IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received February 24, 2019, accepted March 11, 2019, date of publication March 15, 2019, date of current version May 28, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2905424

An Empirical Case Study on the Temporary File Smell in Dockerfiles

ZHIGANG LU[1,2], JIWEI XU[3], YUEWEN WU[1,2], TAO WANG[1], AND TAO HUANG[1]
[1]Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China
[3]School of Computer Science, University College Dublin, Dublin, Ireland

Corresponding author: Jiwei Xu (xujiwei@gmail.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1400804 and Grant 2017YFC0804407, in part by the National Natural Science Foundation of China under Grant 61602454 and 61872344, in part by the Beijing Natural Science Foundation under Grant 4182070, and in part by the Youth Innovation Promotion Association of the Chinese Academy of Sciences Fund under Grant 2018144.

ABSTRACT Docker is widely used in data centers to host services. The docker image adopts a hierarchical storage architecture, which means that the docker image is composed of a set of filesystem layers. In the image building process, only the top layer is read-write, while the bottom layers are all read-only. However, temporary files are often used in the image building process. Nevertheless, if a temporary file is imported and removed in different layers by a careless developer, it will lead a file redundancy. We termed this problem ‘‘temporary file smell.’’ This smell leads to larger-size images, which seriously restricts the efficiency of image distribution and thus affects the scaling ability of services in facing of sudden high load. To address this problem, we make an empirical case study to the real-world Dockerfiles on DockerHub. Based on the case study, we summarize four different smell patterns and propose a state-depend static analysis method to detect this kind of smells. We also provide three feasible fixing methods as selective options to eliminate the temporary file smell.

INDEX TERMS Abstract syntax tree, Docker, Dockerfile, static analysis, temporary file smell.

I. INTRODUCTION

Container technology is widely used in all kinds of data centers due to its powerful elastic scaling ability. It is an enabling technology for supporting the micro-service architecture. As the de facto standard of container technology, Docker plays an important role in hosting micro-service instances. Docker [1] is a lightweight resource management tool that is widely used to host distributed applications in datacenters nowadays. Docker image is the carrier of the distributed application software. To start an instance of a service, its corresponding image must be pulled from the image repository to the host machine through the network. A fast image distribution is important to the quality of service. In this paper, we try to summarize and eliminate human errors that lead to an increase of image size in building process. Since this kind of human errors can result in a large number of temporary files left in images, we call it **t**emporary **f**ile smells (TF smell), which is caused by careless use of Dockerfiles. It is against the best practice [2]. Although some tools (such as Linter [3]) can be used to check file syntax according to best practice attributes. However, it does not care about the real cause why TF smell occurs, thus cannot resolve the problem completely. To clarify the harmfulness and universality of TF smell, we go through real world cases and make some studies. Actually, this kind of careless use has been discussed in many technology forums in ways of ‘‘how to build a smaller image?’’. The way they talked is very like the best practice attributes. However, there is no definite term to describe this problem and no systematic research on this problem. In this paper, we study five research questions that help to understand, detect and eliminate TF smells. These are our initiative value contribution to the TF smells problem.

Dockerfile 1: JDK1.8

```
1: FROM centos:7
2: COPY jdk-8u171-linux-x64.tar.gz .
3: RUN tar zxvf jdk-8u171-linux-x64.tar.gz
```

As we know that a docker image is usually built according to the script file that named ‘‘Dockerfile’’ (similar to the

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

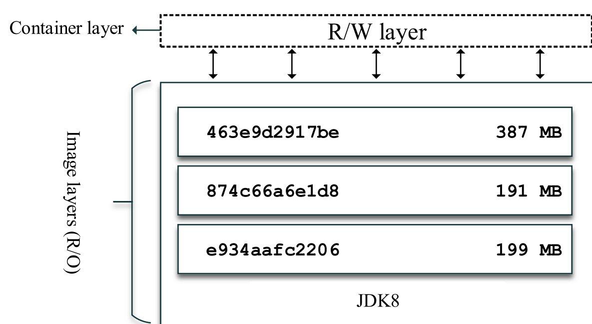

FIGURE 1. **Docker image layers of JDK 1.8.**

Makefile). The TF smell we studied in this paper is referred to the smell that occurred by misunderstanding (including unintentional) the relational mapping from the syntax and semantics of Dockerfile to the Union File System [4] that used by docker image. [1] Dockerfile syntax defines some keywords which can guide the image builder to use certain data resource and control logic to construct an image.

A docker image is a series of layers stacked. Each layer is created according to an instruction in the image's Dockerfile. A layer is a standalone filesystem. Consider as the Dockerfile 1. This Dockerfile creates an image with JDK8. It contains three instructions, each of them creates a layer. The FROM instruction starts the building process by creating a layer from the centos:7 image. The COPY instruction adds the *jdk-8u171-linux-x64.tar.gz* file which locates in the docker client's current directory to the image. The RUN instruction extracts the archive file by using tar command.

Layers of a docker image are read-only. Each layer is an independent file system. When instantiating, an additional read-write layer is added upon the layer stack. This additional layer is called "container layer". All the filesystem changes (e.g. write, modify, delete) are recorded in the container layer. The upper read-write layer distinguishes the docker image and the docker container. The docker storage driver uses copy-on-write technology to support the layered filesystem. Docker container can be saved as an image through specific operations that put the container layer upon the image layer stack and make it read-only. When executing a Dockerfile, it repeatedly performs the three operations, *adds a read-write layer*, *modifies it* and *converts it to read-only*, for each instruction. Figure 1 illustrates the image created according to Dockerfile 1. The three layers are created according to the three instructions in Dockerfile 1. In this example, the *jdk-8u171-linux-x64.tar.gz* is a temporary file. The whole image size is about 777MB, however, the useless temporary file accounts for 191MB (about 25%). Consider as Dockerfile 2 that tries to delete the temporary file.

