STATIC FILE PATH ANALYSIS FOR RELIABLE RESOURCE LOCATING

by

RODNEY XAVIER RODRIGUEZ, M. Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Xiaoyin Wang, Ph.D., Chair
Jianwei Niu, Ph.D.
Rocky Slavin, Ph.D.
Palden Lama, Ph.D.
Na Meng, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2022

DEDICATION

This thesis is dedicated to my brother, David Charles Rodriguez II, who battled leukemia for nearly a year before passing away due to complications. Thank you for showing me what it means to never stop trying.

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor Dr. Xiaoyin Wang, who provided me with the opportunity of working with him as a research assistant as an undergraduate and graduate student. I have gained so much knowledge from him and he has helped me accomplish so much more than I thought was possible. I would also like to thank my other thesis committee members Dr. Xiaoyin Wang, Dr. Jianwei Niu, Dr. Rocky Slavin, and Dr. Palden Lama for their support and feedback on my thesis.

May 2022

STATIC FILE PATH ANALYSIS FOR RELIABLE RESOURCE LOCATING

Rodney Xavier Rodriguez, Ph. D.
The University of Texas at San Antonio, 2022

Supervising Professors: Xiaoyin Wang, Ph.D.

Modern software systems are becoming more complex as their growing software dependencies and resources continue to inflate with each advancement; often requiring a vast collection of third-party API references, OS level dependencies, configuration files, and much more. Maintaining all of these resources can be a difficult task, especially when software supports multiple system configurations and a wide range of dependency versions. Multiple potential issues arise when software refers to these resources, such as: invalid or out-of-date URLs, file path errors, or invalid file manipulations in scripts. We are able to use static analysis to observe the flow of resources and determine the validity of operation upon these resources. This dissertation introduces an analysis framework that abstracts possible values of resource path references which can be used to help software developers maintain different types of resource referencing code. By utilizing the results produced by our framework it is possible to ensure successful execution, code transplantation, identify dependency issues and warn against potential errors. This dissertation also describes how we applied our framework to summarize the network behaviors of applications, support repairing of errors in Dockerfiles, and summarize a file system state resulting from shell script execution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

1.1 Motivation

It is common for developers to experience build issues when attempting to build software from an online repository. It is likely the case that the downloaded software was developed on a system with a different configuration. If the creators of the software have their libraries installed in a specific location or are using remote dependencies but fail to include them in their project then developers may have trouble getting the software to build successfully on their systems. It is also possible that build scripts are erroneous which may make the process of building software more difficult. It is often difficult to debug build scripts with errors in them because missing dependencies are difficult to track down especially if a specific version is required. Additionally, it is possible for build scripts to take a few seconds to a few hours to execute and may end up failing near the end of that time resulting in wasted time and a more difficult time getting the software to compile successfully. The goal of this thesis is to develop a build script analysis tool that is able identify these dependency issues and notify users of any other errors in build scripts that are related to file manipulation.

1.2 Thesis Statement

The goal of this dissertation is to develop an analysis framework that abstracts possible values of file path references and helps developers maintain different types of resource referencing code.

1.3 Contributions

In this dissertation, we present an approach for analyzing and detecting errors in build scripts using file flow analysis and abstract interpretation. We strive to save developers and researchers time in building software so that they may focus on debugging, development, dynamic analysis and utilization of the software. This is especially useful where one may wish to automatically build multiple software projects.

We have developed an extensible framework that can traverse a build script's flow of control while providing hooks at each point allowing us to create a variety of analysis. We have developed three extensions for the framework including, two extensions that calculate the prefix and postfix, respectively for each variable used in a build script, and a third extension that will catch errors in build scripts and notify a user of potential issues with a build script. We also developed a grammar for our framework so that we can express all build scripts in a uniform way. We have manually and automatically converted various real-world build scripts to our universal language and generated some results using our framework.

1.4 Organization

This dissertation is organized as follows. Chapter 2 introduces the background and related work. Chapter 3 describes our empirical study on the usage of a tool that summarizes network behavior of Android applications built on our flow analysis framework. Chapter 4 shows how we applied our framework to support recommending update of Dockerfiles. Chapter 5 demonstrates how we can use path analysis to predict precondition and postcondition of scripts to detect errors. In chapter 6 we conclude by summarizing the lessons learned and our major contributions in the paper.

Chapter 2: BACKGROUND AND RELATED WORK

The purpose of this section is to provide the background of this study and a review of related literature.

2.1 Network Traffic

In this section, we discuss the related works of our papers. These research efforts mainly fall into three categories: network behavior summarization, android code analysis, and string analysis.

2.1.1 Network Behavior Summarization

In the area of network security, an important problem is to identify which application is generating the traffic on the network. Therefore, various techniques are developed to summarize network behaviors of applications. Though the techniques are for a totally different purpose, they are relevant to the research in this paper.

Statistical-information based approaches [49] [32] [13] mainly use the statistical information or the contents of the network traffic (e.g., packet size, data transferring rate, packet intervals) to perform a protocol/domain classification of network traffic. These approaches are able to identify network traffic belonging to applications of certain domains, such as database applications, video players, etc. However, similar to port-based approaches, these approaches are also coarse-grain and cannot support application-level network-traffic classification.

Content-based approaches are able to support application-level network-traffic classification by matching the payload of network packets with pre-generated signatures of specific applications. One necessary and challenging step in these approaches is to generate signatures for large number of applications. Sen et al. [59] proposed to use content-based signatures to identify the P2P network traffic of different P2P applications. These signatures are constructed manually through careful reverse engineering the P2P applications. The other group of approaches try to extract content based network signatures of an application from a large amount of network traffic of the application. There have been many efforts in this part focusing on generating the network signatures of worms from their collected network traffic. These efforts (e.g., Autograph [43], Early-Bird [63], PolyGraph [38]) basically extract common byte flows in worms' network traffic and generate a content-based signature (in the form of a string or a regular expression) for a certain worm or a group of worms. More recently, Park et al. [54] proposed to use the Longest Common Subsequence (LCS) alogrithm to generate a fingerprint of an application from the packets' content in the application's network traffic. Recently, Perdisci et al. [55] proposed a clustering-based approach to generate a signature for a group of malware sharing similar network behavior. This approach generates signatures for various HTTP-based software (not limited to worms, but also include other software applications such as adware, spyware). Dai et al. [24] further extends this approach to Android apps. Although the above network-traces based signature-generation approaches are fully automatic during the signature-extraction phase. All of these efforts require a large amount of annotated representative network traffic for the application under study. Therefore, they all need manual generation of network traffic or the accumulation of network traffic from a monitored network, both of which require a relatively long time and much cost. Compared with the above approaches, our approach leverages and adapts string analysis techniques to statically generate the content-based signatures of android apps without requiring any annotated network traffic. This advantage is especially important for the signature generation of android apps because of the huge number of existing android apps and the rapid development of new android apps.

Network behavior summarization of applications based on system level behavior (e.g., system calls) is another well studied area. Most of the approaches in this area execute the application under monitored environment and collect system event sequences as the behavior signature of the application [15] [58]. Therefore, usually, network accesses are recorded as simple system calls without considering the content sent to or received from the network. Recently, Bayer et al. [12] proposed an approach to cluster malware based on system behavior. Their approach take into account more detailed network traffic information but the considered information still only limited high-level information such as the names of downloaded files.

2.1.2 Analysis of Mobile Applications

Our work is also related to the security analysis of mobile applications. This area is an emerging field in academic research, and some of the recent representative research efforts are presented as below. PiOS [26] is static analysis framework for iOS, which is able to check the leaking of sensitive information by combining data flow analysis and slicing techniques. Stowaway [30] is a automatic tool that is able to determine whether an Android application requests more permissions than it actually requires. The tool is based on a pre-generated mapping from Android system APIs to Android permissions. Enck et al. [29] analyzed the permission system and the permission combinations of Android System to collect a list of dangerous permission patterns and developed Kirin, a service which identifies Android application requesting dangerous permission, so that the users can be warned when installing them. Later, Enck et al. [28] further proposed ded, a de-compiler for Android application, which is able to convert Dalvik Virtual Machine code to JVM code, and then decompile the JVM code using existing Java de-compilers. As for dynamic techniques, TaintDroid [27] dynamically monitors the information flow in Android applications by tracking the propagation of taints throughout the android system. Apex [52] and TISSA [83] are two recent advancements over the current Android permission system to provide more fine-grained permission control and dynamic permission adjustability. These works mainly focus on information leaking or permissions instead of network analysis, and none of these efforts are able to generate network signatures for android apps.

2.2 Dockerfile Analysis

2.2.1 Studies and Analyses of Dockerfiles.

With the increase of software complexity and components, managing of software dependencies [50] and test dependencies [51] has become an important problem. Tufano et al. [67] studied on broken snapshots and likely causes behind broken snapshots. Recent research work on scientific artifact reproduction [14] discussed about the uses of Docker to address the challenge of operating system virtualization, cross-platform portability, and reusable software components. Cito et al. [20] discussed about the rise of Docker adoption in industry, and performed an empirical study on dockerfiles [21]. Rahman and Williams [56] 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 [39].

2.2.2 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. [6] proposed a framework to generate dependency graph of build configuration files. Al-Kofahi et al. [9] proposed a fault localization technique for make files, and SYMake [66] uses a symbolic-evaluation-based technique to detect common errors in Makefile. Following works by Zhou et al. [82] and Al-Kofahi et al. [10] try to find configuration values exercising different parts of makefiles. Shambaugh [60] developed a verifier for puppet configuration script, and Sharma et al. [62] proposed techniques to detect bad smells in configuration files. Recently, Hassan et al. studied the reproduction of building environments [34, 35], and performed AST level analysis to generate fix patch for build configuration files [36] .

