2021 IEEE 15th International Workshop on Software Clones (IWSC)

Type-2 Code Clone Detection for Dockerfiles

Tomoaki Tsuru, Tasuku Nakagawa, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
E-mail: {t-tsuru, t-nakagw, shinsuke, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—This paper proposes a Type-2 code clone detection technique for Dockerfiles. Docker is a platform for realizing containerized virtual environments that has been attracting significant attention as a technology support for server infrastructures. In Docker, the procedure to realize a virtual environment is described in the form of a source code called a Dockerfile. Therefore, code clones such as repetitions and duplications with similar structures are often included. In this study, we propose a Type-2 code clone detection technique for deriving patterns within Dockerfiles. More specifically, our technique detects code clones by normalizing tokens properly, separating Docker syntax and shell syntax, and then adopting suffix array algorithm for each syntax. We conducted application experiments on approximately 5,000 Dockerfiles available on GitHub and found that our technique could detect Type-2 code clones with a high precision. Our technique derived some patterns in Dockerfiles, such as those used for building a new distribution on a container.**

I. Introduction

Docker has been the gold standard for providing container-level virtualization. Docker is one of the key technologies that realizes infrastructure as code (IaC) where the procedure of container (i.e., virtual environment) construction is explicitly described as source code called Dockerfile. IaC enables rapid reproduction of exactly the same infrastructure environment no matter who executes it or when it is executed [1]. However, it has been pointed out that there are still some immature domains of research because Docker and IaC-relating technologies are in their infancy [2].

This paper focuses on code clones (clones) included in Dockerfiles. A clone is a code fragment that is identical or similar to another fragment. In the field of software engineering, numerous studies has been conducted on code clone detection for various programming languages [3] [4] [5]. Clone detection can be applied for various applications; recommendation for usage of application programming interfaces (APIs) [6], refactoring recommendations for redundant code fragments [7], suggestions to change fragments requiring the same change [8], and license violation detection [9].

Clone detection techniques have been proposed for not only procedural languages such as Java and C but also various other languages. On the other hand, there have been a few proposals for Dockerfile clone detection In one such study, Oumaziz *et al.* have proposed a duplicate code detection technique for Dockerfiles [10]. This technique isolates tokens in Dockerfiles and detects duplicates using inverted index algorithm. However, this technique cannot detect clones except for variations in variables (Type-2 clones) because it only targets duplicate code fragments that match perfectly (Type-1).

In this study, we propose a Type-2 code clone detection technique for Dockerfiles. The contributions of this study are as follows:

- Consideration and definition of Type-2 clones in Dockerfiles: Type-2 clones allow for differences in tokens that do not affect the behavior of the program. Therefore, it is essential that variable names, function names, and constants be normalized properly to facilitate for their detection. Dockerfile contains not only variables but also various constant tokens such as file paths and usernames. In addition, there are many notations that do not affect the behavior of Dockerfiles, such as parameter order and option aliases. We consider Type-2 clones in Dockerfiles by organizing the tokens to be normalized.
- Proposal of a clone detection technique in Dockerfiles that considers multiple syntaxes: Dockerfile is a nested language that allows the syntax of multiple languages to be written in a single file [11] [12]. More specifically, it uses shell syntax to describe the internal processing of the container and Docker syntax to describe processing with the outside of the container such as copying files from the host operating system (OS) or specifying the container network ports. Our proposed clone detection technique first generates abstract syntax trees (ASTs) from the given Dockerfiles and then normalizes the tokens. Our technique also separates Dockerfiles into two syntaxes and then uses suffix array algorithm to achieve multi-syntactic clone detection.
- Type-2 clone detection for publicly accessible Dockerfiles and pattern derivation in Dockerfile: Type-2 clone detection using our proposed technique is performed on 4,817 Dockerfiles in 725 Dockerfile repositories available on GitHub. The results of these experiments showed that Type-2 clones are detected with a precision of 95 %. We also visually derive patterns that appear in multiple Dockerfiles, such as those used for building a new distribution on a container and software installations using `make` commands.