Figure 2 illustrates an instance of the image built according to Dockerfile 2. It adds a new layer which records the deletion of the *jdk-8u171-linux-x64.tar.gz* file at the top of the image.

[1]Other kinds of temporary file redundant that are not caused by careless use are beyond the scope of this paper.

Dockerfile 2: JDK1.8

```
1: FROM centos:7
2: COPY jdk-8u171-linux-x64.tar.gz .
3: RUN tar zxvf jdk-8u171-linux-x64.tar.gz
4: RUN rm -f jdk-8u171-linux-x64.tar.gz
```

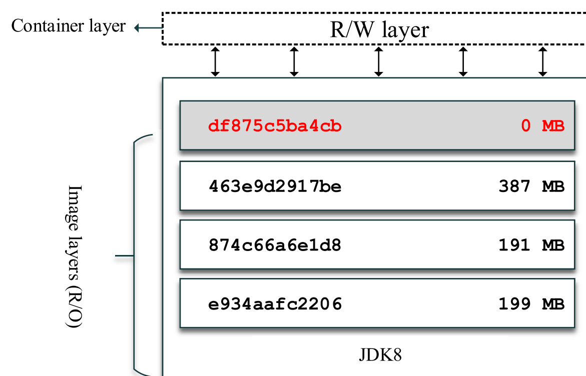

FIGURE 2. **Docker container layers of JDK 1.8.**

Thus, we cannot see this file in the instance's filesystem. The *jdk-8u171-linux-x64.tar.gz* file seems to have been removed from the filesystem of the image, but it still takes up image's storage space. That is a typical example of the TF smell in Dockerfile. In this paper, we first study the universality of the TF smells, then we give some detection methods and guiding opinions to overcome these smells.

Contributions of this paper include: 1) We summarize the programming model of the Dockerfile and classify the popular TF smell into four different patterns. 2) We address the true extent of the TF smell problem in practice through an empirical study. 3) We propose a state-depend static analysis method to detect the TF smell. 4) We summarize three different fixing methods to eliminate the TF smell.

The rest of this paper is organized as follows: Section II illustrates the motivation of this work. Section III analyzes the Dockerfile programming model and gives four different TF smell patterns. Section IV issues the TF smell problems in practice. Section V proposes a state-depend static analysis method to detect the TF smell. Section VI summarizes three different TF smell elimination method. Section VII analyzes the related work. And section VIII gives a conclusion of this paper and describes our future plan.

II. MOTIVATION

We have introduced the phenomenon of TF smells in the last section. In this section, we conduct a series of experiments to investigate the mechanism that why TF smells occurs from an aspect of the filesystem level. Our goal is to answer the following question.

- **RQ1. How does TF smell occurs in filesystem level?**

```
fs/xfs/xfs_inode.c [Create File]:/var/lib/docker/overlay2/28e0c0e5824a11a0ef587ba8002d7…/dif/add.txt inode:35765029
fs/overlayfs/dir.c [Create File]:/add.txt inode:28801
fs/xfs/xfs_inode.c [Remove File]:/var/lib/docker/overlay2/28e0c0e5824a11a0ef587ba8002d7…/difr/add.txt inode:35765029
fs/overlayfs/dir.c [Remove File]:/add.txt inode:28801
```

(a)

```
fs/xfs/xfs_inode.c [Create File]:/var/lib/docker/overlay2/17681f01c66f2154be0d1b5575a90…/dif/abc.txt inode:35765029
fs/overlayfs/dir.c [Create File]:/abc.txt inode:26603
fs/xfs/xfs_inode.c [Create File]:/var/lib/docker/overlay2/8ce2440fec4dcd53de87f4c94569b…/dif3/abc.txt inode:986983
fs/xfs/xfs_inode.c [Remove File]:/var/lib/docker/overlay2/17681f01c66f2154be0d1b5575a90…/difb/abc.txt inode:35765029
fs/overlayfs/dir.c [Remove File]:/abc.txt inode:61424
fs/xfs/xfs_inode.c [Create File]:/var/lib/docker/overlay2/05f970bb49612f80b7b2f06669de6…/diff/abc.txt inode:35765792
fs/xfs/xfs_inode.c [Remove File]:/var/lib/docker/overlay2/725fbb2dabd4df96f31dff6652f17…/dif2/abc.txt inode:18986842
```

(b)

FIGURE 3. Filtered log messages. (a) Log message for Test 1. (b) Log message for Test 2.

A. EXPERIMENTAL SETUP

We build the experimental environment based on CentOS 7 with Linux kernel 4.20. The docker storage driver is Overlay filesystem. The root file system is XFS filesystem. We separately inject logging code to both of the two filesystems and rebuild the kernel to trace the temporary file creation and deletion. For the overlay filesystem, we inject the *printk* function to *ovl_create_or_link* and *ovl_do_remove* functions in */linux-4.20/fs/overlayfs/dir.c* source code file. The two functions are used to link and unlink a user space filepath to a VFS file inode. For the XFS filesystem, we inject the *printk* function to *xfs_create* and *xfs_remove* functions in */linux-4.20/fs/xfs/xfs_inode.c* source code file. The two functions are used to link and unlink a userspace filepath to a XFS file inode.

Test 1

```
1: FROM centos:7
2: RUN echo "abc" > add.txt && rm -f add.txt
```

Test 2

```
1: FROM centos:7
2: RUN echo "abc" > abc.txt
3: RUN rm -f abc.txt
```

Then we create two test Dockerfiles, Test 1 and Test 2, and use *docker build* command to execute them. The two Dockerfiles have the same semantic. Both of them firstly create a temporary file and then delete it. The difference is that in Test 1 the creation and deletion operations are in one command connected by the && operator while in Test 2 the two operations are in two commands.

