the updates.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Dockerfiles, Software Environment, String Analysis

ACM Reference Format:
Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. 2018. RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240470

1 INTRODUCTION

Modern software often depends on a large variety of environment dependencies to be properly deployed and operated on production machines. Databases, application servers, system tools, and supporting files often need to be well installed and configured before software execution, and thus may cause tremendous effort and high risks during software deployment. This is not one-time but continuous cost due to the fast software evolution and delivery nowadays.

A practical approach to alleviate this effort is to use container images. A container image is a stand-alone and executable package of a piece of software with all its environment dependencies, including code, runtime, system tools, libraries, file structures, settings, etc. It can be easily ported and deployed to other machines, but is much lighter-weight than traditional virtual machines which can achieve similar goals.

Despite the large benefit brought by container images during software deployment, they also increase the effort of software developers because they need to generate and maintain the image configuration files which describe how the container images can be constructed with all environment dependencies, such as what tools and libraries should be installed and how the file structure should be set up. A recently study [7] on Dockerfiles by Cito et al. shows that in top projects a docker file is averagely revised 5.8 times each year (note that there can be multiple dockerfiles in one project, and the average and maximum number of dockerfiles per project in our dataset is 4.9 and 41). Such a task can be tedious and error prone because (1) modern software typically relies on many environment dependencies, and due to fast evolution of software requirements and underlying frameworks, such dependencies also need to be changed very frequently; (2) some environment changes (e.g., automatic system updates, environment changes during installation of irrelevant software) can happen without any developer actions so developers may even not be aware about them; (3) developers can easily neglect environment dependencies of their software when they set up or change them because the changes are made in the operating system instead of the software itself; and (4) many environment dependencies (e.g., system tools, supporting files) cannot be checked during software compilation but only used at runtime, so they can be easily missed during compilation and testing (which is hardly thorough). Once an incomplete or erroneous image configuration file is being used, the container image will also be incomplete or contains errors, which may cause failures in production machines.

In this paper, we propose a novel technique, RUDSEA, to help developer update container image configuration files more easily and with more confidence on their correctness. Specifically, based on an existing image container file, RUDSEA first tracks the accesses to the system environment from software source code and build configuration files. Such accesses are extracted as environment-related code scope. Then, for each code commit, RUDSEA traces its impact on environment-related code scope and automatically determines whether certain items in the image configuration file should be updated accordingly. Based on the type of code impact and configuration items, RUDSEA further recommends the actual updates that should be made on the items. We implement our technique for Docker[1], which is currently the dominating framework in container

[1]https://www.docker.com/

images for both software industry and open source community, and the image configuration files for docker are called *dockerfiles*. Notef that, although the implementation and evaluation of this research focus on docker images and dockerfiles, the general approach is applicable to other container images such as Kubernetes, OCI, etc.

To evaluate RUDSEA, we carried out an experiment on a dataset of 375 dockerfiles in 40 software projects collected from GitHub. Our evaluation shows that RUDSEA correctly recommends update locations for 941 of 1,199 instruction updates in dockerfiles, with a precision of 49.8%. Furthermore, RUDSEA is able to correctly recommend the actual revision for 529 of the 1,199 dockerfile updates. To sum up, this paper makes the following contributions.

- RUDSEA, a novel technique on automatically recommending update locations and contents for dockerfiles during software evolution.
- A dataset of dockerfiles and their corresponding historical versions as benchmarks for future research on this topic.
- An empirical evaluation of RUDSEA's effectiveness on real world dockerfiles.

The rest of this paper is organized as follows. First, we will introduce some background knowledge about dockerfiles in Section 2. Then, we describe our approach and detailed techniques in Section 3. After we present our evaluation results in Section 4, we discuss some related works in Section 5 and concludes in Section 6.

2 BACKGROUND

In this section, we will introduce some background knowledge about dockerfiles. A dockerfile typically consists of three parts. The first part (*From*) specifies an existing container image that the configured image is based upon. Some examples of existing images may include a clean Ubuntu system of a certain version, or a publicly available image prepared with Java, Android SDK and databases. The second part (*Parser Directives*) describes rules such as escape characters on parsing the rest of the dockerfile, and is optional. The third part (*Environment Replacements*) is the main part of the dockerfile, and describes how the image should be constructed with a sequence of instructions. The major types of instructions are listed below.