2.3 Analysis and Study of Build and Deployment Failures

Xia et al. [79] studied bugs in software build systems. The paper studies how often bugs appear in software build systems. It also puts the bugs into 11 categories, compares how many bugs are found per kLOC, the severity of each bug, and the duration of time during which the bugs were found. The paper focuses on Ant, Maven, CMake and QMake. The study performed in this paper serves as a great example for why our application would be useful. A similar study may be conducted with projects that utilize a similar tool to ours. Horton and Parnin developed a technique [37] that aims to solve the dependency resolution problem by taking a runnable code snippet and installing all language-level and system-level dependencies so that we may execute the snippet without any import errors. It parses the snippet and generates a Dockerfile to install the required dependencies to install and compile package from the PyPI repository. Wolf et al. proposed an approach [78] to predict build errors from the social relationship among developers. McIntosh et al. [45] carried out an empirical study on the efforts developers spend on the building configurations of projects. Tamrawi et al. [64] proposed a symbolic-execution-based technique to analyze Make files and detect bad smells / common errors. Downs et al. [25] proposed an approach to remind developers in a development team about the building status of the project. Al-Kofahi et al. proposed an approach [11] to detect semantic changes in Make files, and later proposed an a fault localization approach [8] for Make files, which provides the suspiciousness scores of each statement in a Make file for a build error. Rehearsal [61] is a verification framework for configurations written in puppet. In particular, Rehearsal uses several static analyses to shrink the puppet abstraction models to a tractable size, and then frames determinism-checking as decidable formulas for an SMT solver. Tamrawi et al. [65] developed a symbolic execution framework, SYMAKE, for analysing make files and detect errors. SYMake first produces a symbolic dependency graph (SDG), which represents the dependencies among files during the building process. After that, for each string value in the SDG, SYMAKE provides an acyclic graph to represent its symbolic evaluation process. Adams et al. [7] presented a design and implementation of a reverse-engineering framework for build systems. Their implementation is able to build dependency graphs that can be visualized and queried for build-related data. They also discuss how we can use the build-related data for refactoring and validation of a build system.

2.4 String Analysis

Our research is also closely related to string analysis. String analysis is an improvement over data-flow analysis [40]. String analysis [18] is a static analysis technique to estimate possible values of string variables. String analysis has been applied to detecting vulnerabilities [73, 81], repair web interfaces [72], software internationalization [70], inter-component communication analysis [53], etc. Christensen et al. [19] first suggested string analysis, which is an approach for obtaining possible values of a string variable. Then, string analysis is widely used in various areas, especially for detecting and sanitizing SQL Injection vulnerabilities and Cross-Site-Scripting vulnerabilities. Halfond and Orso [33] used string analysis to detect and neutralize SQL injection attacks. Livshits and Lam [44] also applied string analysis to detect SQL injection attacks and other vulnerabilities. Later, Wassermann and Su first developed string-taint analysis [73] to more precisely detect security vulnerabilities. Kieyzun et.al. [42] further improved their approach by considering strings that flow through the database.

Minamide [46] first applied string analysis on web applications. He also first suggested to simulate string operations in the extended CFG with FSTs, and implemented a string analyzer on PHP code to predict the possible values of dynamically generated web pages. Later, Wassermann and Su first developed string-taint analysis [74] to more precisely detect the above two kinds of vulnerabilities [75]. After that, Wassermann and Su [76] further extended their work, and developed an approach to generating test cases for security vulnerabilities. Our previous work [71] [72] extends string taint analysis with conditional and dynamic features. Kieyzun et.al. [41] further improved their approach by considering strings that flow through the database. Compared to these approaches, we apply string analysis on statically generating network signatures, which is a totally different problem. We further proposed techniques to handle obfuscation and various network APIs.

2.4.1 Bug Detection of Shell Scripts

Similar to Lint [4] for C and Findbugs [3] for Java, people have also developed bug detection tools for shell scripts. These tools such as BashLint [1] and ShellCheck [5] focus on coding style issues and provide suggestions to developers. However, they are not able to detect the type of bugs that our techniques can detect.

Chapter 3: SUMMARIZING NETWORK BEHAVIOR OF ANDROID APPS FOR NETWORK CODE MAINTENANCE

Network access is one of the most common features of Android applications. Statistics show that almost 80% of Android apps ask for network permission and thus may have some network-related features. Android apps may access multiple servers to retrieve or post various types of data, and the code to handle such network features often needs to change as a result of server API evolution or the content change of data transferred. Since various network code is used by multiple features, maintenance of network-related code is often difficult because the code may scatter in different places in the code base, and it may not be easy to predict the impact of a code change to the network behavior of an Android app. In this chapter, we present an approach to statically summarize network behavior from the byte code of Android apps. Our approach is based on string taint analysis, and generates a summary of network requests by statically estimating the possible values of network API arguments. To evaluate our technique, we applied our technique to top 500 android apps from the official Google Play market, and the result shows that our approach is able to summarize network behavior for most apps efficiently (averagely less than 50 second for an app). Furthermore, we performed an empirical evaluation on 8 real-world maintenance tasks extracted from bug reports of open-source Android projects on Github. The empirical evaluation shows that our technique is effective in locating relevant network code.

3.1 Introduction

In this network era, most software access remote servers for various reasons, and mobile apps, such as Android apps, often extensively use network due to the limited computation power of mobile devices and requirements to access real-time data. For example, travel apps such as Orbitz[1] and Expedia[2] need to request availability information of hotel rooms and air tickets; messaging

[1] `https://play.google.com/store/apps/details?id=com.orbitz`
[2] `https://play.google.com/store/apps/details?id=com.expedia.bookings`

apps such as SnapChat[3] and FaceBook Messenger[4] need to exchange text / multi-media messages through network servers; even simple flashlight apps such as Tiny Flashlight[5] collect user data and send them to remote servers for usage-pattern study and advertisement. Actually, statistics on Android permissions [80] show that, the network permission is the most popular permission among Android apps, and about 80% Android apps request the network permission.

When Android apps evolve, the maintenance of network-related code is often an important task in the maintenance process. First of all, due to various app features related to the network, a substantial portion of code in Android apps is often about sending network requests and processing network responses. Furthermore, as client-side code, Android application and server-side code on the remote servers form a whole system. However, the two portions of code often do not evolve simultaneously. It is common that Android app code and server-side code are maintained by different development groups, especially when the server-side code is also responding to web user interfaces. In many other cases, an Android app my use various third-party web services (e.g., Google and Facebook services for related features, Admob services for advertisement), and the evolution of third-party services is usually out of the control of Android app developers. Client developers often use mocking techniques [51] to keep some control on these dependencies during development phase, but they finally need to adapt their code to accommodate changes in third party services.

The maintenance of network related code in Android apps is often tedious and error-prone due to two major reasons. First of all, since lots of software features in Android apps require interaction with network, network-related code often scatters in different components, and thus it is difficult to locate the code to be revised. Second, for better flexibility and re-usability of code, developers often have to dynamically concatenate constant strings and user inputs to generate a network request. The generation process often involves complicated string operations and invocations to network-related APIs. In such a scenario, developers may not have an intuitive understanding of the network

[3]`https://play.google.com/store/apps/details?id=com.snapchat.android`
[4]`https://play.google.com/store/apps/details?id=com.facebook.orca`
[5]`https://play.google.com/store/apps/details?id=com.devuni.flashlight`

requests generated by the code, as well as how their changes may affect network behaviors of the app.

To sum up, maintenance of network code is an important task in the evolution of Android apps, and developers can benefit a lot from techniques that can summarize network behaviors of android apps as more intuitive models, and techniques that can provide traceability from intuitive models to the code base. In this chapter, we propose a novel fully-automatic approach to generate such models (we referred to them as *traceable network summaries*) for an Android app. In our approach, the traceable network summary of an app describes all possible network requests with a sequence of string constants extracted from the source code.

Examples of such summaries are presented at the end of Section 3.2. All the string constants in the signatures have their code-location information attached with them so that it is easy to trace from a string in the summary to the corresponding code location. The basic idea of our approach is to statically analyzes the byte code of an android app, and to use string analysis to estimate the possible contents of the requests sent to the network[6]. Our approach is based on the observation that, the content of the sent network request are usually generated with one or more request-generating API methods (which we refer to as *network API methods* in the rest of this paper) by concatenating the arguments of these API methods. Therefore, we will be able to estimate the content of network requests, if we are able to estimate the possible values of the arguments of network APIs, and to model the network API methods on how they generate the contents of network requests.

In particular, our approach consists of the following five steps. First, for a given android app, we translate the Dalvik byte code in its apk file to Java byte code using dex2Jar [2], an off-the-shelf tool. Second, we locate in the Java byte code all the invocations of the network API methods that are in our pre-defined network API method list. It should be noted that this list is generated manually only once and then used when generating network summaries for all android apps, and for each network API method, we prepare an *API grammar template* which presents how the

[6]It should be noted that although the detailed design and implementation of our approach is for android apps, the basic idea of our approach may be applicable to other mobile apps or even PC applications

API manipulates its arguments to generate a network request. Third, we set each argument arg of these located network-API-method invocations as the input of string taint analysis [74] and perform the analysis on Java byte code to generate a string-operation grammar that estimates the possible values of arg. Fourth, for each invocation of a network API method, we combine the API method's API grammar template with the string-operation grammars of all of its arguments, to generate a *combined grammar* that is able to estimate the network-request content generated by this network-API-method invocation. Fifth, we generate the network summary of an app as a set of constant-string sequences, from combined grammars of all network-method-invocations in the app.

In this chapter, we focus on the HTTP-based network requests, which is the most popular network traffic sent by android apps. Previous statistics [80] show that, on 35,000 apps with network access permission, more than 31,000 are using HTTP/HTTPS. Furthermore, more than 70% of these apps use only HTTP, and some of the apps that use HTTPS also use HTTP and generates HTTP traffic. Therefore, when preparing the list of network API methods and grammar templates, we only consider API methods that participate in the generation of the HTTP-request contents. However, our approach is general and can be applied to other network protocols as long as we can prepare the API-method lists and grammar templates for a network protocol.

To evaluate the applicability and efficiency of our approach, we implemented a prototype tool called NetDroid, and applied the tool on top 500 android apps (requesting the network permission). Among these apps, NetDroid is able to generate signatures for 455 of them with an average processing time of 49 seconds. Furthermore, to evaluate the effectiveness of the generated network signatures, we collected 8 real-world maintenance tasks of network code from open source Android projects in Github. NetDroid was able to generate signatures for all the involved projects, and the generated signatures successfully located 11 of 14 code revision locations.

This paper makes the following main contributions.

- We identify and demonstrate the complexities in summarizing network behavior and maintaining network-related code.

- We propose a novel approach based on string analysis to statically generating tracable network summaries for android apps.
- We implement our approach as NetDroid, an automatic prototype tool.
- We apply NetDroid on top 500 real-world android apps in the official Google Play Market to evaluate its applicability and efficiency, and evaluated the quality of generated summaries with 8 real-world maintenance tasks on network code.

The rest of this chapter is organized as below. Section 3.2 presents two real-world examples to better explain the challenges and motivate our technical choice. Section 3.3 presents the overview and details of our approach. Section 3.4 presents the evaluation of our approach. Then we discuss some important issues in Section 3.5.