B. OBSERVATION

In Test 1, we create a temporary file "add.txt" through the output redirection operator (>) in a RUN command. Then, delete the file in the same RUN command. That means the creation and deletion operations are in the same layer. After analyzing the log message, we observed 107 creation operations and 69 deletion operations, which include both files and folders. We use the keyword "add.txt" to filter these message and get 2 creation operations and 2 deletion operations. We list the filtered log message in Figure 3(a). From the log message, we can see that the temporary file is created and removed in both XFS filesystem and Overlay filesystem. Furthermore, in the same filesystem, the inode number is the same for both creation and deletion operations. It indicates that the temporary file is actually removed from the image filesystem.

However, the situation is different in Test 2. Since the creation and deletion operations are in different RUN command, the two operations are actually performed in different layers. We analyze the log message and observed 209 creation operations and 110 deletion operations, which include both files and folders. We use the keyword "abc.txt" to filter the message and get 4 creation operations and 3 deletion operations. The filtered logs are listed in Figure 3(b). From the aspect of Overlay filesystem, the system creates the "/abc.txt" file in inode 26603 and delete it in inode 61424. It indicates the file is marked as having been removed in inode 61424, however it is still stored in inode 26603. From the aspect of ZFS filesystem, there are 3 creation and 2 deletion operations of "abc.txt". Of these operations, one creation-deletion operation pair of these operations aims at same inode with number 35765029. The second creation to inode with number 986983 stores the actual file. The rest creation-deletion operation pair with different inode number 35765792 and 18986842 simulates the file deletion operation for the top-most filesystem layer. It indicates that the file is actually stored in lower-level filesystem layer of the image, however, from the top-most aspect, we cannot see it in image's filesystem.

C. PERFORMANCE

According to the above experiment, we can detect the TF smell by analyzing the log message that the injected code printed. However, since the code is injected into the key area

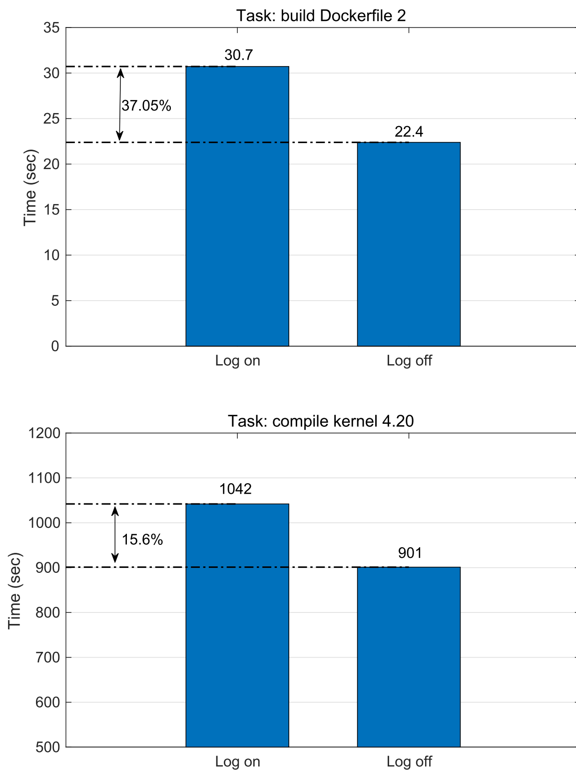

FIGURE 4. Performance loss of monitoring.

of the kernel, it can generate a large number of logs in the system running and significantly reduce system performance. We use the *time* tool to test the performance lost through two experiments. The two tests are 1) build docker image with Dockerfile 2 and 2) compile linux kernel 4.20. In order to simplify the process, we only take the execution time as the indicator and ignore other indicators such as CPU and memory usage.

The experimental results are listed in Figure 4. For the docker image building test, if we don't inject any logging code, it takes 22.4 seconds to complete the task; if we inject the logging code, it takes 30.7 seconds to complete the task. The performance loss is about 37.05%. For the kernel compiling test, if we don't inject any logging code, it takes 901 seconds to complete the task; if we inject the logging code, it takes 1042 seconds to complete the task. The performance loss is about 15.6%. Therefore, we will try to avoid using the inject code method to detect TF smells for its poor performance, because the performance loss influences the whole system and all the other tasks in that host will be affected. So, we need to develop a lightweight, non-log based detection method.

III. TF SMELL PATTERNS

In this section, we introduce the Dockerfile programming model and summarize the possible TF smell patterns based on the model. Specifically, we aim at answering the following research question:

- **RQ2. How many different TF smell patterns?**

A. DOCKERFILE PROGRAMMING MODEL

A Dockerfile can be modeled as a set of sequential structure instructions, while an instruction is composed of only one keyword (also regard as operator) and one operand. Operands could be classified into *resource type*(rt) and *command type*(ct) depending on the operator it follows. Take Dockerfile 2 as an example. The operands after **FROM** and **COPY** are resource type since they are resource names or file paths. While the operands after **RUN** are command type since they are shell scripts. A Dockerfile can be formalized as follows:

$$\forall i : D \cdot i = < rt_operator, rt_operand > \\ \vee < ct_operator, ct_operand >$$

where D represents a Dockerfile and i represents an instruction.

To study the TF smell problem, we need to address how the temporary files are imported. Firstly, the *rt_operator*s with file manipulation capacity can import temporary files into an image. We call these operators *fm_operator*. Secondly, the *ct_operand* can also import a temporary file into an image since the shell command can be used to process data resource. We call these operand *fm_operand*. Then we have the following formula.

$$fm_operator \subset rt_operator \\ fm_operand \subset ct_operand$$

We can see that the *fm_operator* and *fm_operand* are the only two ways to import a temporary file into an image.

B. PATTERNS

Now, we can summarize the TF smell patterns according to the *fm_operator* and *fm_operand* respectively. After enumerating all the *operator*s, it is easy to find that the *fm_operator* only includes **COPY** and **ADD**. However, the *fm_operand* is shell command including scripts, so it's hard to enumerate all the possible options. There are many different ways to load files into an image's filesystem through shells. The shell can locate a resource file according to URL and access it through a network or shared storage. There are many commands have such function, such as *wget*, *curl*, *scp* and *et al*. We call these shells *build-in-cmd*. We have discussed that the TF smell is usually caused by an incorrect file delete operation. In Linux system, the delete operation is usually executed through *rm* command. Based on the above analysis, we summarize 4 different TF smell patterns: **COPY/rm** pattern, **ADD/rm** pattern, **built-in-cmd/rm** pattern and **no rm** pattern.