- *RUN & WORKDIR*: executing a system command or executable within the working directory specified by *WORKDIR*.
- *CMD & ENTRYPOINT*: setting the default command (*CMD*) to be executed and arguments(*ENTRYPOINT*) to be use when executing the container image.
- *LABEL*: Setting environment variables in the container image.
- *EXPOSE*: exposing a network port in the container image.
- *ENV*: defining a variable to be used in the rest of the dockerfile.
- *ADD / COPY*: add a new directory / file in the file system of the container image, and copy directories / files from hosting system to the image.

From the list, we can see that three types of instructions will be updated frequently during software evolution, which are *RUN* instructions (updating versions of tools / libraries to be installed), *Label* instructions (updating environment variables), and *Add / COPY* instructions (changing default file structures). By contrast, other instructions are either typically stable (e.g., *EXPOSE*, *CMD & ENTRYPOINT*) or used only in the dockerfile itself (e.g., *ENV*). Therefore, our paper focuses on the updates of *RUN*, *LABEL*, and *ADD / COPY* instructions.

3 APPROACH

As shown in Figure 1, our approach consists of two major components. The first component extracts software code that is related to the items in dockerfiles. Here the software code base includes source files, build configuration files, and property files. The core part of this component is value dependency analysis, and we apply it to both old and new versions to acquire the results for both versions. The second component receives the analysis results of two code versions and generates the actual updates. It leverages change impact analysis to determine whether the code change may affect the environment-related code, and equivalence analysis to check whether new code is added as the equivalent part of known environment-related code.

3.1 Extracting Environment-related Code Scope

The major challenge of extracting environment-related code is the complicated interface between software and its environment. While software libraries and their versions are typically listed in build configuration files (e.g., `makefile` for GNU Make, `pom.xml` for Maven, `build.gradle` for Gradle), references to file paths and environment variables are often scattered in source code, build configuration files, property files, etc. A thorough definition of all possible environment interfaces requires huge manual effort, and the definition can easily be out-of-date due to quick evolution of the underlying development frameworks, build configuration tools, and their various plug-ins.

To overcome this challenge, RUDSEA uses a different solution. Our intuition is that, all the environment related code, no matter how they interface with environment, must refer to the values in the items of dockerfiles. Note that here we assume that the original version of the dockerfile is a correct one. Simply put, we can search for the values from dockerfile items in the constant string values in various source files, since such values must be used when software interfaces with the environment.

However, a simple keyword search does not work, because developers frequently use string concatenations and value assignments to generate runtime values from the string constants. For example, the dockerfile may refer to a file path `/home/project-name/foo/bar`, while in the source code, the file path may be a string concatenation expression such as `"/home/" + project + "/" + module + "/bar/"`, where `project` and `module` are variables for flexibility of changing sub-projects and modules. In such cases, the original values will not be detectable with simple keyword search, but string concatenations and assignments need to be considered. In our initial implementation of RUDSEA, we consider only string concatenations, as we find that other string operations are rarely used in generating library names, file paths, and environment variable values.

Therefore, RUDSEA uses a two-stage approach, which first locates the initial string constants which are long enough substrings of a dockerfile item. Then, RUDSEA performs value dependency

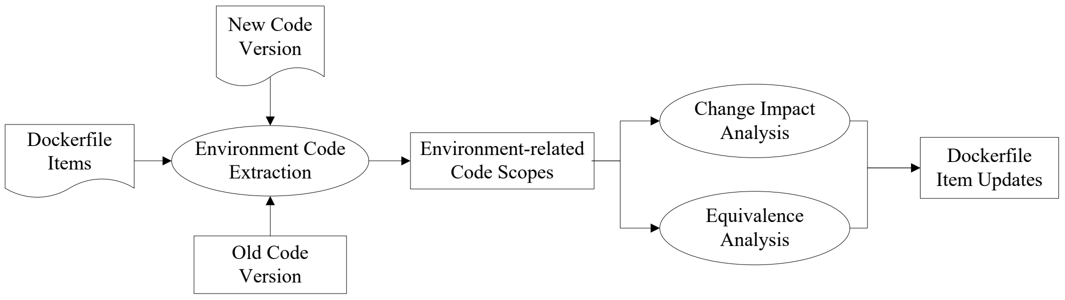

Figure 1: RUDSEA Overview

analysis to compute additional values through manipulating these initial string constants. As our analysis is light weight, we need only a parser and known string concatenation functions (which are typically only several, and very similar among all programming languages) for each programming language used in the software project.