Work licensed under Creative Commons Attribution 4.0 License. https://creativecommons.org/licenses/by/4.0/
DOI 10.1109/IWSC53727.2021.00007

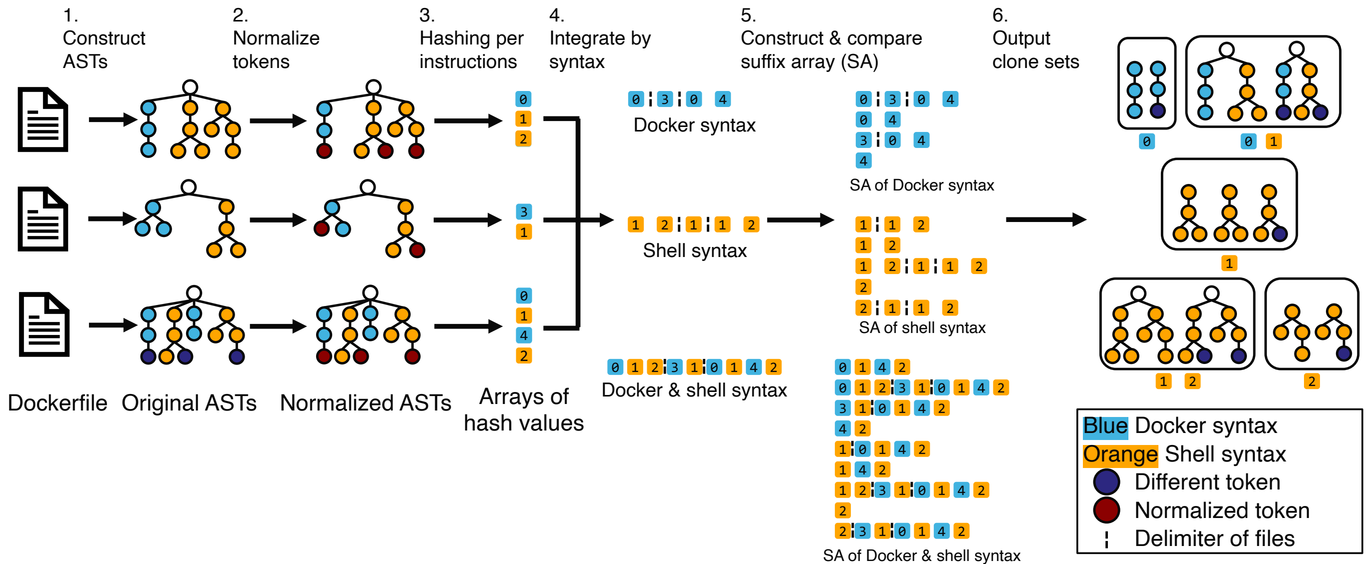

Fig. 1: Overview of Type-2 clone detection for Dockerfiles

II. Preliminaries

A. Code Clone

A code clone (clone) is a code fragment that is identical or similar to another fragment. A set of code fragment that is identical or similar to another fragment is called a clone set. Clones are classified as follows based on the degree of similarity to their correspondences [3].

- Type-1: an identical copy except for variations in white spaces and comments.
- Type-2: a syntactically identical copy except for variations in identifiers such as variables and functions.
- Type-3: a copy with further statement-level modifications such as added or removed statements.

So far, numerous clone detection techniques and tools have been proposed for general programming languages [3] [4] [5]. General applications of clone detection include recommendation for usage of application programming interfaces (APIs) [6], refactoring recommendations for redundant code fragments [7], suggestions for changes to fragments requiring the same change [8], and license violation detection [9]. Clone detection research has also been tackled in various languages other than programming languages, such as build tools [13] and requirement specifications [14].

B. Docker

Docker is a platform that implements operation-system-level virtualization called container. Docker has attracted significant amounts of attention in recent years; 87 % of information technology companies and a variety of open-source software use Docker [15] [16].

Docker is one of the key technologies to realize infrastructure configuration automation called infrastructure as code (IaC) by describing the procedure for building containers as source code. Dockerfile is the source code describing the procedure for building containers. Dockerfile composes an image which is the template of the container. Dockerfiles is a nested language within which shell syntax in the `RUN` instruction can be embedded [11] [12].