The smell in Dockerfile 2 is a typical example of COPY /rm pattern. It's the most simple TF smell pattern. Dockerfile 2 use the COPY instruction to import a temporary file into the image and try to use the shell command *rm* to delete the temporary file.

The error mechanism of ADD/rm pattern is similar to the COPY/rm pattern. It uses the ADD operator to import a temporary file into the image and try to use the shell command

rm to delete the temporary file. However, the semantics of the two operators are different. The COPY operator only imports a file into the image without doing anything. The ADD operator can also import a file into the image, but it can uncompress some kinds of archive files, e.g. *tar, bzip2, gzip* files. Let's consider Dockerfile 3 and Dockerfile 4. Dockerfile 3 would not lead to the TF smell while Dockerfile 4 would. Since the ADD operator uncompresses the *jdk-8u171-linux-x64.tar.gz* file and only import the files after uncompressed in Dockerfile3. Although the *tar zxvf jdk-8u171-linux-x64.tar.gz* command in line 3 and the *rm -f jdk-8u171-linux-x64.tar.gz* command in line 4 will fail to execute, the image building process is not affected. However, Dockerfile 4 will cause a TF smell since the COPY operator can only import *jdk-8u171-linux-x64.tar.gz* file into the image without doing anything.

The forms of build-in-cmd/rm pattern are variety since there are many different ways to import temporary files into an image. However, there is still a common feature of this pattern. That is the import and deletion operations of temporary files are in different image layers. Dockerfile 5 illustrates an example of this pattern. In this example, the image downloads the *jdk-8u171-linux-x64.rpm* file through *wget* command in layer 2 and tries to delete it in layer 4 during the image building process. This can undoubtedly result in the TF smell.

The no/rm pattern is an implicit error that is hard to be recognized because there is no obvious evidence to indicate this smell. As a file is imported to the image without deletion, it's hard to determine will it be used in the running state or not. We will solve this problem in other work, so we don't discuss it in this paper.

IV. TF SMELLS IN PRACTICE

The examples listed above are made by us. They look very simple and are easy to avoid. Does it possibly exist in practice? In this section we will screen the docker's official registry DockerHub [5] for TF smells to answer the following research question:

- **RQ3. How common of TF smells in practice?**

Dockerfile 3: JDK1.8

```
1: FROM centos:7
2: ADD jdk-8u171-linux-x64.tar.gz .
3: RUN tar zxvf jdk-8u171-linux-x64.tar.gz
4: RUN rm -f jdk-8u171-linux-x64.tar.gz
```

Dockerfile 4: JDK1.8

```
1: FROM centos:7
2: COPY jdk-8u171-linux-x64.tar.gz .
3: RUN tar zxvf jdk-8u171-linux-x64.tar.gz
4: RUN rm -f jdk-8u171-linux-x64.tar.gz
```

A. DATASETS

We use the searching function of DockerHub to list the Top-100 pages[2] of images that tagged *isAutomated* with keywords *java*, *tomcat*, *mysql* and *hadoop* respectively, and use a web spider to crawl all the searching results' Dockerfile. The searching function returns 10 images for every page. The return page numbers of *java*, *tomcat*, *mysql* are much larger than 100, while the return page number of *hadoop* is only 62. Theoretically, we should respectively get 1000 Dockerfiles of *java*, *tomcat*, and *mysql*. However, there are some HTTP 404 (Page Not Found) errors, the actual numbers we get are less than 1000. Table 1 lists the actual numbers of Dockerfiles we got with different keywords.

Dockerfile 5: JDK1.8

```
1: FROM centos:7
2: RUN wget http://download.oracle.com/otn-pub/
   java/jdk/8u171-b11/512cd62ec5174c3487ac17c
   61aaa89e8/ jdk-8u171-linux-x64.rpm
3: RUN rpm -ivh jdk-8u171-linux-x64.rpm
4: RUN rm -f jdk-8u171-linux-x64.rpm
```

TABLE 1. Dockerfile amounts with different keywords.

Keyword	*java*	*tomcat*	*mysql*	*hadoop*	Total
Counts	809	932	906	595	3242

B. TF SMELLS DISCOVERING

We use a semi-automatic method to analysis all the samples to find the TF smells of different patterns. The analysis process has two steps.

i) For **COPY/rm** and **ADD/rm** pattern, the first step is to filter all Dockerfiles through a condition that the *rt_operator* set of a file contains **COPY** or **ADD** instruction and the *ct_operand* set of the same file contains ***rm*** command. For the **build-in-cmd/rm** pattern, since the build-in-cmd is a set that is hard to enumerate, we choose a typical case *wget/rm* smell to analysis. The first step is to filter all Dockerfiles through a condition that the *ct_operand* in instruction *A* contains *wget* command and the *ct_operand* in instruction *B* contains *rm* command, where *B* is after *A*.

ii) We manually check all the filtered Dockerfiles to judge whether a file has a certain type of TF smell. As we know that manually check is toilsome and error-prone. However, it seems hard to do this job automatically through static analysis. The reasons include: 1) the temporary filename and its path may be changed in the image building process; 2) the filename is not determined in importing, for example, a short URL; 3) the Dockerfile may use a wildcard to represent temporary file; 4) the other conditions that need to execute.

[2]The DockerHub only allows to access the top 100 pages of searching results.