3.1.1 Locating Initial String Constants. The first step of locating initial string constants is to extract dockerfile item values from dockerfiles. To achieve this, we use a dockerfile parser to extract all argument values of *RUN*, *Label*, and *Add / COPY* instructions. Since *RUN* instructions often take Linux utility commands (e.g., *mkdir*, *apk-get install*) as their parameters, and such commands are not necessarily referred in the software code base, we filter out all such commands from dockerfile item values.

After collecting the list of dockerfile item values, RUDSEA extracts all string constants from the software code base, and verifies whether their length is over 3 and is a substring of any dockerfile item values. If so, the string constant is added to the set of initial string constants. In particular, given a string constant str, and a set of dockerfile item values D, Formula 1 presents a boolean function *env* which checks whether str is an initial string constant. In the rest of the paper, the set of initial string constant is denoted as $Init$.

$$env(str) = len(str) \geq 3 \wedge \exists d \in D, d.contains(str) \tag{1}$$

In the formula, we use $len(x)$ to represent the length of string x, and $x.contains(y)$ to represent string y is a substring of x. We will use this check function also in our value dependency analysis to make the abstract domain bounded. Based on the initial string constants, RUDSEA performs value dependency analysis which checks how string constants are combined with each other to form more values, and tracks the string manipulation process.

3.1.2 Value Dependency Analysis. The value dependency analysis in RUDSEA is a static analysis on string concatenations and assignments within the software code base. Value dependency analysis uses an abstract domain $< \Gamma, T >$. Γ is a set of mappings from the set of string variables V in the software code base, to sets of string values generated from the set of string constants (S) in the code base. Specifically, Γ is defined in formula 2.

$$\Gamma = \{var \rightarrow L | var \in V \wedge L \subset S^*\} \tag{2}$$

For each value in L, we also track the locations of string constants that form each value in T, so basically T is a mapping from a string value in L to a set of program points.

Why RUDSEA does not use automatons to represent string values? In our value dependency analysis, to track string concatenations and assignments, we use a string set domain instead of an automaton as in string taint analysis [22] for two reasons as follows. First, string taint analysis (and also the original string analysis [5]) uses the Mohri-Nederhof algorithm to handle strongly connected components in string dependencies, and generates an approximate automaton, which is a slow process and typically results in over-approximation and affect analysis accuracy. Second, in string taint analysis, the tracing from original string constants to the final string values is at character level, which makes it difficult to propagate updates from original string constants to the final string values.

Despite the accuracy, efficiency, and straightforward tracing provided by the string value set domain, its major drawback (and why it cannot be used in general string analysis) is that it is not bounded. When a string variable is written within an unbounded loop or recursive method, the possible values of the variable can be infinite.

In the specific application scenario of RUDSEA, we find that this problem can be solved. Our idea is to use the *env* function to bound the string value set in our domain. The intuition is that, if a possible value of a string variable does not satisfy *env* function, it will not be a reference to dockerfile item values, and thus can be discarded. Therefore, given that dockerfile item values are finite, all string values in our abstract domain will be perfectly bounded (without any accuracy loss regarding reference to dockerfiles) by the dockerfile item values through *env* function. In particular, the transfer functions of value dependency analysis on string initialization, string assignments, and string concatenations are defined in

Once value dependency analysis converges at a fixed point, we can tell for each variable, what are its possible values (satisfying *env* functions) and the original string constants and string concatenations used in forming each value. If a string variable var contains a value val that is identical with any dockerfile item value, we will consider var and all the statements used in forming val as in the environment-related code scope. Specifically, we denote all the dockerfile item values generated from software code base with value dependency analysis as Gen, and Gen is formally defined in Formula 3. Recall that D is the set of all dockerfile item values extracted from the dockerfiles.