Docker and other IaC-related technologies are in their infancy, and there are various IaC-related studies such as characterizing Dockerfiles smells [17] and code completion for Dockerfiles [18]. On the other hand, it has been pointed out that there are some IaC domains that are still under-researched [2].

C. Duplicates in Dockerfiles

There are a few studies on code clone detection in Dockerfiles. Oumaziz *et al.* propose a duplicate code detection technique for Dockerfiles [10]. They use inverted index algorithm [19] with some Dockerfile instructions as a chunk to detect duplicate code after parsing the Dockerfiles. The size of chunks goes from one Dockerfile instruction to the maximal number of instructions contained in the Dockerfile. In Dockerfile, there is the practice of writing multiple shell commands inside a single `RUN` instruction. In their technique, `RUN` instructions are separated into internal shell commands during parsing in order to detect duplicate codes inherent in shell commands.

Oumaziz *et al.* only detect exact duplicates (Type-1 clones) because they do not apply token normalization. This means their technique cannot detect Type-2 clones, which are caused by variations in the tokens such as variables. In addition, even if similar Docker instruction or shell command sequences appear in multiple Dockerfile projects, their technique cannot be applied to pattern derivation in Dockerfiles due to the presence variations in token strings.

III. Proposed Technique

We propose a Type-2 clone detection technique for Dockerfiles. This technique can derive patterns that appear in multiple Dockerfiles. Figure 1 shows an overview of Type-2 clone detection for Dockerfiles. The input of our technique is a set of Dockerfiles and the output is the detected clone sets. The flow of our technique is described in Figure 1. This technique first constructs ASTs corresponding to each of the multiple Dockerfiles given as input (step 1) and then normalizes each token (step 2). This normalization process allows us to extend the Type-1 clone detection to Type-2 clone detection. Next, per-instruction hashing (step 3) and per-syntax integration (step 4) are applied. Finally, Docker syntax and shell syntax are separated in advance, followed by suffix array algorithm (step 5) and clone set detection (step 6) to achieve per-syntax clone detection.

Suffix array, which is a kind of data structure used in string search algorithms, consists of an array of suffixes in a search string that have been sorted in lexicographic order [20]. Suffix array has also been applied in the field of clone detection [21] [22]. Inverted index algorithm can detect clones scalably by setting the appropriate size of chunks [19]. The smaller the size of chunks in inverted index algorithm, the longer it takes to detect clones using that algorithm. On the other hand, suffix array algorithm can detect clones in linear time without predetermining the chunk size [20]. Therefore, our proposed technique uses suffix array for code clone detection. Figure 2 shows the procedure for constructing a suffix array based on the token sequence "`A B A C D A B`" and the clone detection technique. First, each suffix of the input token sequence is stored in an array via suffix enumeration (step 1). Next, the suffix array is sorted in lexicographic order (step 2). Finally, this algorithm outputs clone sets forward matching subtokens for suffixes in the suffix array (step 3). In the token sequence example of "`A B A C D A B`", there are three clones: "`A`", "`A B`", and "`B`".

Our proposed technique adopted the technique of constructing ASTs in Dockerfiles by Henkel *et al.* [11]. This is because their ASTs have tokens parsed not only at Docker syntax level but also at the shell command level, which makes it easier to normalize the tokens and separate Docker syntax and shell syntax with our proposed technique. Henkel *et al.* presents ASTs in Dockerfiles as JavaScript Object Notation (JSON) format. They take Dockerfiles as input and output ASTs according to the following procedure. First, they convert Dockerfiles into concrete syntax trees at the Docker syntax level. Next, they construct syntax trees at the shell script grammar level for the `RUN` instruction whose argument is shell syntax. In their technique, Bash Shell, which is generally used in Dockerfiles, is adopted as shell script for parsing. Finally, shell command is parsed to construct AST, which is output in JSON. Note that Henkel *et al.* limit the