TABLE 2. TF smells with different keywords.

	java	*tomcat*	*mysql*	*hadoop*	Sum
COPY/rm	2	3	5	14	24
ADD/rm	14	10	15	15	54
wget/rm	28	50	4	19	102
Total	**38**	**57**	**21**	**36**	**152**
Ratio	**4.70%**	**6.12%**	**2.32%**	**6.05%**	**4.69%**

The amounts of TF smells we found is listed in Table 2. We totally find 152 files[3] that have TF smell from 3242 Dockerfiles, the error ratio is 4.69%. Some files contain more than one kind of TF smell patterns. Actually, there is a large amount of smell that are not included in this statistic. Firstly, the *no rm* smell are not analyzed since we count the *rm* command as a filter condition in the first step. Nevertheless, we still find 15 *no rm* smells from the filtered files. Secondly, there are other types of *build-in-cmd*, such as *curl*, *apt-get* and so on. However, the 4.69% error rate is still very high. That means the TF smell is an urgent problem in the real world.

V. TF SMELL DETECTION WITH STATE-DEPENDENT STATIC ANALYSIS

From the above analysis, we can see that TF smell is very common. It will increase the size of the image and slow down the image distribution speed. In this section, we propose a state-dependent static analysis method (SDSA) to resolve the following research problem.

- **RQ4. How to detect TF smells?**

A. ABSTRACT SYNTAX TREE

To detect the TF smells, we need to build the abstract syntax tree (AST) for the Dockerfile according to the programming model. The outermost control structure is sequential since the Dockerfile can be treated as a set of instructions starting with a keyword(operator in the programming model). In the programming model of Dockerfile, an instruction is represented as an operator and operand(s) after the operator. When constructing the AST, we set the Dockerfile name as the root and set all the instructions as branches of the root. The operand with most instructions cannot be further analyzed, so their subtree depth is only two. While operands of some special instructions are linux shell scripts, they can be further analyzed. Figure 5 represents the abstract syntax tree of Dockerfile 5.

In the abstract syntax tree, we classify all the nodes into 4 categories. We use different colors to mark the 4 categories in figure 5. The red nodes are called *operator node*. The blue nodes are called *resource node*. The green nodes are called *cmd node*. And the purple nodes are called

[3]URLs of these Dockerfiles can be found in https://github.com/iscas-xujiwei/tfsmell/blob/master/urls

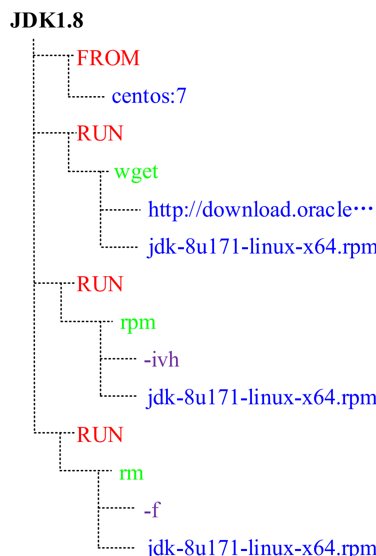

FIGURE 5. Abstract syntax tree of dockerfile 5.

parameter node. We assume all the shell scripts can be executed successfully as the developer's intent. In that case, we need not consider the syntax of parameter nodes in the next analysis. So, we remove the parameter nodes to prune the abstract syntax tree.

B. CHALLENGES IN STATIC ANALYSIS DETECTION

We first introduce a few concepts that would bring challenges to the method.

- **Filename** is the name of a file. It's unique in a directory. However, it can be repeated among different directories. The filename cannot contain some special characters includes pathname separator /. Usually, the filename has an extension, which is split by a dot character.
- **Directory** is used to contain files and sub-directories. The naming rules of a directory are almost as same as the file. It can use all characters that can be used by a filename to name a directory.
- **Directorypath** is the path from the root directory to the current directory.
- **Filepath** is a unique resource indicator. It is composed by a directorypath and a filename. Both the directorypath and filepath can be a relative path or absolute path.
- **Wildcard** is a kind of placeholder represented by a single character, such as an asterisk (*), which can be interpreted as a number of literal characters or an empty string. It is often used in file matching.

From the above concepts, we can infer some challenges that are hard to be addressed by pure static analysis.

1). It is hard to determine a path is a directorypath or a filepath, in other words, it is hard to determine a path that indicates a file or a directory. For example, the path '/root/tmp' can be either a directory or a file.

2). It is hard to determine the actual endpoint of a relative path. For example, the directorypath 'tmp/' can be referred to '/tmp/', '/root/tmp/' or other directories.

3). The wildcard and environment variables are often used in Dockerfile. So the full character matching is not always working. It must be interpreted to match the corresponding file in the analysis.

C. STATE-DEPENDENT FILE MATCH

To address the first two challenges, we propose a state-dependent file matching method in detecting TF smells. The main idea of this method is to use a state table (ST) storing the variables that represent the current status of resources. The state table is designed to disambiguate ambiguity when a resource is hard to be determined.

First, the ST records *CurrentDirectory* variable. This variable is used to record the absolute directory path of the current working directory. Its initial value is '/'. There are two different ways to change the value of *CurrentDirectory*. One is through the 'WORKDIR' operand and the other is through the shell command 'cd' after the 'RUN' operand. When the abstract syntax tree scanner meets the operator node 'WORKDIR', the *CurrentDirectory* variable is updated to the value of its child resource node, since this value represents the operand after 'WORKDIR' in the abstract syntax tree. When the abstract syntax tree scanner meets the cmd node with command 'cd' under the operator node 'RUN', the *CurrentDirectory* variable is updated to the value of the last child resource node. If the resource node value is an absolute path, the *CurrentDirectory* is set to the node value. If the resource node value is a relative path, then the *CurrentDirectory* value appends the resource node value.