3.2 Example

In this section, we present two examples to better explain our problem and how our approach works.

The first code sample is from HomeActivity.class of the app t4t.power.management, that manages the power level of an Android phone, collects certain messages, and sends the content to the server. To get the code shown below, we first convert the apk file to a Jar file using dex2jar tool, and then decompiled the Jar file using JD[7]. The following code sample first concatenates the constant strings “http://ggtrack.org/SM1c?device_id=”, “&adv_sub”, with the phone number get from `getPhoneNumber()`, to generate a string URL. Then, the code initiates an HttpGet object with the URL.

```
Object localObject7 = new java/lang/StringBuilder;
String str5 = "http://ggtrack.org/SM1c?device_id=";
((StringBuilder)localObject7).<init>(str5);
str5 = getPhoneNumber();
localObject7 = ((StringBuilder)localObject7)
```

[7]`http://java.decompiler.free.fr/`. We do not decompile the Jar file but directly use the Java byte code in our approach, we decompile the code here only for the ease of understanding.

```
    .append(str5);
str5 = "&adv_sub=";
localObject7 = ((StringBuilder)localObject7)
    .append(str5);
str5 = getPhoneNumber();
localObject7 = ((StringBuilder)localObject7)
    .append(str5);
localObject7 = ((StringBuilder)localObject7)
    .toString();
localStringBuilder.<init>((String)localObject7);
HttpGet localHttpGet =
    new org/apache/http/client/methods/HttpGet;
localObject7 = localStringBuilder.toString();
localObject1 = localHttpGet;
localObject2 = localObject7;
((HttpGet)localObject1).<init>((String)localObject2);
```

The second code sample is from Y.class[8] of the app Youtube. The function of the following code is to generate a URI, which is later packaged and sent to the Internet.

```
...
Object localObject1 = new android/net/Uri$Builder;
((Uri.Builder)localObject1).<init>();
String str1 = "http";
localObject1 = ((Uri.Builder)localObject1)
    .scheme(str1);
str1 = "gdata.youtube.com";
localObject1 = ((Uri.Builder)localObject1)
    .authority(str1);
str1 = "feeds";
localObject1 = ((Uri.Builder)localObject1)
    .appendPath(str1);
str1 = "api";
localObject1 = ((Uri.Builder)localObject1)
    .appendPath(str1);
localObject1 = ((Uri.Builder)localObject1).build();
b = (Uri)localObject1;
...
```

From the two code samples above, we have the following observations. First of all, developers

[8]The class name is renamed due to obfuscation in the standard Android build process.

do manipulate strings in the code to generate the content of network requests. Second, the second code sample shows that some of the packet generation APIs (e.g. Uri.Builder) can be more complex than taking a method that takes a single string argument (e.g., `((HttpGet)g).<init>("...");`). In particular, when using such API methods to generate the content of network requests, an object will take multiple string arguments in multiple method invocations, and generate the content by concatenating the arguments. Based on the above observations, we choose to leverage string analysis as the basic technique in our approach. Furthermore, we propose techniques to handle various complex request-generation APIs. For the above example, our approach is able to generate two constant-string sequences (as a part of the summaries of the two apps) as below:

```
Sequence 1:
Host:ggtrack.org
START -> http://ggtrack.org/SM1c?device_id=
    -> &adv_sub -> END

Sequence 2:
Host:gdata.youtube.com
START -> http:// -> gdata.youtube.com
    -> /feeds -> /api -> END
```

3.3 Approach

The overview of our approach is presented in Figure 4.1. From the figure, we can see that the input of our approach is an apk file and a prepared list of network API methods with their grammar templates. The output of our approach is a network summary, which is in the form a set of string constant sequences. As mentioned in introduction, the 5 major components of our approach are Dex2Jar, network invocation handling, string taint analysis, grammar combination, and summary generation. In this section, we will first introduce the representation of tracable network summaries, and then describe in detail the design of last 4 components.

3.3.1 Representation of Network Summaries

In our paper, a traceable network summary is represented as a set of constant string sequences. Each constant string sequence has two reserved constants "START", and "END" to be the beginning and ending constants, and is in the form as below.

$$START \rightarrow C_1 \rightarrow ... \rightarrow C_i \rightarrow ... \rightarrow C_n \rightarrow END \tag{3.1}$$

It should be noted that, in our approach, we use such a sequence to present all network requests that contains C_1, ..., C_i, ... C_n in sequence. In other words, if the alphabet of characters in all network requests is Σ, the sequence in Formula 1 represents all network requests in the form as below.

$$\Sigma^* C_1 \Sigma^* ... \Sigma^* C_i \Sigma^* ... \Sigma^* C_n \Sigma^* \tag{3.2}$$

Compared with arbitrary automata used in existing string (taint) analyses [17] for value summarization, the restricted form in Formula 1 has better readability for developers, because the former often consists of hundreds of states and transitions (as illustrated in TransVis [74]), while the latter is restricted by the number of constants defined in the program. Furthermore, the summary of an app is the union of all sequences. Therefore, it is possible to map one or several sequences to a certain restful API at the server side. For multiple sequences with common prefixes, it is straightforward to combine them as a tree.

3.3.2 Handling Network-API-Method Invocations

In this subsection, we introduce how we handle network-API-method invocations. Our work includes two parts. The first part is building a list of network API specifications that are able to model the semantics of these APIs (i.e., modeling how they generate the network-packet contents by concatenating their arguments). We use *API grammar templates* to specify the semantics of each API. An API grammar template is a context-free grammar with parameters. The parameters repre-

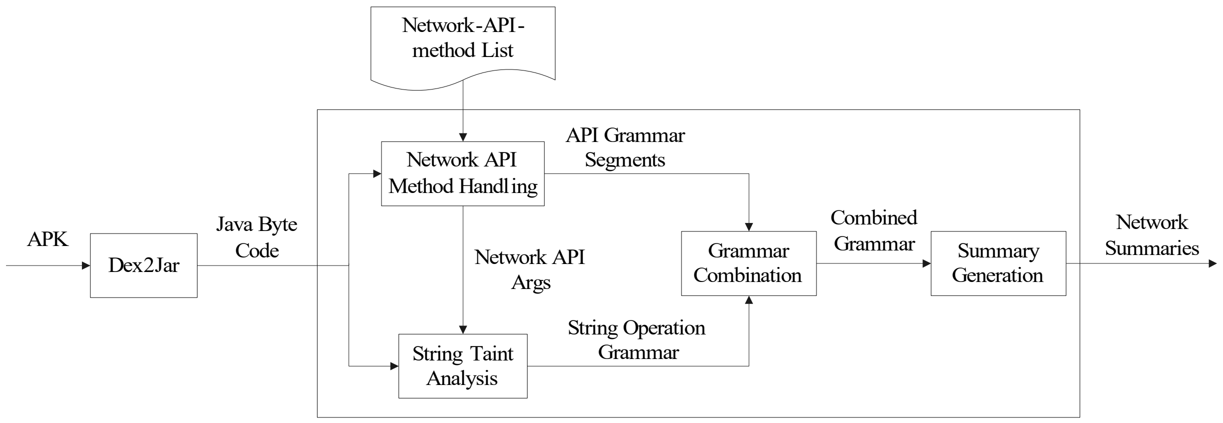

Figure 3.1: The overview of our approach

Table 3.1: Examples of Network API methods in Android system

API	Type
android.net.Uri$Builder: android.net.Uri$Builder scheme	combining
android.net.Uri$Builder: android.net.Uri$Builder authority	combining
android.net.Uri$Builder: android.net.Uri$Builder appendPath	combining
java.io.OutputStream: void write	conditional
java.io.ObjectOutputStream: void writeObject	conditional
java.io.ObjectOutputStream: void writeChars	conditional
java.io.ObjectOutputStream: void writeUTF	conditional
org.apache.http.client.methods.HttpGet: void <init>	simple
org.apache.http.client.methods.HttpPost: void <init>	simple
org.apache.http.client.methods.HttpPut: void <init>	simple
org.apache.http.client.methods.HttpDelete: void <init>	simple
org.apache.http.client.methods.BasicHttpRequest: void <init>	simple
...	...

sent the parameters of the API, the start variable of the grammar template represents the generated content of network request, and the productions in the template help to model the semantics of the API method. For example, the API grammar template for the API method: "java.net.url(String protocol, String host, String path)" is shown as below.

```
S0 -> S1 S2
S1 -> <protocol> "://"
S2 -> <host> <path>
```

The grammar shows that, in the generated URL, the value of `host` will appear before `path`, and the constant string “://” will be added to format the request. When NetDroid locates a invocation of the API, it will replace “protocol”, “host” and “path” in the grammar with their corresponding real arguments in the API invocation.

To generate the list of API grammar templates, we need to manually study the possible network libraries in android system. Since we focus on HTTP requests in this chapter, we consider only the network API methods that may generate HTTP requests. In the android system, there are three sources of network API methods: Java network libraries, Apache network libraries, and android network libraries. It should be noted that, for each android app, the android system requires all its required third-party libraries to be packaged within the app. This design decision is made to help separate apps at runtime into sandboxes for better system security. It also means that it is sufficient for us to collect the network API methods in the android system. Because the byte code of all third-party network libraries must be packaged within the apk, and can be directly analyzed.

Given the list of network API methods and their grammar templates, our network-API-method handling component locates network API invocations, and binds arguments to parameters in the grammar templates. For each network API method, we refer to the grammar template with binded arguments as *API grammar segments*. When we locate an invocation of the API, such as `URL u = new URL("http", str1, str2)`, we will generate an API grammar segment for the invocation as below. In the API grammar segment, the parameters are replaced with arguments, so that we can later calculate the context-free grammar $g1$, and $g2$ for the arguments $str1$ and $str2$ with string taint analysis, and combine the API grammar segment with $g1$ and $g2$. In the grammar segment, we use <:str1> to represent an argument $str1$.

```
S0 -> S1 S2
S1 -> "http" "://"
S2 -> <:str1> <:str2>
```

The API grammar segment can handle simple network API methods that take all arguments at one time. However, as we shown in Section 3.2, there are some more complicated network API methods. We divide these API methods into two categories: combining network APIs, and conditional network APIs. Combining network APIs are those network APIs that may generate a network packet by concatenating multiple arguments from multiple invocation sites. For example, in the our second code sample in Section 3.2, the class `Uri.builder` can generate a URL by sequentially invoke three methods: `scheme(String str)`, `authority(String str)`, and `appendPath(String str)`. Then a URI is generated based on the arguments of all the method invocations. Conditional network APIs are usually general IO API methods, and whether their arguments will be sent as the content of a network request depends on the code context. For example, the method `write(byte[] bytes)` defined in the class `java.io.OutputStream` can be used to send data to either network or the file system. Whether the method sends data to the network or to the file system depends on whether the `OutputStream` belongs to a socket or a file. In our work, we manually examined the API documentation[9] of Android SDK, and identified the network API methods based on whether at least one of their parameters are sent to the network. Table 3.1 lists some popular API-methods of each type, and we introduce as follows how we handle these two types of complicated network API methods in Section 3.3.2, and Section 3.3.2, respectively.