$$Gen = \bigcup_{var \in V} \Gamma(var) \cap D \tag{3}$$

Then the environment-related code scope can be formally defined as in Formula 4. Recall that T is a part of our abstraction domain which maps any string value in Γ to program points involved in generating the value. *Gen* and T will be further used in our Dockerfile change generation stage.

$$Scope = \bigcup_{val \in Gen} T(val) \tag{4}$$

3.2 Dockerfile Change Generation

Given a new software version, RUDSEA's dockerfile change analysis tries to find out what updates on the code will affect items in dockerfiles. Note that RUDSEA does not take single code commit as its input, because dockerfiles are often not updated until a new release so there may be many code commits in between. Environment change analysis include the change impact analysis which examines whether known environment-related code scope will be affected by the changes, and the equivalence analysis which examines whether a new environment-related code scope is added.

3.2.1 Change Impact Analysis. In the change impact analysis, RUDSEA will perform value dependency analysis on the new version of the software, and map the analysis results (string constants and statements involving string concatenations / assignments) with that from the original version with a file difference tool. In the rest of this section, we denote *Gen*, T, and *Scope* generated from the value dependency analysis on the original version as Gen_{old}, T_{old}, and $Scope_{old}$, while the corresponding results on the new version as Gen_{new}, T_{new}, and $Scope_{new}$. We further define the set of variables that has at least one possible value in *Gen* as *Gvar*. We refer to such variables as *docker variables*. Similarly, we have $Gvar_{old}$ and $Gvar_{new}$. Note that *Gvar* is formally defined in Formula 5.

$$Gvar = \{var | var \in V \wedge \Gamma(var) \cap Gen \neq \emptyset\} \tag{5}$$

The intuitive assumption behind our change impact analysis is as follows. If a variable *var* has a dockerfile item value in its possible value set $\Gamma(var)$ (i.e., *var* is a docker variable), it is likely to be used for environment interfacing. Therefore, if it holds a different set of values in the new version, the new set of values are likely to be also used for environment interfacing and should be added to the dockerfile. Furthermore, if a docker variable is deleted in the new version, the corresponding dockerfile item value may also need to be deleted if no other docker variables hold the same value in the new version.

As an example, consider a variable *var* having a possible value `"/home/foo/bar"` in the old version, and the value is a dockerfile item value. In the new version, if the same variable has a possible value `"/home/foo/bar2"`, then it is likely that we should add `"/home/foo/bar2"` to dockerfiles. In particular, if `"/home/foo/bar"` is no longer in $\Gamma_{new}(var)$, we should replace `"/home/foo/bar"` with `"/home/foo/bar2"`. If `"/home/foo/bar"` is still in $\Gamma_{new}(var)$, we should insert a new instruction that performs exact the same operation on `"/home/foo/bar2"` as on `"/home/foo/bar"`. If the variable *var* is deleted in the new version, and no other docker variables has `"/home/foo/bar"` in its possible values, the value should be deleted from the dockerfile.

A complication in this process is when a old docker variable (*var* in $Gvar_{old}$) holds multiple values in Gen_{old}, or hold other values that are not in Gen_{old}. In such cases, when the possible values of *var* contains some new value in the new version, it is hard to tell which old value this new value is replacing or complementing. Our solution is to compare their forming process stored in T. Given a new value *newv* in $\Gamma_{new}(var)$, we compare $T_{new}(newv)$ with each of the old values *oldv* in $\Gamma_{old}(var)$, and map this new value to an old value *oldv* whose forming process $T_{old}(oldv)$ is most similar to $T_{new}(newv)$. Specifically, we measure similarity by the size common program points between $T_{old}(oldv)$ and $T_{new}(newv)$.

3.2.2 Equivalence Analysis. While change impact analysis is able to recommend dockerfile updates related to existing dockerfile item values. There are also other cases where a new environment dependency is added. RUDSEA needs to also detect those cases and find out where the insertions need to be made.