Second, the ST records types (file or directory) of different paths. We use symbol 'F' to represent file path, symbol 'D' to represent a directory path and symbol 'U' to represent those paths without clear types. For example, **S1**: 'ADD jdk-8u171-linux-x64.tar.gz /usr/local/tmp/java' will be saved in ST as Table 3 illustrated; while **S2**: 'RUN tar zxvf /usr/local/tmp/java/jdk-8u171-linux-x64.tar.gz' will alter '/usr/local/tmp/java' → 'U' to '/usr/local/tmp/java' → 'D' and add '/usr/local/tmp/java/jdk-8u171-linux-x64.tar.gz' → 'F'. Table 4 lists these changes.

Third, the ST also records the common directory structure of Linux system. Such as '/etc', '/bin', '/root' and so on.

When proceeding a file match operation, we compare the absolute path of resources. If the file path is a relative path, we translate it into the absolute path according to the *CurrentDirectory*. The path type variables are used when a path is ambiguous. For example, in **S1**, '/usr/local/tmp/java' can be a file path or a directory path. If it is a file path, the compression file is added as */usr/local/tmp/java*. If it is a directory path, the compression file is copied to */usr/local/tmp/java/jdk-8u171-linux-x64.tar.gz*. That will misguide the result in complete absolute path matching. Through the ST, we can accurately estimate its real path. If the path is marked 'U' in ST, we would try all the two possibilities in file matching and file searching. Through the state-dependent file matching method, we can resolve the first two challenges.

TABLE 3. Variables in ST.

	Path	Type
1	/usr	D
2	/usr/local	D
3	/usr/local/tmp	D
4	/usr/local/tmp/java	U

TABLE 4. Variables' change in ST.

	Path	Type
4	/usr/local/tmp/java	D
5	/usr/local/tmp/java/jdk-8u171-linux-x64.tar.gz	F

D. SYNTAX INTERPRETATION

We use the plain syntax interpretation to address the third challenge. There are three kinds syntax of the Dockerfile needed to be interpreted in detecting TF smells.

The first is the wildcard. As we know that, the wildcard is a powerful tool of Linux shell. It is often used in shell scripts. There is no exception in the Dockerfile. Question mark(?) can represent any single character. Asterisk(*) can represent any number of characters (including zero, in other words, zero or more characters). These two characters are often used in file deletion. When detecting TF smells, we use the deterministic finite automaton (DFA) to test whether different character strings have the same syntax.

The second is the environment variable. There are two kinds of different environment variables. One is the Dockerfile environment variable and the other is the system environment variable. The Dockerfile environment is used in the docker image building phase. It is identified by the keyword ARG. For example, Dockerfile 6 is a real case from the DockerHub. It illustrates the usage of the Dockerfile environment variable. The system environment variable can be used in both the docker image building phase and the image running phase. It can be identified by the keyword ENV. And it can also be identified by shell commond *set* or be written to the system configuration file and loaded when it is needed.

Dockerfile 6: hadoop-spark-base

```
1: ARG VERSION=2.12.1
2: ARG ARCHIVE_NAME=scala-${VERSION}
3: ARG URL=http://downloads.lightbend.com/scala/
   ${VERSION}/${ARCHIVE_NAME}.tgz
4: RUN wget ${URL}
```

The third is the return of a shell command. The common cases are using *grep*, *sed* or *awk* to obtain or create a

TABLE 5. Detection ratio of different TF smells.

	COPY/rm	ADD/rm	wget/rm	Total
TF Smell Files	24	54	102	152
Detected Files	24	54	102	152
Detected Smells	36	54	116	206

character string. The possibility of this kind of usage exists, but it is relatively rare in the practice. To address this problem, we need to use the dynamic analysis method.

E. TF SMELL DETERMINATION

TF smells detection is a three-step process. First, building the abstract syntax tree. We use a syntactic analyzer to make lexical analysis and syntactic analysis on the Dockerfile and set up an abstract syntactic tree from the structure of the Dockerfile. When building the abstract syntax tree, all nodes are marked by types mentioned in subsection A of section V.

Second, semantic analyzing. We use a semantic analyzer to go through the abstract syntax tree. The semantic analyzer is used to 1) build and maintain the state table; 2) create two file lists **L1** which is used to store all files' identifiers that introduced by *fm_operator* and *fm_operand* and **L2** which is used to store all files' identifiers that deleted by *ct_operand*; 4) convert all relative paths in **L1** and **L2** to absolute paths.

Third, file matching. Apply the syntax interpretation technology to match files between **L1** and **L2**. If $\{l1 = l2 | l1 \in L1, l2 \in L2\}$, we call $\langle l1, l2 \rangle$ is a match *m*. There are three situations: 1) If $l1$ is an operand of *rt_operator*, then *m* is a TF smell; 2) When $l1$ is in a *ct_operand*, if *ct_operator*s of $l1$ and $l2$ are different, then *m* is a TF smell; 3) When $l1$ is in a *ct_operand*, if *ct_operator*s of $l1$ and $l2$ are the same one, then *m* is not a TF smell.

F. VERIFICATION

We designed an experiment to estimate the effectiveness of our TF smell detection method. Dockerfiles that have TF smells are selected as the test cases. The test cases contain 152 Dockerfiles. 24 files of them have **COPY/rm** smell, 54 files of them have **ADD/rm** smell and 102 files of them have **wget/rm** smell. We use our SDSA method to detect TF smell of these files.

Table 5 illustrates the experimental result. From the result, we can see that all different kinds of TF smells have been detected correctly. The method has detected 36 **COPY/rm** smell, 54 **ADD/rm** smell and 116 **wget/rm** smell. As we have noticed that the total number is not equivalent to the sum of the three numbers. That is because there are usually more than one kinds TF smells in a single file. We use SDSA method to detect another instance of **build-in-cmd/rm**, **curl/rm** smell, and find 3 **curl/rm** smells from the 152 Dockerfiles.

VI. TF SMELL ELIMINATION

We can accurately identify TF smells by using SDSA method. In this section, we will answer the following research question.

- **RQ5. How to fix or avoid TF smells?**

According to our observation, we classify the temporary files that have smells into 4 catalogs: i) executable file, such as *.rpm*, *.deb*, *.sh* file; ii) archive file, such as *.zip*, *.gz*, *.bz*2 file; iii) data file, such as *.sql*, *.csv* file; iv) setup configuration file.