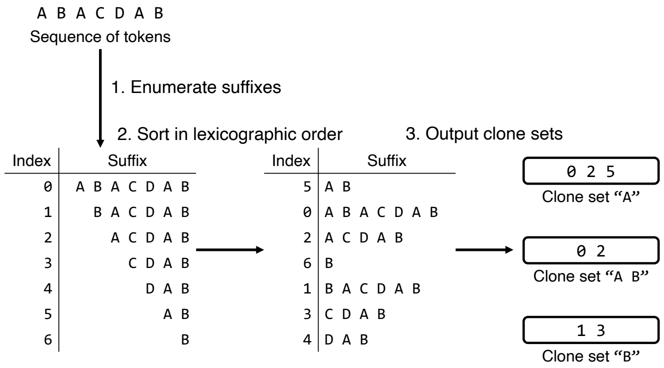

Fig. 2: Procedure for constructing a suffix array and clone detection technique

parsing to the top 50 shell commands that are most frequently used in Dockerfile.

In the following sections, we discuss the tokens to be normalized in step 2 (Section III-A) and the reason for pre-separating Docker syntax and shell syntax in step 4 (Section III-B).

A. *Considering Normalized Tokens*

To detect Type-2 clones, it is necessary to properly normalize the tokens in source codes. In general clone studies targeting Java, C and other similar programming languages, identifier names such as variable names and function names are often selected as the normalization targets [23]. This is because variable and function names are a kind of label information, and the same behavior can be realized even if the labels are different. This is also true in Dockerfiles, where there are temporary variables (variables declared with `ARG` instructions), etc., that are subject to normalization. Furthermore, in terms of the identity of the behavior in source codes, the order of parameters and options in shell script (`ls -l .` and `ls . -l`) and the aliases of option names (`ls -list` and `ls -l`) in shell script, there are many other notations that can achieve the same behavior. Hence, Type-2 clones in Dockerfile can be detected by properly normalizing these tokens.

This study defines the tokens to be normalized as shown in Table I and the tokens that are not to be normalized as shown in Table II. Figure 3 shows an example of token normalization using our proposed technique. One of the tokens to be normalized is the order parameters and options in shell commands. Shell commands define the order of the parameters and options for each commands. Some commands allow the order of parameters and options to be different. As a result, commands that have the same behavior despite having variations of parameters and options can exist in multiple Dockerfiles. `RUN apt install --yes --no-install-recommends` and `RUN apt install --no-install-recommends --yes`

```
3   FROM ubuntu:18.04
    ...
16  ARG PGP_KEYSERVER=ha.pool.sks-keyservers.net
    ...
59  RUN set -eux; ¥
60    ¥
61    savedAptMark="$(apt-mark showmanual)"; ¥
62    apt-get update; ¥
63    apt-get install --yes --no-install-recommends ¥
      ...
72    ; ¥
73    rm -rf /var/lib/apt/lists/*; ¥
      ...
83    for key in $OPENSSL_PGP_KEY_IDS; do ¥
84      gpg --batch --keyserver "$PGP_KEYSERVER" --recv-keys "$key"; ¥
85    done; ¥
      ...
255   chown -R rabbitmq:rabbitmq "$RABBITMQ_HOME"; ¥
      ...
    ...
```

(a) Before normalization

```
3   FROM ubuntu:$$TAG
    ...
16  ARG $$VAR=ha.pool.sks-keyservers.ne
    ...
59  RUN set -e -u -x; ¥
60    ¥
61    $$VAR="$(apt-mark showmanual)": ¥
62    apt-get update; ¥
63    apt-get install --no-install-recommends --yes ¥
      ...
72    ; ¥
73    rm --force --recursive $$PATH; ¥
      ...
83    for $$VAR in $OPENSSL_PGP_KEY_IDS; do ¥
84      gpg --batch --keyserver "$$OP_ARG" --recv-keys "$$OP_ARG"; ¥
85    done; ¥
      ...
255   chown -R $$USER:$$GROUP "$RABBITMQ_HOME"; ¥
      ...
    ...
```

Brown: Normalization of tokens
Bule-green: Alias of options
Underline: Separate or reordering options
Gray: Remove options

$OPENSSL_PGP_KEY_IDS is not normalized because it is the environment variable

(b) After normalization

Fig. 3: Example of normalization for an actual Dockerfile [1]

are examples of clones with variations of the order of options.