Combining Network API Methods

Combining network API methods are typically a group of methods defined in a request-generating class (e.g. Uri.builder). An instance (object) of the class, after initiated, will call methods in this group for one or more times to acquire all the data to be put into a network request. Therefore, to handle combining network API methods, we need to trace the life cycle of request generation instances, and build an API grammar segment for the instance as a whole. Therefore, the API grammar templates for combining network API methods need to consider the current state of the request-generating object. We present the API grammar templates of the methods in Uri.builder as below. In the template, the parameter `<head>` denotes the start variable of the current grammar, and represents the current state of the packet-generating object. In the grammar, `S[i]` denotes a new nonterminal that is different from all existing nonterminals.

```
<init>:
S0 -> ""
```

[9]`https://developer.android.com/reference/packages.html`

Algorithm 1: Concatenation of Grammar Segments for Combining Network API methods

```
Require:
    $G$ is the inter-procedure control flow graph
    $M$ is a map from network-API-methods to grammar templates
    $start$ is an initialization statement of a network-request-generating instance
Ensure:
    $S$ is the concatenated grammar segment for $start$
 1: $worklist \leftarrow \emptyset$
 2: $worklist.enqueue(start)$
 3: while $worklist \neq \emptyset$ do
 4:     $current = worklist.pop()$
 5:     $S' \leftarrow S \cup M.get(current)$
 6:     if $S'! = S$ then
 7:         $suc \leftarrow G.successors(current)$
 8:         $worklist.enqueueAll(suc)$
 9:     end if
10: end while
```

```
scheme(String scheme):
S1 -> <head> <scheme> "://"
authority(String authority):
S2 -> <head> <authority>
appendPath(String path):
S[i] -> <head> "/" <path>
```

Algorithm 1 shows the worklist-based algorithm we use to concatenate grammar segments of combining Network API methods.

During the analysis, we first locate all initializations of the request-generating objects (e.g., the invocation of `Uri.builder.<init>` at Line 2 of code sample 2 in Section 3.2). Then we use the initialization statement as the starting node for our algorithm. We first generate an API grammar segment for it according to the API grammar template. For example, we generate `S0->""` for `Uri.builder.<init>`. Then, we leverage standard context-sensitive inter-procedure data flow analysis [57] to trace the generated object obj. Along the data flow, when obj invokes a combining network API method (e.g., `schema(String scheme)`), we will merge the API grammar segment of the combining network API invocation with the current grammar segment of obj. As an example, for the code sample 2 in Section 3.2, we will have the following API grammar segment.

```
S0 -> ""
S1 -> S0 <:str1-3> "://"
S2 -> S1 <:str1-5>
S3 -> S2 "/" <:str1-7>
S4 -> S3 "/" <:str1-10>
```

When there are branches, there can be multiple start variables for obj because we will add start variables (i.e., `S[i]`) along all paths. In this case, we will add a final start variable into the grammar segment of obj that deduces all the current start variables of the grammars. Furthermore, we add a new non-terminal for each method invocation at a different location, so a same non-terminal will be added when a method invocation is analyzed for the second time, and the analysis will converge with the existence of loops in the data flow. As an example, when our analysis goes through the following code sample, it will add `S0->""` to the grammar segment at Line 1, and add `S1->S0 <:str1>` to the grammar segment the first time it goes through Line 3. The second time it goes through Line 3, `S1->S1 <:str1>` will be added, because `S1` is the current `<head>`, and the new non-terminal will still be `S1`. Apparently, the analysis will reach a fixed point the third time Line 3 is processed.

```
1  ((Uri.Builder)localObject1).<init>();
2  while(...)
3      localObject1 = ((Uri.Builder)localObject1)
           .appendPath(str1);
4
```

Conditional Network API Methods

To handle conditional network API methods, we need to determine whether the object that invokes a conditional network API method is under a network-related context (e.g., determine whether an output stream belongs to a socket). The API grammar templates for such APIs are not special, but we must correctly differentiate the network-API-method invocations that are under a network-related context and the invocations that are not. NetDroid leverages data dependence analysis on the objects that invokes conditional network APIs. Specifically, NetDroid checks whether the data flow of the object reaches any point in the byte code, where the object is related to another indication API (which indicates a network-related context). For example, when an OutputStream object *ob* has data dependency with an invocation of the API `java.net.Socket: getOutputStream()`, NetDroid will determine that all the conditional network API invocations of *ob* are real network API invocations. In the network API handling step, NetDroid will consider only the located real network API invocations, while ignore all other conditional network API invocations.

3.3.3 Apply String Taint Analysis

The third component of NetDroid uses string taint analysis to estimate the possible values of all the network arguments in the located network API invocations. String taint analysis [74] [69] is able to estimate the possible values of a given string variable in the code, and trace values back their origins in the code. By analyzing the data flow of string variables and string concatenations, for a given string variable v, string taint analysis is able to generate a context-free grammar, whose language represents the possible values of v, and whose code-location attributes on the terminals record the origin of values.

After string taint analysis is applied, NetDroid is able to generate a context-free grammar for each argument of each network-API-method invocation. As an example, for the argument `localObject2` in Line 16 of code sample 1, after this step, we can generate a grammar for it as below. The language of this grammar actually represents the possible values of `localObject2`. In the grammar, "<???>" denotes any string, because the phone number is read from the phone storage, and string taint analysis is not able to estimate it.

```
S0 -> "http://ggtrack.org/SM1c?device_id="
S1 -> S0 <???>
S2 -> S1 "&adv_sub"
S3 -> S2 <???>
```

3.3.4 Grammar Combination

The component of grammar combination, for each network API invocation inv, combines the API grammar summary of inv with the grammar of each argument of inv. This process is straightforward. We just replace the arguments in the API grammar summary of inv with the start variables of the grammar of each argument. For example, in the code sample 2, we can combine the API grammar segment shown in Section 3.3.2 with four simple grammars generated by string taint analysis, and get the combined grammar as below.

```
S0 -> ""
S1 -> S0 S1-3 "://"
S2 -> S1 S1-5
S3 -> S2 "/" S1-7
S4 -> S3 "/" S1-10
S1-3 -> "http"
S1-5 -> "gdata.youtube.com"
S1-7 -> "feeds"
S1-10 -> "api"
```

3.3.5 Extracting Tracable Network Summaries

Finally, we need to extract the network summaries from a set of combined grammars generated from the grammar combination component. To generate signatures of constant string sequences, we enumerate all the limited deduction trees of the grammar (i.e., we deduce only once for recursive nonterminals). Therefore, for each deduction tree, we generate a sequence of constant strings by ignoring the terminals which are not constant strings (i.e., “<???>”). It should be noted that, according to our definition in Section 3.3.1, ignoring non-constant strings and deducing only once for recursive nonterminals actually generates a conservative approximation of the original grammar.

Using the approach above, we can generate a set of constant-string sequences from each combined grammar. Then, we merge all these sets, and compare all these constant-string sequences to remove all the duplicate constant-string sequences. If we can find common prefixes (e.g., host names), we merge all the constant-string sequences with a common prefix as a tree for better pre-

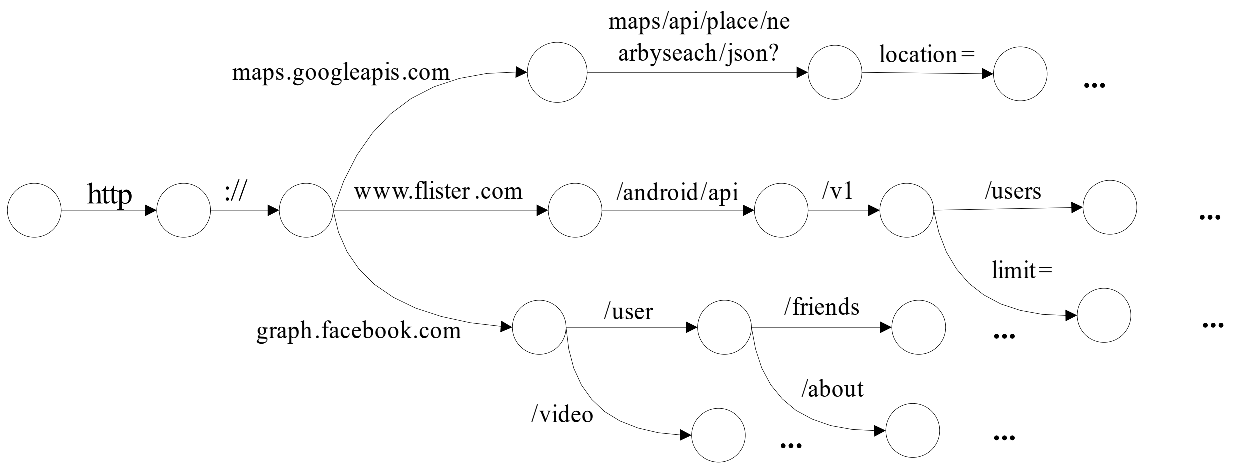

Figure 3.2: An example of tree-based network summary for Flister

sentation. Figure 3.2 shows part of a prefix tree summary generated from Flister app. This part of the summary shows that the app is accessing 3 domains, and for each domain, the summary provides the paths and parameter templates used. Thus if the parameter names or paths are changed, it would be easy for Flister developers to find what needs to be changed.

3.3.6 Trace from Summaries to Code

After a summary is generated, since all the string constants involved in the summary have their code location recorded, it is straightforward to trace from the constant strings in the summary to code locations. When the network request format needs to be changed due to server-side code changes or API changes, developers can simply trace from the string constants in the affected string-constant sequences.

3.4 Evaluation

To evaluate our approach, we implemented our approach as a prototype called NetDroid[10] base on the Soot framework [68], and carried out an experiment on the real-world apps from android market.