There are two basic principles to fix and avoid TF smells. The first is to avoid using temporary files. If it's unavoidable, then the second is to import and remove temporary files in the same layer. There are differently detailed fixing and avoiding methods specific to each type of TF smells and files.

A. DIRECTLY COPY METHOD

Directly copy method transforms the temporary file into the final form before the building and copy the final form content into the container in the building. The transform process can be automatic or manual. This method is suitable for cases that only a simple transform action is performed on the temporary file. The archive files in **COPY/rm** and **ADD/rm** pattern are typical cases. Since the COPY and ADD instruction copy new files or directories from *src* and add them to the filesystem of the container at the path *dest*. They run in a separate layer. Any other operations to the temporary file must be in new layers. Once the temporary form is imported, it cannot be removed through the normal operations provided by docker. To avoid using a temporary file is an optional way to resolve this problem. These smells are always accompanied by a form transforming operation (uncompressing). Directly copy method could avoid the creation of the temporary file layer. For specific types of archive files, the ADD instruction can directly uncompress them in the importing process. It is also an optional way to use the ADD instruction instead of the COPY instruction under this situation.

The limitation of this method is that it is only suitable for the archive file. Since the archive file can uncompress outside the image, but other kinds of files must be performed (install or execute) within the image environment, because execution outside would not work for these files.

B. INSTRUCTION MERGE METHOD

Instruction merge method merges several instructions into a single one to reduce read-only layers that hold temporary files of an image. This method is suitable for **build-in-cmd/rm** pattern for all file types. Since the operations on temporary file in **build-in-cmd/rm** pattern is always scattered in different RUN instructions. Each instruction creates a read-only layer. To combine these scripts into a single instruction can remove the temporary file physically not only logically. Take Dockerfile 5 as an example, the three RUN instructions can be merged into a single instruction with logic symbol '&&'. Dockerfile 7 illustrates the script form after fixing.

This method can automatically fix all of these kinds of smells in the programming stage of Dockerfiles.

Dockerfile 7: JDK1.8

```
1: FROM centos:7
2: RUN wget http://download.oracle.com/otn-pub/
   java/jdk/8u171-b11/512cd62ec5174c3487ac17c
   61aaa89e8/ jdk-8u171-linux-x64.rpm \
   && rpm -ivh jdk-8u171-linux-x64.rpm \
   && rm -f jdk-8u171-linux-x64.rpm
```

The limitation of this method is that it's only work in the case of **build-in-cmd/rm** smell. It is not suitable for **COPY/rm** and **ADD/rm** smells, since the COPY and ADD instructions cannot be merged with the RUN instruction.

C. EXTERNAL STORAGE METHOD

External storage method requires a container to mount a volume which stores the specific file in its first initialization after building. This method separates a whole building process into several pieces. When a TF smell occurs, the normal building process stopped and a temporary image is generated. Then start a container instance with a temporary image, mount the volume that stores the specific file and perform relative operations. After that, go on the normal building process. This method is powerful to handle most kinds of TF smells. It can be down manually or automatically. Its disadvantages are also obvious. It perplexes the image building process and lower image building efficiency.

In this section, we summarize three methods to eliminate TF smells. These methods are suitable for different patterns, work on different levels and also have different limitations and disadvantages. Facing the complicated smell patterns, combining these methods reasonable could eliminate TF smells effectively.

VII. RELATED WORK

The purpose of our work is to slim the docker image and thus to speed up docker image distribution. And we use static analysis method to achieve this goal. So, we will introduce the related work from the three aspects: Docker image distribution, Docker image slim and Dockerfile analysis.

Docker image distribution. A fast distribution of the docker image means a faster scaling speed and a better service quality. How to speed up the image distribution is becoming an emergency problem [6] [7]. To resolve this problem, researchers brought many solutions, such as prefetching [7] or storage sharing [8]. These methods aim to share the same image data among different host machines through the network. They don't care about the image size.

Docker image slim. DockerSlim [9] is an open source tool to optimize the docker image. It uses a combination of static and dynamic analyses to generate smaller-sized container images. It simply identifies the files not required by the application and excludes them from the container. However, the interactive debugging or profiling tools are also removed from the filesystem. Jörg Thalheim *etc.* proposed CNTR [10], which can transparently combine two container images by using system calls of Linux kernel. CNTR provides the ability to attach new tools to the container dynamically.

Dockerfile analysis. Dockerfile runs as a script(similar to a Makefile) that defines exactly how to build an image [11]. Studies on Dockerfile mainly focus on the state [12] and evolution [13] [14] [15] of hosted applications. Schermann *etc.* [16] persist the Dockerfile and its evolutionary process into relational database through their tools developed by themselves for further analysis. Linter [3] can be used to check file syntax as well as arbitrary semantic and best practice attributes [2]. It can also combine multiple adjacent RUN instructions into one and thus can avoid some simple TF smells. However, it does not care about the real causes why TF smells occurs, thus cannot resolve complicated situations, such as the RUN instructions is partitioned by other instructions. Cito *et al.* [17] use Linter to analysis over 7000 Dockerfiles on GitHub. They also integrate quality checks into the container build process. However, they mainly focus on version pinning warnings and not good at detecting TF smells.

As far as we know, there is no systematic research on Dockerfile TF smells at present. Although the best practice [2] can help to resolve many kinds TF smells, lots of developers still make mistakes. The 4.69% error ratio of 3242 Dockerfiles is powerful evidence to prove this. So, it needs a detection method to warning the developers. TF smells result in a large number of redundant files. Analyzing its execution semantics [18] is an important means to eliminate redundant files [19].