To solve this issue, we develop equivalence analysis which checks which two program points have similar usage in the program. They are considered equivalent program points. In our analysis, we consider similar code inside one basic block or in different alternative blocks (i.e., basic blocks within the same level in a conditional statement). Examples of alternative blocks are `if` and `else` blocks within one conditional statement, or `case` blocks within one switch statement.

The intuition behind equivalence analysis is that if a writing statement to a string variable *equiv* is inserted as a equivalent program point of a writing statement s which writes to a docker variable *var* with dockerfile item value *val*, the inserted writing statement will be considered as a new docker variable, and its possible values will be recommended for insertion into dockerfiles. For each possible value of *equiv*, RUDSEA recommends an insertion of a new instruction that performs exact the same operation on *equiv* as on *val*.

3.3 Implementation

We implemented the value dependency analysis of RUDSEA for Java, PHP, and Gradle. To support Maven, simple property files, and XML property files, we further convert all dependencies and property definition in such files as string constant assignments (i.e., assignment of property value to property name, and dependency values to a special variable "dependency"), thus they can be handled by the dockerfile-update generation component of RUDSEA.

4 EVALUATION

To evaluate the effectiveness of RUDSEA, we carried out an experiment on a set of software projects with dockerfiles, and used their version histories as ground truth to check how accurate RUDSEA's recommendation is. Specifically, we try to answer the following two research questions.

- **RQ1:** How effective is RUDSEA on recommending update locations in Dockerfiles?
- **RQ2:** How effective is RUDSEA on recommending updates in Dockerfiles?
- **RQ3:** What are the major reasons causing RUDSEA to fail on recommending correct updates?

In the rest of this section, we introduce the dataset construction, evaluation metrics, evaluation results, and threats to validity in the following four subsections, respectively.

4.1 Dataset of Dockerfiles

We collected a set of Docker-using open source projects in Github[2]. In particular, we searched through top Java and PHP projects by number of stars and check whether the project contains dockerfiles. If so, we added the project into our dataset. We stopped after we collected 20 PHP projects and 20 Java projects. Then, we checked the history of the dockerfiles in these projects. In some projects, dockerfiles have their own repository, so we gathered the dockerfiles from there. In some other projects, dockerfiles are attached with each release (so they do not have a version history), we collected all dockerfiles from all releases so that they form a version history. From the version history of dockerfiles, we used diff to generate ground truth updates of dockerfiles. We further removed all internal updates of dockerfiles (e.g., updates of comments, refactorings). Finally, we acquired a dataset of 375 external updates of dockerfiles, each of which can be ascribed to one or more updates in the source code and / or build configuration files. In our evaluation, we use the updates in the source code and / or build configuration files as input, and the corresponding dockerfile updates as output. It should be noted that each update may involve multiple instruction updates. In total, the dockerfile updates include 1,199 instruction insertions, revisions, and deletions.

One question should be studied is how large the dockerfiles are, so that we can see how difficult the update localization is. To answer this question, we further performed an empirical study on our dataset. In the 40 Java and PHP projects, there are 197 dockerfiles in total. The number of dockerfiles in a single project ranges from 1 to 41, and the average number is 4.9. The number of valid lines (excluding blank and comment lines) in dockerfiles varies from 1 to 64 lines, and the average is 28 lines. Since there are often multiple docker files in one project, the average number of dockerfile lines in a project is 137 lines, and the number of lines ranges from 12 lines to 622 lines. Although dockerfiles are relatively smaller than source code, they are condense formatted (i.e., there are often multiple commands to be executed in one line), and their dependency on the code is latent. So the localization of updates is still a difficult problem.

4.2 Metrics

In our experiment, we use the traditional metrics of precision, recall, and F-score to measure the effectiveness of techniques. We consider a recommended location to be correct, if the recommended instruction to be updated is revised, deleted, or have another instruction inserted before of after it in the real version history.

For a recommended update to be correct, we require the recommendation has the same type (insertion, update, or deletion), same instruction type, and argument value. Here we consider equivalent updates as also correct. For example, recommending a same insertion at a different location from the real insertion is also considered correct as long as the location difference does not cause difference in semantics.