[10]Available at `http://xywang.100871.net/netdroid`

Table 3.2: Basic information of maintenance tasks

Task	Project	Issue No.	Description	KLOC	API	# co
1	spring-social	178	parameter deleted	35	FaceBook API	
2	restfb	201	change name of a parameter	66	FaceBook API	
3	Wizcorp	1051	change parameter value format	122	Twitter API	
4	android-simple-facebook	197	change method name	13	FaceBook API	
5	socialauth	249	parameter deleted	7	Twitter API	
6	Jasig	886	parameter becomes required ...	69	Github API	
7	caskdata	3356	id field deprecated	23	CDAP API	
8	dotCMS	386	method name change	13	CMS API	

3.4.1 Research Questions

To evaluate the effectiveness of our approach, we first need to study whether our approach is applicable and efficient on real-world apps. Then we need to evaluate the quality of the summaries generated by our approach. Specifically, we should evaluate the quality of the generated summaries in real-world maintenance tasks of network code. Therefore, we try to answer the following three research questions.

- **RQ1:** Is our approach robust and efficient enough to handle most Android projects?
- **RQ2:** Is our approach able to generate meaningful network summaries?
- **RQ3:** How effective our generated network summaries are in helping developers doing real-world network-code related maintenance tasks?

3.4.2 Applicability

To answer the first research question, we applied our prototype NetDroid on 500 android apps from the Google Play Market. The 500 apps are top ranked in Google market and request network access permission. We refer to this set of android apps as Top-500-Set below. The size of apk files in Top-500-Set ranges from 30KB to 120.6MB, and the total size of the 500 apk files is 5.7GB. We also report the size of the Jar file translated from the apk file, because the apk file usually contains not only code, but also supporting files such as figures or even video snippets. Therefore, the sizes of apk files may not be precise indications of the scalability of our approach. In contrast, the generated Jar File is more precise because it contains only Java byte code. The size of the translated jar files ranges from 7kB to 8.9MB, and the total size is 346MB.

NetDroid is able to successfully process 455 android apps, which takes a proportion of 91.0% of all the 500 android apps. For all the 45 apps, the reason why NetDroid can not process them is failure in the translation from apk files to jar files with Dex2Jar, or the loading phase of Soot (due to errors in the translated Java byte code). We further investigate the Java byte code of these classes and found that some Java byte code fails to pass the type check of Soot. Such failures are due to the imprecision in the process of translating Dalvik byte code to Java byte code. Actually, since our tool is designed for developers, such failures may not happen in reality, because developers can choose to compile their project to Java byte code and directly apply NetDroid on it without translation. We integrate the translation from apk files to Jar files in our tool because developers do not have to change their building configuration, and we are able to evaluate our tool on top android apps which we do not have source code for.

Among the 455 apps that NetDroid can successfully process, NetDroid generates an invalid summary for 33 of the apps. A summary is invalid if it does not have any constant string sequence, or all of its constant string sequences are empty. The reason for invalid network summaries is that, the value of the network arguments in the located network API invocations come from android system library so that they are estimated as any string by string taint analysis, and therefore an invalid summary will be generated. It may be helpful to build an android system model and further trace to the constant strings in the android system. However, if the values of network arguments eventually come from user input, it maybe impossible to generate precise network summaries for those parts statically.

To sum up, NetDroid is able to successfully generate valid network summaries for 422 apps from the Top-500-Set, which makes a proportion of 84.4%. Furthermore, the failing reasons of the 88 apps show that our approach has the potential to perform better in reality when source code is available.

3.4.3 Supporting Maintenance Tasks

Although we can use the top apps downloaded from the official Google Play Market to evaluate the efficiency and robustness of NetDroid, we can evaluate the helpfulness of the generated summaries only with open source Android projects, because the latter have public available version history for us to collect real-world maintenance tasks.

To perform our study, we first collected 8 real-world maintenance tasks related to network code. It should be noted that, although maintenance of network code is common, it is difficult to identify such tasks for two reasons. First of all, a lot of network code maintenance tasks are not bug fixes, because when the developers know about the changes on server-side code or third-party web services, they may direct change the code so that the error is not released to public available versions. However, Github does not allow search on code commit messages, so it is not possible for us to search for network-code-related code commits directly. By contrast, we have to search in the bug reports (Github allows searching of bug reports) to find network-code-related bug fixes. Second, network-related code is often used in various features so the bug reports related to network code may be of various forms, and it is difficult to find them with specific keywords. During our searching, we found that the API change of third-party web services is one common reason of network-code related maintenance, so we search Github bug reports with names of popular third-party web services such as Twitter, Facebook, and Google.

The collected tasks are presented in Table 3.2. In the table, columns 1-7 present the task ID, the project where the bug report is from, the bug report number, a description of the change on the RESTful API, the size of the source code base, the relevant RESTful API, and the code change locations in the code version history. We use the code revision locations in the version history as the ground truth of our empirical study.

The result of our study is presented in Table 3.3. Columns 1-9 present the task id, number of code locations reported by the summary, number of code locations actually revised, the number of true positives, false negatives, false positives, precision, recall, and F score. From the results, we have the following observations.

First of all, our approach is able to generate summaries for all the 8 apps involved in the maintenance tasks. Second, by tracing from the summaries to code, our technique is able to cover 11 of 14 revised code locations, with 47 false positive code locations in total. This indicates that, when a server restful API is changed, the developers need to examine about 5 code locations to find the actual code location to be changed, which is very reasonable in a debugging process.

It should be noted that, when tracing from constant strings in our summaries back to the code, developers may actually not revise the string constant itself, but perform the change at a specific point on the data flow from the string constant to the network packet. Therefore, NetDroid provides all code locations along the data flow path from the code location of the string constant to the network request generation APIs. Such a strategy will cause some false positives, but it will largely reduce the number of false negatives. Furthermore, since the reported code locations are still few and along the same data flow, they should not cause big burden on the developers to identify the correct location to perform the change.

We further studied the reason of false positives and false negatives generated by our approach and describe them as follows.

False Positives. Our approach generates 47 false positives in total. Among these false positives, 40 are along the data flow path from the string constant to the network API. As we mentioned above, these false positives are not very harmful because they help developers to understand how the network request is generated and decide where to change the code. The rest 7 false positives are due the imprecision of our analysis. Since our analysis uses approximation when analyzing string operations, it may mistakenly involve irrelevant string constants to the network summary.

False Negatives. Our approach generates only 3 false negatives. 1 of the false negatives in task 8 is due to the usage of reflection to perform network method calls, which we cannot handle for now. The rest 2 false negatives are due to the concatenation of user input. Our analysis reports only code locations between the code location of the affected string constants and the network API. Therefore, user input is traced only after they are concatenated with string constants. In the two false negatives, the two revised code locations are simply on the code processing just user input,

Table 3.3: Code Location for Maintenance Tasks

Task	Located	Changed	TP	FN	FP	P(%)	R(%)	F(%)
1	13	2	2	0	11	15.4	100	26.7
2	5	1	1	0	4	20	100	33.3
3	7	1	1	0	6	14.2	100	24.9
4	2	1	1	0	1	50	100	66.7
5	1	1	1	0	0	100	100	100
6	21	2	1	1	20	4.8	50	8.8
7	3	1	1	0	2	33.3	100	50.0
8	6	5	3	2	3	33.3	40	36.3

so our approach was not able to locate them.

3.4.4 Threats to Validity

The main threats to the construct validity is that, in our empirical evaluation on software maintenance tasks, our assumption of developers' knowledge may be different the developers' actual knowledge. To reduce this threat, we assume developers know only the name of changed parameters / methods in the Restful API, and we then search the network summaries and trace back to the code based on only that name. Therefore, the usage scenario of NetDroid in our empirical should not be easier than the actual scenario. The main threats to the internal validity is that, the reported evaluation results may be only applicable to the apps used in our evaluation. We use keywords of popular web services such as Facebook and Twitter to more effectively find network-code-related bug fixes. We note that these keywords may cause our results biased to maintenance tasks involving third-party web services, but we believe that such maintenance tasks are common and they are not significantly different from the network-related code maintenance tasks that do not involve third-party web services. To reduce this threat, we use a large set of top apps to generate network summaries. Furthermore, we choose apps from different domains in the study on software maintenance tasks, so that it is more likely that our results are also applicable to other apps.

3.5 Discussion

3.5.1 Limitations

Our current approach to generate network summaries has the following major limitations.

First of all, our approach is based on the network-related API methods presented in Table 3.1. Therefore, our approach cannot handle the cases where a different set of network APIs are used. This limitation can be overcome by extending our API set.

Second, since our approach is based on static analysis of Java code, it cannot handle the Android apps that send out network request with native code. Additionally, it cannot handle dynamic code features such as runtime code loading and reflection, as shown in our evaluation.

Third, our approach focuses on the generation of network requests. However, there are another portion of code that parses network responses and process the data. Such code are also network related and may evolve frequently due to evolution of server-side code or third-party web services. Our current approach is not able to support maintenance of such code.

3.5.2 Dynamic Approaches to Network Behavior Summarization

From our evaluation, we can see that, our static approach is able to automatically generate network summaries for most of the apps, and it is able to cover the whole code base of the app, so it is able to find some network behaviors that are very difficult to be revealed dynamically. However, the static approach may be not precise enough in some apps, may generate lots of false positives, some invalid summaries, and cannot handle dynamically loaded code. Therefore, dynamic approach to network behavior summarization may well complement our approach. With proper testing of the Android apps, network traffic collection, as well as tainting of data sent to the network, dynamically generated traceable network summaries may resolve some limitations of our approach.

Chapter 4: RECOMMENDING UPDATES OF DOCKERFILES VIA SOFTWARE ENVIRONMENT ANALYSIS

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 chapter, 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.

4.1 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, third-party libraries, 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 [21] 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 chapter, 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 images for both software industry and open source community, and the image configuration files for docker are called *dockerfiles*. Note that, although the implementation and evaluation of this

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

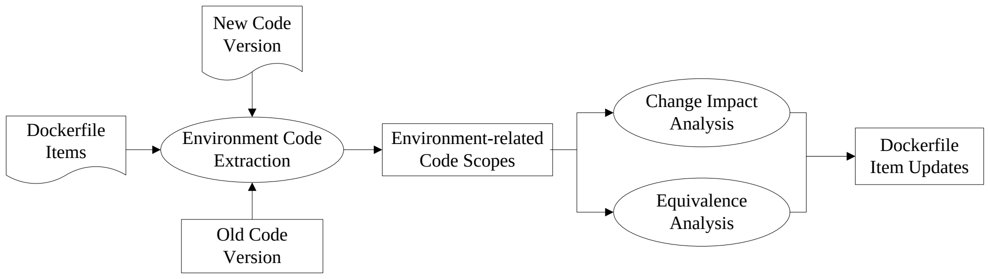

Figure 4.1: RUDSEA Overview

research focus on docker images and dockerfiles, the general approach is applicable to any Open Container Initiative (OCI) compliant container image.