VIII. CONCLUSIONS

Through an empirical study of Dockerfiles in DockerHub, we find that the TF smells widely exist. We classify TF smell into four different types and address three of them in this work. This paper proposes state-dependent static analysis method to detect TF smells. Experimental results show that the method can effectively find the known type smells. Three fixing methods are proposed to eliminate the smell for different types of data and smells.

The smell addressed in this paper is very popular. However, there are still other minority smells left unresolved. For example, in the *rayyildiz/java9* case,[4] it uses the script ''apt-get install wget'' to install the *wget* tool and ''apt-get remove –purge wget'' script to remove the *wget* tool in different layers. Strictly, this kind of smells is still within the scope of this paper. However, the variant forms have myriad ways. So, it needs more prior knowledge to adopt all sorts of unexpected cases. In the future, we will try to use the dynamic programming analysis [20] based on log message to identify the TF smells and the machine learning method to gain more prior knowledge.

[4]https://hub.docker.com/r/rayyildiz/java9/~/dockerfile/

REFERENCES

[1] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[2] (Jun. 2019). *Best Practices for Writing Dockerfiles*. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[3] (Jun. 2019). *Dockerfile-Lint*. [Online]. Available: https://github.com/projectatomic/dockerfile_lint

[4] C. P. Wright and E. Zadok, "Kernel korner: Unionfs: Bringing filesystems together," *Linux J.*, vol. 2004, no. 128, p. 8, 2004.

[5] (Dec. 2018).*Dockerhub*. [Online]. Available: https://hub.docker.com/

[6] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Jul. 2018, pp. 1–8.

[7] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *Proc. 14th (Usenix) Conf. File Storage Technol. (FAST)*, vol. 16, 2016, pp. 181–195.

[8] L. Du, T. Wo, R. Yang, and C. Hu, "Cider: A rapid docker container deployment system through sharing network storage," in *Proc. IEEE 19th Int. Conf. High Perform. Comput. Commun.*, Dec. 2017, pp. 332–339.

[9] Dec. 2018. *Dockerslim*. [Online]. Available: https://github.com/docker-slim/docker-slim

[10] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *Proc. Usenix Annu. Tech. Conf. (USENIX ATC)*, Boston, MA, USA, 2018, pp. 199–212. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/thalheim

[11] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Syst. Rev.*, vol. 49, no. 1, pp. 71–79, 2015.

[12] Q. Liu, W. Zheng, M. Zhang, Y. Wang, and K. Yu, "Docker-based automatic deployment for nuclear fusion experimental data archive cluster," *IEEE Trans. Plasma Sci.*, vol. 46, no. 5, pp. 1281–1284, May 2018.

[13] R. Tommasini, B. de Meester, P. Heyvaert, R. Verborgh, E. Mannens, and E. D. Valle, "Representing dockerfiles in Rdf," in *Proc. 16th Int. Semantic Web Conf.*, vol. 1931, 2017, pp. 1–4.

[14] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang, "An insight into the impact of dockerfile evolutionary trajectories on quality and latency," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 2018, pp. 138–143.

[15] F. Hassan, R. Rodriguez, and X. Wang, "RUDSEA: Recommending updates of dockerfiles via software environment analysis," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 796–801.

[16] G. Schermann, S. Zumberi, and J. Cito, "Structured information on state and evolution of dockerfiles on github," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, Gothenburg, Sweden, 2018, pp. 26–29.

[17] 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. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 323–333.

[18] G. J. Holzmann, "Cobra: A light-weight tool for static and dynamic program analysis," *Innov. Syst. Softw. Eng.*, vol. 13, no. 1, pp. 35–49, 2017.

[19] J. Xu, W. Zhang, Z. Zhang, T. Wang, and T. Huang, "Clustering-based acceleration for virtual machine image deduplication in the cloud environment," *J. Syst. Softw.*, vol. 121, pp. 144–156, Nov. 2016.

[20] Y. Zheng, S. Kell, L. Bulej, H. Sun, and W. Binder, "Comprehensive multiplatform dynamic program analysis for java and android," *IEEE Softw.*, vol. 33, no. 4, pp. 55–63, Jul./Aug. 2016.

ZHIGANG LU was born in Suzhou, Jiangsu, China, in 1979. He received the bachelor's degree from the Tianjin Institute of Technology, in 2003, and the master's degree from The Hong Kong Polytechnic University, in 2009. He is currently pursuing the Ph.D. degree in distributed systems with the Institute of Software, Chinese Academy of Sciences.

From 2003 to 2008, he was involved in software research at Siemens, Germany, and Compal Technology. Since 2017, he has been a Researcher with the Electronic Information Center, Xi'an Jiaotong University. His research interests include container technology, multi-source, heterogeneous software data environments, and so on.

JIWEI XU received the B.S. degree in software engineering and the M.S. degree in computer software and theory from Wuhan University, Wuhan, China, in 2008 and 2010, respectively, and the Ph.D. degree in computer software and theory from the University of Chinese Academy of Sciences, Beijing, China, in 2016.

From 2016 to 2019, he has been an Assistant Research Fellow with the Institute of Software, Chinese Academy of Sciences, Beijing. He is currently a Postdoctoral Research Fellow with the School of Computer Science, University College Dublin, Ireland. His research interests include distributed systems, software engineering, virtualization technology, cloud computing, the IoT, and so on.

YUEWEN WU received the M.E. degree in software engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2013. He is currently pursuing the Ph.D. degree in software engineering with the Institute of Software, Chinese Academy of Sciences, Beijing, China. His current research interests include cloud computing, performance modeling for big data analytics, and resource management for cloud applications.

TAO WANG received the M.S. degree in computer architecture from the University of Electronic Science and Technology of China, Chengdu, China, in 2008, and the Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2014, where he is currently an Associate Professor. His current research interests include fault diagnosis, software reliability, and autonomic computing for cloud computing systems.

TAO HUANG received the Ph.D. degree from the Graduate School, Chinese Academy of Science. He is currently a Professor and a Supervisor of doctoral student with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. His current research interests include software engineering and distributed computing.