The option controlling logs and the option controlling interactions are also contained in the tokens to be normalized in our proposed technique. The options for controlling logs, such as the log output option `--verbose` in `wget` command and the error output option `--show-error` in `curl` command, only control the log output. The presence or absence of these options does not change the behavior of Dockerfiles. Additionally, assuming that the Dockerfiles have been built without errors, the options for controlling interactions such as `--force`, which is a force execution option for `rm` command, do not change the behavior of Dockerfiles with or without those options. Therefore, our proposed technique also normalizes the options for controlling logs and the options for controlling interactions. These options are removed during the normalization process because their presence or absence do not change the behavior of Dockerfiles. For example, the command `RUN curl -fsSL $URL` has four options: the `--fail` option (`-f`) to prohibit log output in case of connection failure, the `--silent` option (`-s`) to prohibit log output, the `--show-error` option (`-S`) to allow log output

[1] https://github.com/docker-library/rabbitmq/blob/master/Dockerfile-ubuntu.template

TABLE I: Normalized tokens

Token	Reason and example of normalization
Temporary variables	Temporary variables do not affect the external processes in Docker container. `ARG tag=bionic` → `ARG $$VAR=bionic`
File paths	Variations of file paths do not affect behavior. `cd webapps/convertigo` → `cd $$PATH`
URLs	URLs do not affect resources outside Docker container. `wget http://example.com/app/app.tar.gz` → `wget $URL`
Username and group names	Variations of usernames and group names do not affect behavior. `USER spark:spark` → `USER $$USER:$$GROUP`
Tags included `FROM` instruction	Differences in container functionality among variations of tags are insignificant. `FROM ubuntu:16.04` → `FROM ubuntu:$$TAG`
Arguments of options	The roles of tokens set as arguments are assigned for each option. `gpg --keyserver $KEY_SERVER` → `gpg --keyserver $OP_ARG`
Order of parameters and options	Variations of the order of options allow for equivalent commands to be executed. `set -xe` → `set -e -x`
Aliases of options	Aliases of options allow for equivalent commands to be executed. `rm -fr` → `rm --force --recursive`
Options for controlling logs	Variations with or without options for controlling logs do not affect behavior. `wget -q` → `wget`
Options for controlling interactions	Variations with or without options for controlling logs do not affect behavior if Dockerfiles have no errors. `apt-get install wget -y` → `apt-get install wget`
Options for controlling output files	Variations of file paths do not affect behaviors. `wget -o $file` → `wget`

TABLE II: Denormalized tokens

Tokens	Reason
Environment Variables	Environment variables affect external processes in the Docker container.
Images of the Docker Container	Each image contains a different package manager.
Network Ports of the Docker Container	Variations in port numbers affect processes outside the Docker container.

only in case of error, and the `--location` redirection option (`-L`). The `--fail`, `--silent`, and `--show-error` options are classified as options for controlling logs, and the `--location` option is an option controlling interaction. Therefore, the command `RUN curl -fsSL $URL` is normalized to `RUN curl $URL`.

B. Docker Syntax and Shell Syntax Segregation

Dockerfile is a nested language that contains the shell syntax [11] [12]. We expect to derive patterns that do not depend on a specific Dockerfile by detecting clones in each syntax. Therefore, we separate Docker syntax and shell syntax to detect Type-2 clones. Clones that contain both Docker syntax and shell syntax are also detected because such clones that contain both syntaxes are useful.

IV. Experiments

This study confirms that our proposed technique can detect Type-2 clones of Dockerfiles and patterns. As a comparison, we use the Type-1 clone detection technique without normalization in our proposed technique, which is referred to hereafter as the "no-normalization" detection technique. The metrics are the number of detected clone segments, the number of detected clone sets, the number of clone segments per detected clone set, the length of clone segments per detected clone set, and the precision. The number of clone segments per clone set and the length of clone segments per clone set by our proposed technique are expected to increase since Type-2 clones include Type-1 clones. Precision is the rate of candidates that are actual clones among the detected clones. The higher the precision is, the fewer false clone are detected by the clone detection technique. Therefore, a higher precision of a clone detection indicates a superior clone detection technique. We sampled the detected clone sets to a confidence level of 95 % and a confidence interval of 5 % for each techniques. The precision is based on visual confirmation because the set of true clones in the target Dockerfiles is unknown.