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 chapter is organized as follows. First, we will introduce some background knowledge about dockerfiles in section 4.2. Then, we describe our approach and detailed techniques in section 4.3. Finally, we present our evaluation results in section 4.4.

4.2 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.

4.3 Approach

As shown in Figure 4.1, 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.

4.3.1 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 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.

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 $Init$.

$$env(str) = len(str) \geq 3 \land \exists d \in D, d.contains(str) \tag{4.1}$$

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.

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.

$$\Gamma = \{var \rightarrow L | var \in V \wedge L \subset S^*\} \tag{4.2}$$

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 [73] for two reasons as follows. First, string taint analysis (and also the original string analysis [18]) 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.

$$Gen = \bigcup_{var \in V} \Gamma(var) \cap D \tag{4.3}$$

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.

$$Scope = \bigcup_{val \in Gen} T(val) \tag{4.4}$$

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.

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.

$$Gvar = \{var | var \in V \wedge \Gamma(var) \cap Gen \neq \emptyset\} \tag{4.5}$$

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 $T_{new}(newv)$.

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.

4.3.3 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.

4.4 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.

4.4.1 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.

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

Table 4.1: Effectiveness of RUDSEA on recommending update locations

Project	**# of Actual 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

4.4.2 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.

4.4.3 Evaluation Results

To answer **RQ1**, we present our evaluation results in Table 4.1. 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 4.2 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

Table 4.2: Effectiveness of RUDSEA on recommending updates

Project	# of Actual 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

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.

4.4.4 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.

4.4.5 Studies and Analyses of Dockerfiles.

With the increase of software complexity and components, managing of software dependencies [50] and test dependencies [51] has become an important problem. Tufano et al. [67] studied on broken snapshots and likely causes behind broken snapshots. Recent research work on scientific artifact reproduction [14] discussed about the uses of Docker to address the challenge of operating system virtualization, cross-platform portability, and reusable software components. Cito et al. [20] discussed about the rise of Docker adoption in industry, and performed an empirical study on dockerfiles [21]. Rahman and Williams [56] 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 [39].

4.4.6 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. [6] proposed a framework to generate dependency graph of build configuration files. Al-Kofahi et al. [9] proposed a fault localization technique for make files, and SYMake [66] uses a symbolic-evaluation-based technique to detect common errors in Makefile. Following works by Zhou et al. [82] and Al-Kofahi et al. [10] try to find configuration values exercising different parts of makefiles. Shambaugh [60] developed a verifier for puppet configuration script, and Sharma et al. [62] proposed techniques to detect bad smells in configuration files. Recently, Hassan et al. studied the reproduction of building environments [34, 35], and performed AST level analysis to generate fix patch for build configuration files [36] .

Chapter 5: DETECTING SCRIPT ERRORS

In this chapter, we use our previously proposed static analysis framework on real world data as well as compare it to existing tools commonly used by today's developers. We evaluated our analysis on 58 docker files and the experiment shows that FiFA is able to generate an expressive directory tree and report file manipulation warning and errors.

5.1 Introduction

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

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

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

The first step of our technique is to design an abstract domain that summarizes the path-existence states of a file system. Despite the extensive research efforts [47] [23] on summarizing memory states including modeling both variable values [48] [22] and heap shapes [77] [16] [31], there have been few techniques developed to summarize file-system states, which brings two special challenges. First of all, unlike memory locations which are usually referred to by variable names and field names hard-coded in the source code, file system paths are often string values generated at run time by concatenating string variables. Second, although a file system state on path existence can be presented as a path tree, unlike the tree or graph domains used in heap shape analysis where edges can be labeled with predefined field names/indexes, the possible edge labels (folder and file names) in a directory tree are generated by the program at run time and thus are usually unbounded.

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

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

- An intermediate language FMIL that captures path-related semantics of file manipulation scripts.

- A static analysis FiFA that infers all possible directory tree states of running a given file manipulation script. Within FiFA, we use FSS automaton as abstract domains and finite state tranducers to define transfer functions.

- An evaluation on 58 dockerfiles which shows that FiFA is able to generate preconditions within reasonable amount of time.

The rest of this chapter is organized as follows. Section 5.2 is going to present a motivating example to illustrate the requirement of FiFA and just the techniques we develop. We will introduce the details of FiFA in Section 5.4, and present our evaluation setup and results in Section 5.5. Finally, we conclude in Section 6.3.

5.2 Example

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

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

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

```
  && mv testingbot-tunnel.jar /usr/local/bin \
  && java -jar /usr/local/bin/testingbot-tunnel.jar --version; \
   else echo "Testing Bot Disabled"; \
   fi
```

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

5.3 Intermediate Language

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

Since our technique uses its own intermediate language and we want to analyze Docker files in our evaluation, we needed a way to convert the Docker files to data that our framework can understand. To do this we used a Dockerfile parser to look for Docker commands that could potentially modify the file system[2]. Commands such as COPY, ADD, and RUN can affect the Docker container's file system. The COPY and ADD commands are fairly straight forward and could be translated to our framework's copy command. The RUN command is the most challenging one to transform because it could be any shell command within the container. It is possible to make almost any modification to the file system using the RUN command. There were so many possibilities and due to time constraints we had to only consider the most common RUN commands. As

[2]https://github.com/asottile/dockerfile

per Docker's Dockerfile documentation, the RUN command is run in a shell, which by default is '/bin/sh -c' on Linux[3]. In an effort to handle most RUN file cases, we implemented a shell parser to parse all the RUN commands within our Dockerfile data set[4]. We searched through all the RUN commands and counted each shell command occurrence. We decided to support the top 25 shell commands with the highest number of occurrences that may modify the file system.

5.4 Approach

5.4.1 Abstraction Domain

In our technique, we use automaton as the abstract domain. The automaton represents all possibly existing file paths in the file system at a program point. For example, for the following piece of code, the automaton at the end of the program is shown in Figure 5.1

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

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

5.4.2 Transfer Functions

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

Since the basic operations such as union are well defined for automaton abstract domain, we

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

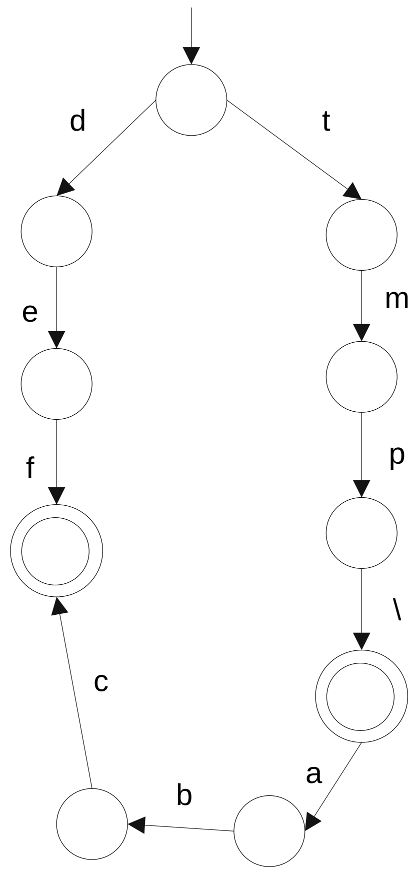

Figure 5.1: An Exemplar FSS Automaton

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

$$\begin{aligned} touch\ x : N &= N \cup automaton(x) \\ D &= D \cup parentdir(x) \end{aligned} \tag{5.1}$$

$$\begin{aligned} mkdir\ x : N &= N \cup automaton(x + \text{“/”}) \\ D &= D \cup parentdir(x) \end{aligned} \tag{5.2}$$

$$\begin{aligned} cp\ x\ y : N &= N \cup automaton(y + basename(x)) \\ D &= D \cup automaton(x) \end{aligned} \tag{5.3}$$

$$rm\ x : D = D \cup automaton(x) \tag{5.4}$$

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

We use function $basename(x)$ to denote the automaton generated by extracting the base file-name of a path variable x. Here we can see that because x is an automaton, the function $basename$ can only be implemented as a transducer, which is presented in Figure 5.2. In the finite state transducer, we use eps to denote ϵ, and texttu0001-uffff to denote the whole character set Σ, and $*$ as an output indicates that the output of the transducer at the specific transition will be the same as the input.

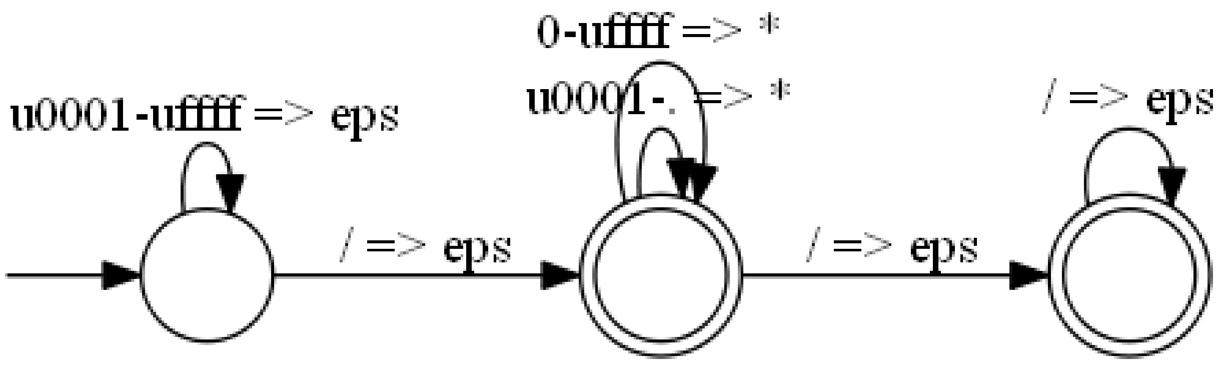

Figure 5.2: FST to extract the basename

We use function $parentdir(x)$ to denote the automaton generated by extracting the parent directory of a path variable x. For example, the parent directory of a regular expression `a/b/c|a/d` would be `a/b/|a/`. The transducer to extract parent directory from a path variable x is presented in Figure 5.3.