[2]The dataset is available at https://sites.google.com/site/rudseaproject/

Table 1: Results on recommending update locations

Project	# of Inst. Updates	P (%)	R (%)	F (%)
PHP	720	53.9	79.7	64.3
Java	479	44.5	76.6	56.3
All	1,199	49.8	78.5	60.9

Table 2: Results on recommending updates

Project	# of Inst. Updates	P (%)	R (%)	F (%)
PHP	720	28.7	42.6	34.3
Java	479	27.0	46.3	34.1
All	1,199	28.0	44.1	34.3

4.3 Evaluation Results

To answer **RQ1**, we present our evaluation results in Table 1. In the table, we present the type of projects, the number of actual instruction updates, precision, recall, and F-score in Columns 1-5, respectively. From the table we can see that RUDSEA is able to achieve high recall (averagely 78.5%) and acceptable precision (averagely 49.8%) in recommending update locations. Note that, since averagely less than four updates are performed in each commit, achieving a precision at around 50% means that developers need to inspect averagely eight locations, and finding four of them correct.

To answer **RQ2**, we present the results in Table 2 with the same format. From the table we can see that RUDSEA can achieve an average recall of 44.1% on recommending direct updates. This means that RUDSEA can recommend exactly correct updates for 529 of 1,199 instruction updates, which may save a large amount of effort of developers. Compared with the recall on location recommendation, we can see that for the updates RUDSEA successfully recommends locations, about 56% (529) are exactly correct updates. To answer **RQ3**, we studied the remaining 412 incorrect updates and find the errors mainly fall into three categories.

First, RUDSEA may insert an instruction at a wrong location. For simplicity, when RUDSEA finds that a docker variable has a new value which can be mapped to a dockerfile item value v in change impact analysis or equivalence analysis, RUDSEA always insert an extra instruction after the instruction handling v. Since instructions in dockerfiles are executed in sequence, such an insertion location may be wrong, especially when v is handled in a long instruction concatenated with "&&". This category accounts for 207 incorrect updates and we believe that most of them can be resolved by more fine-grained rules on dockerfile insertions.

Second, although RUDSEA correctly recommends an insertion, the inserted argument may not be correct. Developers sometimes add extra parameters to the *RUN* instructions they added, but RUDSEA is not able to recommend such parameters as it does not understand their semantics. This category accounts for 90 incorrect updates.

Third, when a docker variable cannot be mapped to a variable in the new version, RUDSEA simply deletes dockerfile item values in its possible value set from dockerfile. Some complicated version updates of the software cause difficulties in finding correct mapping of variables between versions and thus RUDSEA may delete a value that should be revised. This category accounts for 65 incorrect updates and we believe that they can be partly resolved by using more precise version diff tools.

4.4 Threats to Validity

The major threat to the internal validity of our evaluation is whether the ground truth updates we used in our experiment are all correct. Although we use real-world updates, developers may make erroneous updates or miss some updates, which may cause inaccuracy in our results. Also, the implementation of RUDSEA may be not perfect and involve bugs. The major threat to the external validity is that our evaluation results apply to only the subject projects and updates, or only Java / PHP projects. To reduce this threat, we use projects from Github based on different programming languages.

5 RELATED WORK

Studies and Analyses of Dockerfiles. With the increase of software complexity and components, managing of software dependencies [12] and test dependencies [13] has become an important problem. Tufano et al. [19] studied on broken snapshots and likely causes behind broken snapshots. Recent research work on scientific artifact reproduction [4] discussed about the uses of Docker to address the challenge of operating system virtualization, cross-platform portability, and reusable software components. Cito et al. [6] discussed about the rise of Docker adoption in industry, and performed an empirical study on dockerfiles [7]. Rahman and Williams [15] performed an empirical study on the type of defects in dockerfiles. Docker is also used for lightweight virtualization for developers for distributed application development, build and ship [11].