A. Subjects

This experiment targets 4,817 Dockerfiles contained in 725 repositories that are popular on GitHub. In Docker, there is a practice that the entire flow of container building is often described in a template file, within which the defining minor differences between distributions and versions in temporary variables that can be replaced with text are defined [10]. The target Dockerfiles are automatically generated by executing or compiling this template. Technically, these automatically generated Dockerfiles are Type-2 clones, but since they are obvious clones that have been generated automatically, they should not be detected. Therefore, for repositories that contain template files, we exclude the Dockerfiles in the repository and use the template files themselves as clone detection targets.

B. Results

1) Number of clones and precisions: Table III shows the experimental results with the number of clone segments, the number of clone sets, the number of clone segments per clone set, the length of clone segments per clone set, and the precision detected in this experiment. Comparing our proposed and no-normalization detection techniques, per detected clone set, our proposed technique has a larger number of clone segments and a longer length of clone segment than those for the no-normalization detection technique. The increased number and lengths of clone segments per detected clone set identified by our proposed technique met expectations. On the other hand, our proposed technique detected fewer clone sets than the no-normalization detection technique. The reason why our proposed technique detects fewer clone sets than the no-normalization detection technique is that segments that are not related to Type-1 clones are detected as Type-2 clones by the normalization process. As a result, independent clone sets in Type-1 are considered as equivalent clone sets in Type-2.

Comparing the precision of our proposed technique with that of the no-normalization detection technique, the presicion of our proposed technique is lower than that of the no-normalization detection technique. The more normalization is applied, the lower the precision tends to be in general [4]. Generally speaking, Type-2 clones are more difficult to detect than Type-1 clones, and Type-3 clones are even more difficult to detect than Type-2 clones. However, our proposed technique has a high precision of 95 % and is useful as a Type-2 clone detection technique. However, the high precision of 95 % achieved by our proposed technique demonstrates its usefulness as a Type-2 clone detection technique, particularly since, as reported by Sheneamer *et al.*, the precisions of most clone detection techniques for general programming languages are less than 50 % [4]. It should also be noted that the reason why the precision of our proposed technique is higher than that of clone detection techniques for general programming languages is that, unlike general programming languages,

TABLE III: Number of clones and precisions

	Proposed technique (no-normalization)			Proposed technique (normalization)		
	Docker syntax	Shell syntax	Both syntaxes	Docker syntax	Shell syntax	Both syntaxes
# of clone segments	217,766	433,161	930,134	240,639	458,737	987,597
# of clone sets	50,608	87,126	204,840	50,756	84,011	203,103
# of clone segments per clone set	4.30	4.97	4.54	4.74	5.46	4.86
Length of clone segments per clone set	18.36	11.96	20.78	18.64	12.63	21.30
Precision	100.00 %	98.70 %	99.48 %	95.31 %	98.44 %	96.09 %

```
1 FROM scratch
2 MAINTAINER James Mills, prologic at shortcircuit dot net dot au
3 ADD rootfs.tar.xz /
4 CMD ["/bin/bash"]
```

(a) Clone segments in a Dockerfile of CRUX image [4]

```
1 FROM scratch
2 MAINTAINER Vlad Glagolev <stealth@sourcemage.org>
3 ADD smgl-stable-0.62-docker-x86_64.tar.xz /
4 CMD ["/bin/bash"]
```

(b) Clone segments in a Dockerfile of Source Mage image [5]

Fig. 4: Distribution building pattern for Type-2 clone detected by our proposed technique

```
3  RUN SWIPL_VER=7.5.11 && ¥
       ...
12     && cd swipl-$SWIPL_VER && ./configure && make
   && make install ¥
```

(a) Clone segments in a Dockerfile of SWI-Prolog image [6]

```
49 RUN wget https://github.com/protocolbuffers/protobuf/releases/
   download/v3.7.1/protobuf-cpp-3.7.1.tar.gz ¥
       ...
51     && cd protobuf-3.7.1 ¥
52     && ./configure --disable-shared --with-pic && make
   && make install && ldconfig ¥
```

(b) Clone segments in one Dockerfile on GitHub [7]

Fig. 5: Software installation pattern using make commands for Type-2 clone detected by our proposed technique

Docker syntax and shell syntax assign runtime roles to tokens.