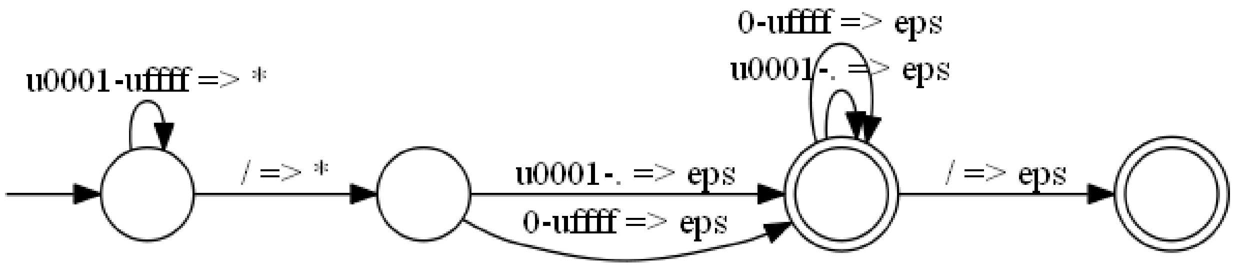

Figure 5.3: FST to extract the parent folder

The definition of paths in file manipulation scripts can be various. For example, `mkdir abc` and `mkdir abc/` both create a folder with name `abc`. This may cause our automaton to have extra file path separators, resulting in double file path separators or extra file path separator at the end of a path value. To remove such extra path separators, we design the following two finite state transducers as shown in Figures 5.4 and 5.5.

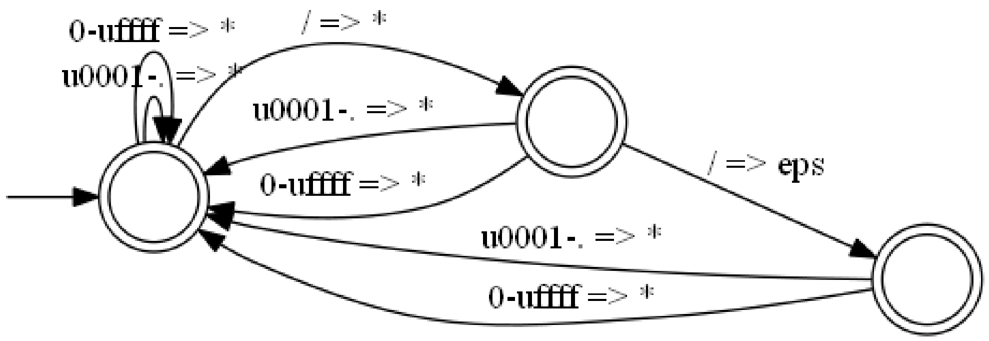

Figure 5.4: FST to remove double file path separators

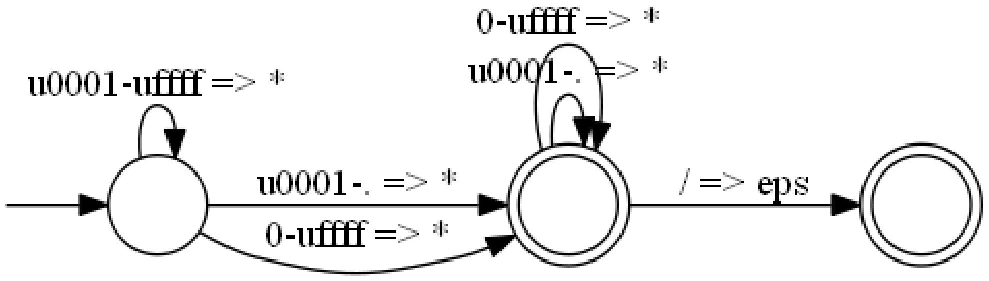

Figure 5.5: FST to remove the last file path separator

5.5 Evaluation

5.5.1 Evaluation Setup

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

[5]https://github.com/veggiemonk/awesome-docker

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

5.5.2 Evaluation Results

Our evaluation results are presented in Table 5.1. The Columns 2-9 present the execution time of Control Flow analysis, String analysis, FiFA round 1, FiFA round 2, total line number of the docker file, total line number of the converted FMIL file, the number of nodes in precondition, and the number of nodes in post condition, respectively. From the results we can see that on average FiFA's execution time is 80 milliseconds in total. Note that we run FiFA twice when applying it. In the first round, FiFA will report errors when a file is referred to but not generated by earlier statements. Since most file-manipulation scripts run in an existing file system and assumes the existence of certain paths, we consider these reported errors to be warnings and simply add the missing path to the initial FSS automaton. It should be noted that in this process, FiFA can infer the precondition required by the file-manipulation script to successfully run. In the second round, FiFA will use the inferred precondition as the initial FSS automaton so that FiFA will report "file not found" errors in the second round because all preconditions are satisfied.

Dockerfile	Live Ana.	Str. Ana.	FiFA Run 1	FiFA Run 2	LOC Dockerfile	LOC FFA	Pre Size	Post Size
appcelerator/amp	2	0	102	84	5	3	47	62
bcicen/ctop	0	1	263	366	13	7	46	109
caicloud/cyclone	0	0	8	13	18	1	2	2
Century/dray	0	0	8	10	5	1	6	6
cisco/elsy	0	1	69	120	16	2	69	76
cloud66/habitus	0	0	128	136	14	5	40	45
containous/traefik	0	0	33	20	16	2	15	15

[6]https://sites.google.com/site/fifarepo/

ContainX/netshare	0	0	86	89	7	3	3	52
cpuguy83/ambassador	0	0	85	112	5	2	84	100
crosbymichael/dockersql	0	0	39	32	5	2	43	43
CWSpear/persist	0	0	15	15	9	1	2	29
Dataman/crane	0	0	5	5	7	1	2	2
deltaskelta/alertd	0	1	15	9	30	2	14	14
dnephin/dobi	0	1	13	20	6	3	15	20
docker-flow/monitor	0	0	75	80	12	2	63	89
docker-slim/slim	0	0	18	19	8	2	16	16
docker/distribution	1	0	113	120	5	4	53	126
docker/libcompose	0	0	107	163	51	9	71	116
docker/machine	0	0	36	25	14	3	37	40
drone/drone	0	0	18	17	42	6	2	36
etcd-io/etcd	0	1	3	3	34	2	2	2
fabiolb/fabio	0	0	21	17	42	6	2	36
fsouza/dockerclient	0	0	31	58	14	5	40	45
genuinetools/bane	0	0	62	50	25	6	62	101
genuinetools/img	0	0	46	27	15	3	88	88
genuinetools/reg	0	0	48	45	26	7	60	97
goharbor/harbor	0	0	45	29	13	3	88	88
GoogleContainer/diff	0	0	22	21	6	2	2	41
GoogleContainer/test	0	0	14	21	7	1	37	57
google/cadvisor	0	0	16	16	42	6	2	36
harbur/captain	0	0	6	6	6	2	2	13
hashicorp/nomad	0	0	18	24	6	2	40	42
hasura/gitkube	0	1	22	19	42	6	2	36
howtowhale/dvm	0	0	51	54	13	3	88	88

iron-io/functions	0	0	4	3	4	1	2	2
istio/istio	0	0	26	45	16	1	45	45
ivanilves/lstags	0	0	25	14	17	2	26	26
jessfraz/dockfmt	0	0	158	146	25	6	64	105
jlhawn/dockramp	0	0	8	8	14	1	11	12
jwilder/docker-gen	0	0	7	5	8	1	2	6
jwilder/dockerize	0	0	23	24	12	2	30	30
kubernetes/kubernetes	0	0	14	14	43	3	2	36
mayflower/docker-ls	0	0	14	14	5	1	2	2
moncho/dry	0	0	12	16	16	2	21	21
mudler/companion	1	0	63	28	15	3	88	88
mutable/factory	1	0	30	51	14	5	40	45
openfaas/faas	0	0	20	19	42	6	2	36
oracle/smith	0	0	18	7	12	3	15	15
prologic/autodock	0	0	35	47	20	3	50	79
rancher/convoy	0	0	8	13	11	1	33	32
rancher/rancher	0	0	20	20	42	6	2	36
remind101/empire	0	0	16	16	5	2	38	38
rexray/rexray	0	0	15	15	11	4	2	54
stelligent/mu	0	0	3	3	6	1	2	2
theupdate/notary	0	0	10	12	6	2	26	27
tianon/gosu	0	0	22	26	49	4	35	56
tpbowden/router	0	0	10	12	5	1	27	26
weaveworks/weave	0	0	5	6	14	1	12	12
Average	**0**	**0**	**38**	**42**	**17**	**3**	**30**	**45**

Table 5.1: Execution Time and Sizes of Automatons (ms)

5.5.3 Qualitative Analysis

In this subsection, we present a simple example to show what our pre-conditions and post-conditions look like. The following dockerfile is from `dnephin/dobi` that we analyzed using our framework. It consists of six Dockerfile commands.

```
FROM     rails:5
WORKDIR /code
COPY     setup.sh /code/
RUN      ./setup.sh
WORKDIR /code/blog
CMD      ["bin/rails", "server", "-b", "0.0.0.0"]
```

Then, we were able to transform the Dockerfile commands to the following FMIL code shown below.

```
cd '/code';
touch '/code/setup.sh';
touch 'setup.sh';
cd '/code/blog';
```

Running this FMIL script in our framework gives us the following output.

```
dnephin_dobi_Dockerfile
***** Run: 0
***** Run: 1
touch: cannot touch 'setup.sh**': File already exists
Variable analysis elapsed time: 5ms
Grammar analysis elapsed time: 1ms
FFA first run elapsed time: 59ms
FFA second run elapsed time: 59ms
Process finished with exit code 0
```

The pre-condition and post-condition FSS automatons of the dockerfile generated by our FiFA analysis tool are presented in Figure 5.6 and Figure 5.7, respectively. It should be noted that, the original automatons generated by FiFA have only single character on each transition. For better readability, we combine multiple transitions on a chain to a single transition with a word on it. From the figures, we can see that FiFA successfully infers that the paths `/code/` and `/code/blog/` must exist before running the dockerfile. The post-condition also contains `/code/setup.sh` that is created by the `COPY` command at Line 3 of the original dockerfile.