Analysis of Building Configuration Files. As build configuration files are getting complex and diverse, research on build configuration file is getting importance that includes dependency analysis, migration of build systems and empirical studies. To keep consistency during revision, Adams et al. [1] proposed a framework to generate dependency graph of build configuration files. Al-Kofahi et al. [2] proposed a fault localization technique for make files, and SYMake [18] uses a symbolic-evaluation-based technique to detect common errors in Makefile. Following works by Zhou et al. [24] and Al-Kofahi et al. [3] try to find configuration values exercising different parts of makefiles. Shambaugh [16] developed a verifier for puppet configuration script, and Sharma et al. [17] proposed techniques to detect bad smells in configuration files. Recently, Hassan et al. studied the reproduction of building environments [8, 9], and performed AST level analysis to generate fix patch for build configuration files [10] .

String analysis. String analysis [5] is a static analysis technique to estimate possible values of string variables. String analysis has been applied to detecting vulnerabilities [22, 23], repair web interfaces [21], software internationalization [20], inter-component communication analysis [14], etc.

6 CONCLUSION AND FUTURE WORK

In this paper, we present RUDSEA, which is a novel approach to recommend updates for dockerfiles during software evolution. RUDSEA leverages tracks environment accesses from code to extract environment-related scopes from the old software version and the new software version. Then, RUDSEA generates updates from the two versions of analysis results. Our evaluation on 40 projects and 1,199 real-world instruction updates shows that RUDSEA can recommend correct update locations for 78.5% of the updates, and correct updates for 44.1% of the updates, with moderate false positives.

ACKNOWLEDGMENT.

The authors are supported in part by NSF Awards CCF-1464425, CNS-1748109, and DHS grant DHS-14-ST-062-001.

REFERENCES

[1] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter. 2007. Design recovery and maintenance of build systems. In *ICSM*. 114–123.

[2] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2014. Fault localization for Make-Based build crashes. In *ICSME*. IEEE, 526–530.

[3] Jafar Al-Kofahi, Tien N Nguyen, and Christian Kästner. 2016. Escaping AutoHell: a vision for automated analysis and migration of autotools build systems. In *RELENG*. 12–15.

[4] Carl Boettiger. 2015. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 71–79.

[5] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. 2003. Precise analysis of string expressions. In *SAS*. Springer, 1–18.

[6] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. 2015. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *FSE*. 393–403.

[7] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the Docker container ecosystem on GitHub. In *MSR*. IEEE, 323–333.

[8] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *ESEM*. 38–47.

[9] Foyzul Hassan and Xiaoyin Wang. 2017. Mining readme files to support automatic building of java projects in software repositories: Poster. In *ICSE, Poster*. 277–279.

[10] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *ICSE*. 1078–1089.

[11] Muhamad Fitra Kacamarga, Bens Pardamean, and Hari Wijaya. 2015. Lightweight Virtualization in Cloud Computing for Research. In *Intelligence in the Era of Big Data*, Rolly Intan, Chi-Hung Chi, Henry N. Palit, and Leo W. Santoso (Eds.). 439–445.

[12] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In *ISSTA*. 215–225.

[13] Shaikh Mostafa and Xiaoyin Wang. 2014. An empirical study on the usage of mocking frameworks in software testing. In *QSIC*.

[14] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-component Communication Analysis. In *ICSE*. 77–88.

[15] Akond Rahman and Laurie Williams. 2018. Characterizing defective configuration scripts used for continuous deployment. In *ICST*. 34–45.

[16] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: a configuration verification tool for puppet. In *International Conference on Programming Language Design and Implementation*. 416–430.

[17] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *MSR*. IEEE, 189–200.

[18] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. SYMake: A Build Code Analysis and Refactoring Tool for Makefiles. In *ASE*. 366–369.

[19] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Soft.: Evo. and Proc.*, 29, 4 (2017).

[20] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Transtrl: An automatic need-to-translate string locator for software internationalization. In *ICSE, Tool Demo*. 555–558.

[21] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *FSE*. 16.

[22] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*. 32–41.

[23] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based Symbolic String Analysis for Vulnerability Detection. *Form. Methods Syst. Des.* 44, 1 (Feb. 2014), 44–70.

[24] Shurui Zhou, Jafar Al-Kofahi, Tien N Nguyen, Christian Kästner, and Sarah Nadi. 2015. Extracting configuration knowledge from build files with symbolic analysis. In *RELENG*. 20–23.