2) Examples of detected Type-2 clone: We visually derived some Dockerfiles patterns from among the Type-2 clones detected by our proposed technique in this experiment. Figure 4 and Figure 5 show some of the derived patterns.

Figure 4 shows the pattern used for building a new distribution on a container. Dockerfiles copy the tar archive from the host OS to an empty container, called scratch, and then extract the tar archive on the container to build a new distribution. Note that the `MAINTAINER` instruction sets the author as metadata in the container. We consider the `MAINTAINER` instruction to be a comment when it is used in general languages to detect clones because this instruction does not change the behavior of the container. This pattern was derived as 13 clone segments from 5 repositories.

Figure 5 shows the software installation pattern using `make` commands. Both Dockerfiles configure a Makefile under the directory where the target software is located after being changed to that directory. Next, the binary file generated by `make` command is installed into the directory specified by `install` label in the Makefile. The `--disable-shared` option in `configure` command (line 52 of Figure 5(b)) controls the output of dynamic and static libraries while the `--with-pic` option in the same command controls the output of position-independent codes. These option are classified as options for controlling output files. This experiment detects line 12 of Figure 5(a) and lines 51-52 of Figure 5(b) as Type-2 clones because these options are classified as options for controlling output files. This pattern was derived as 12 clone segments from 5 repositories.

[4]https://github.com/cruxlinux/docker-crux/blob/master/Dockerfile

[5]https://github.com/vaygr/docker-sourcemage/blob/master/stable/Dockerfile

[6]https://github.com/swi-prolog/docker-swipl/blob/master/7.5.11/alpine/Dockerfile

[7]https://github.com/eclipse-openj9/openj9/blob/master/buildenv/docker/jdk8/x86_64/ubuntu16/jitserver/buildenv/Dockerfile

V. Related Works

A. Clone Detection Techniques for General Programming Languages

Numerous clone detection techniques have been proposed for general programming languages [3] [4] [5].

Typical lexical clone detection tools include CCFinder [24] and Clone Miner [21]. CCFinder [24] detects token-based clones using suffix tree algorithm [25]. Clone Miner [21] detects token-based clones using suffix array algorithm [20]. The goal of this study is to derive patterns in Dockerfiles. We focus on syntactical clone detection rather than lexical detection because it is anticipated that clone detection in Dockerfile instructions enable pattern derivation in Dockerfiles.

Falke *et al.* detect AST-based clones using suffix tree algorithm [26]. This algorithm can detect clones in linear time and space. Suffix array algorithm [20] is more memory efficient than suffix tree algorithm while the two algorithms take the same amount of time. Therefore, our proposed technique uses suffix array algorithm to detect clones.

B. Clone Detection for Various Other Languages

Clone detection research has also been tackled in various languages other than programming languages, such as build tools [13] and requirements specifications [14]. McIntosh *et al.* detects Type-1 clones for build systems such as CMake/Autotools (C/C++) and Ant/Maven (Java) [13]. Juergens *et al.* detects clones with exact word matches (Type-1) in the quality assessments of software requirement specifications [14]. Bellon *et al.* classifies clones into three types (Type-1, Type-2, and Type-3) based on the degree of similarity to their correspondences while assuming a general programming language [3]. On the other hand, clones detected in those languages is mostly an exact duplicate (Type-1) because the classification of clones in source code written in languages other than

general programming languages are undefined. Problems caused by clones in those languages may be missed because they do not detect clones that are not perfect matches. Therefore, this study defines and detects Type-2 clones in Dockerfiles, which is one of the source codes that describe the procedure for building a virtual environment.

VI. Conclusion

In this study, we proposed definition and a detection technique for Type-2 clones in Dockerfiles. We conducted experiments on 4,817 files in 725 GitHub repositories. The experiment results showed that our proposed technique was effective because it achieved the precisions higher than 95 % for Docker syntax, shell syntax, and both syntaxes. We were also able to visually derive Type-2 clones that could be patterns, such as those used for building a new distribution on a container and software installation using `make` commands.

As a future work, we plan to propose a technique for detecting clones with statement-level modifications (Type-3) in Dockerfiles because some such clones that are caused by variations in shell commands that have the same behavior, such as package managers or variations in the order of Docker instructions or shell commands.

Acknowledgment

This research was supported in part by JSPS KAKENHI Japan (Grant Numbers: JP21H04877, 20H04166).

References

[1] Y. Jiang and B. Adams, "Co-evolution of Infrastructure and Source Code - An Empirical Study," in *Proc. Working Conference on Mining Software Repositories*, 2015, pp. 45–55.

[2] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "What is DevOps? A Systematic Mapping Study on Definitions and Practices," in *Proc. the Scientific Workshop Proceedings of XP2016*, 2016, pp. 1–11.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[4] A. Sheneamer and J. Kalita, "A Survey of Software Clone Detection Techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 2016.

[5] H. Min and Z. Li Ping, "Survey on Software Clone Detection Research," in *Proc. International Conference on Management Engineering, Software Engineering and Service Sciences*, 2019, pp. 9–16.

[6] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting Working Code Examples," in *Proc. International Conference on Software Engineering*, 2014, pp. 664–675.

[7] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, "Proactive Clone Recommendation System for Extract Method Refactoring," in *Proc. International Workshop on Refactoring*, 2019, pp. 67–70.

[8] M. Mondal, C. K. Roy, and K. A. Schneider, "An Exploratory Study on Change Suggestions for Methods Using Clone Detection," in *Proc. International Conference on Computer Science and Software Engineering*, 2016, pp. 85–95.

[9] A. Monden, S. Okahara, Y. Manabe, and K. Matsumoto, "Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations," *IEEE software*, vol. 28, no. 2, pp. 42–47, 2010.

[10] M. A. Oumaziz, J.-R. Falleri, X. Blanc, T. F. Bissyandé, and J. Klein, "Handling Duplicates in Dockerfiles Families: Learning from Experts," in *Proc. International Conference on Software Maintenance and Evolution*, 2019, pp. 524–535.

[11] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "A Dataset of Dockerfiles," in *Proc. International Conference on Mining Software Repositories*, 2020, pp. 528–532.

[12] ——, "Learning from, Understanding, and Supporting DevOps Artifacts for Docker," in *Proc. International Conference on Software Engineering*, 2020, pp. 38–49.

[13] S. McIntosh, M. Poehlmann, E. Juergens, A. Mockus, B. Adams, A. E. Hassan, B. Haupt, and C. Wagner, "Collecting and Leveraging a Benchmark of Build System Clones to Aid in Quality Assessments," in *Proc. International Conference on Software Engineering*, 2014, pp. 145–154.

[14] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can Clone Detection Support Quality Assessments of Requirements Specifications?" in *Proc. International Conference on Software Engineering*, 2010, pp. 79–88.

[15] Portworx, "Annual Container Adoption Report," https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf, 2019, [Online; accessed 18. Jan. 2021].

[16] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An Empirical Analysis of the Docker Container Ecosystem on GitHub," in *Proc. International Conference on Mining Software Repositories*, 2017, pp. 323–333.

[17] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study," *IEEE Access*, vol. 8, pp. 34 127–34 139, 2020.

[18] K. Hanayama, S. Matsumoto, and S. Kusumoto, "Humpback: Code Completion System for Dockerfiles Based on Language Models," in *Proc. Workshop on Natural Language Processing Advancements for Software Engineering*, 2020, pp. 1–4.

[19] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-Based Code Clone Detection: Incremental, Distributed, Scalable," in *Proc. International Conference on Software Maintenance*, 2010, pp. 1–9.

[20] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *Siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[21] H. A. Basit and S. Jarzabek, "Efficient Token Based Clone Detection with Flexible Tokenization," in *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007, pp. 513–516.

[22] G. Li, Y. Wu, C. K. Roy, J. Sun, X. Peng, N. Zhan, B. Hu, and J. Ma, "SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration," in *Proc. International Conference on Software Analysis, Evolution and Reengineering*, 2020, pp. 272–283.

[23] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[24] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[25] D. Gusfield, *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[26] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

[27] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of discrete algorithms*, vol. 2, no. 1, pp. 53–86, 2004.