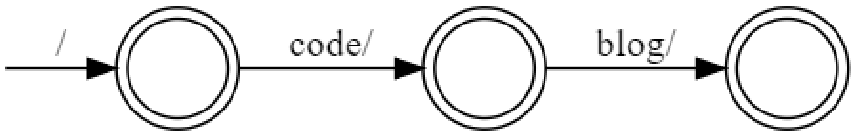

Figure 5.6: Pre-condition FSS Automaton

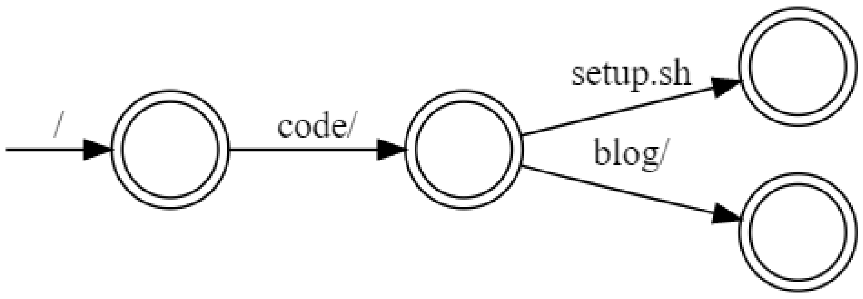

Figure 5.7: Post-condition FSS Automaton

Due to the large size of generated automaton, we cannot show the complete automaton FiFA generated for the example in Section 2 of this paper. But they can be found at our anonymized website[7].

5.5.4 Comparison With ShellCheck

In this subsection, we present the comparison of our technique with ShellCheck, an existing tool for detecting shell script warnings, errors and bad practices. Table 5.2 shows the errors detected by our technique and ShellCheck. We add a check mark to the column 3 and/or 4 if the error can be detected by corresponding tool. From the table, we can see that FFA and ShellCheck detect different types of errors in shell scripts so FFA can provide additional help to shell script developers on top of ShellCheck.

Script	Error	FFA	ShellCheck
aambrioso1_pypi-flask-app_server_setup.sh	WARNING: ’x’ does not exist	✓	
	cp: cannot stat ’x’: No such file or directory	✓	
	rm: cannot remove ’x’: No such file or directory	✓	
	cp: cannot stat ’x’: No such file or directory	✓	

[7]longtable

	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
	SC1091: Not following: 'x' was not specified as input		✓
aim-leo_db-auto-backup_add-schedule.sh	WARNING: 'x' does not exist	✓	
	SC2230: which is non-standard. Use builtin 'command -v' instead.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2006: Use $(...) notation instead of legacy backticked '...'.		✓
andcan_dust_build.yml.sh	WARNING: 'x' does not exist	✓	
	SC2148: Tips depend on target shell and yours is unknown. Add a shebang.		✓
	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
bismite_bismite-sdk_build_template.sh	cp: 'x' and 'x' are the same file	✓	
	mkdir: cannot create directory 'x': File exists	✓	
	cp: cannot stat 'x': No such file or directory	✓	
	WARNING: 'x' does not exist	✓	
	SC2086: Double quote to prevent globbing and word splitting.		✓

ColinPitrat_caprice32 _.travis.yml.sh	touch: cannot touch 'x': No such file or directory	✓	
	WARNING: 'x' does not exist	✓	
	SC2148: Tips depend on target shell and yours is unknown. Add a shebang.		✓
	SC2069: To redirect stdout+stderr, 2>&1 must be last...		✓
eclipse-embed-cdt_eclipse-plugins_builds-upload.sh	cp: 'x' and 'x' are the same file	✓	
	SC2012: Use find instead of ls to better handle non-alphanumeric filenames.		✓
hezrq_dotfiles_install.sh	WARNING: 'x' does not exist	✓	
	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
homeauto_nethunter_build.sh	mkdir: cannot create directory 'x': File exists	✓	
	rm: cannot remove 'x': No such file or directory	✓	
	cp: cannot stat 'x': No such file or directory	✓	
	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
	SC2034: 'x' appears unused. Verify use (or export if used externally).		✓

	SC2006: Use $(...) notation instead of legacy backticked '...'.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2004: / is unnecessary on arithmetic variables.		✓
	C2035: Use ./*glob* or – *glob* so names with dashes won't become options.		✓
ibm-garage-cloud_ibm-garage-tiles_release.yaml.sh	mkdir: cannot create directory 'x': File exists	✓	
	WARNING: 'x' does not exist	✓	
	SC2148: Tips depend on target shell and yours is unknown. Add a shebang.		✓
	SC2002: Useless cat. Consider 'cmd < file | ..' or 'cmd file | ..' instead.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
jhu-cs318_vagrant_bootstrap.sh	cp: cannot stat 'x': No such file or directory	✓	
	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
JonMarten_RNAseq_scrap_3_3a_map_cis_eQTLs_submissions_script_3phenotest.sh	WARNING: 'x' does not exist	✓	
	SC1091: Not following: 'x' was not specified as input		✓

jpentland_rcfiles_startup.sh	SC2009: Consider using pgrep instead of grepping ps output.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2002: Useless cat. Consider 'cmd < file | ..' or 'cmd file | ..' instead.		✓
	SC2068: Double quote array expansions to avoid re-splitting elements.		✓
lawrencewoodman_sblasm _.travis.yml.sh	WARNING: 'x' does not exist	✓	
	SC2148: Tips depend on target shell and yours is unknown. Add a shebang.		✓
layer6ai-labs_RecSys2020_run.sh	mkdir: cannot create directory 'x': File exists	✓	
	WARNING: 'x' does not exist	✓	
	SC2034: 'x' appears unused. Verify use (or export if used externally).		✓
linkedin_kafka-monitor_single-cluster-monitor.sh	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2068: Double quote array expansions to avoid re-splitting elements.		✓
MaxDesiatov_swiftwasm-build_build-toolchain.sh	mkdir: cannot create directory 'x': File exists	✓	
	SC1113: Use #!, not just #, for the shebang.		✓

	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2034: 'x' appears unused. Verify use (or export if used externally).		✓
	SC2230: which is non-standard. Use builtin 'command -v' instead.		✓
nwcd-samples_aws-instance-scheduler_buildspec.yml.sh	WARNING: 'x' does not exist	✓	
	SC2148: Tips depend on target shell and yours is unknown. Add a shebang.		✓
	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
	SC2006: Use $(...) notation instead of legacy backticked '...'.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2154: bucket is referenced but not assigned.		✓
pablochacin_operator-sh_demo.sh	WARNING: 'x' does not exist	✓	
	SC1091: Not following: ./screenplay.sh was not specified as input (see shellcheck -x).		✓
PokemonGoF_PokemonGo-Bot_setup.sh	WARNING: 'x' does not exist	✓	
	cp: cannot stat 'x': No such file or directory	✓	

	rm: cannot remove ’x’: No such file or directory	✓	
	mkdir: cannot create directory ’x’: File exists	✓	
	SC2164: Use ’cd ... || exit’ or ’cd ... || return’ in case cd fails.		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC1091: Not following: bin/activate was not specified as input (see shellcheck -x).		✓
	SC2162: read without -r will mangle backslashes.		✓
	SC2027: The surrounding quotes actually unquote this. Remove or escape them.		✓
seanbuckley_dotfiles_setup.sh	WARNING: ’x’ does not exist	✓	
	mkdir: cannot create directory ’x’: File exists	✓	
	rm: cannot remove ’x’: No such file or directory	✓	
	SC1091: Not following: ./utils.sh was not specified as input (see shellcheck -x).		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
spring-projects-experimental_spring-graalvm-native_build-samples.sh	WARNING: ’x’ does not exist	✓	

	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
Stivvo_wayPreview_compile.sh	SC2164: Use 'cd ... || exit' or 'cd ... || return' in case cd fails.		✓
The-BB_debian-keenetic_debian-buster-mips.sh	mkdir: cannot create directory 'x': File exists	✓	
	cp: cannot stat 'x': No such file or directory	✓	
	WARNING: 'x' does not exist	✓	
	rm: cannot remove 'x': No such file or directory	✓	
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2125: Brace expansions and globs are literal in assignments. Quote it or use an array.		✓
	SC2024: sudo doesn't affect redirects. Use ..| sudo tee file		✓
VicerExciser_watershedpi _launcher.sh	WARNING: 'x' does not exist	✓	
	SC1012: \t is just literal 't' here...		✓
	SC2003: expr is antiquated...		✓
	SC2004: $/${} is unnecessary on arithmetic variables.		✓
	SC2005: Useless echo? Instead of 'echo $(cmd)', just use 'cmd'.		✓

	SC2006: Use $(...) notation instead of legacy backticked ‘...‘.		✓
	SC2046: Quote this to prevent word splitting.		✓
	SC2059: Don’t use variables in the printf format string. Use printf "..%s.." "$foo".		✓
	SC2086: Double quote to prevent globbing and word splitting.		✓
	SC2126: Consider using grep -c instead of grep|wc -l.		✓
	SC2164: Use ’cd ... || exit’ or ’cd ... || return’ in case cd fails.		✓
	SC2196: egrep is non-standard and deprecated. Use grep -E instead.		✓

Table 5.2: Errors Detected by FFA and ShellCheck

Chapter 6: CONCLUSION

6.1 Network Traffic

In this paper, we propose a novel approach to statically generate traceable network summaries for android apps. Our approach is based on grammar templates of network API methods and string taint analysis, and we further propose new techniques to handle complex network API invocations, and to generate signatures from string-operation grammars. We evaluate our approach on top 500 android apps and 8 maintenance tasks from open-source android projects. The results show that our approach is able to efficiently generate signatures with high quality for most of the apps. The empirical study on maintenance tasks shows that the signatures generated by our approach are able to help developers precisely and quickly locate the code locations in network-related code maintenance tasks. There are several directions to further improve our work, which are listed as below.

First of all, we plan to apply our approach on a larger set of apps and real-world maintenance tasks. We also plan to adapt our approach to GUI-based Java software projects and evaluate our adapted approach on open-source Java applications.

Second, NetDroid currently generates some invalid signatures because it cannot trace into the system library. It is inefficient to analyze the whole android system, and we plan to build an android system model for the network signature generation problem, so that we can reduce the invalid signatures.

Third, our tool NetDroid is not able to process some apps due to the failure in loading some classes in the Java byte code. We may extend our tool to tolerate such failures or avoid such failures by enhancing the converting tool from Dalvik byte code to Java byte code.

6.2 Recommending Updates of Dockerfiles

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.

6.3 File Flow Analysis

In this paper, we present FiFA, a novel file flow analysis framework that infers pre-conditions and post-conditions of file-manipulation scripts. We developed FiFA on an intermediate language called FMIL, as well as a compiler to translate Dockerfiles to FMIL. Our evaluation on more than 60 Docker files shows that our analysis can efficiently perform the analysis averagely within 80 milliseconds.

In the future, we plan to extend our translator tool to handle shell environment variables, Dockerfile ENV commands, and conditional statements within RUN statements. We may also wish to add support for additional commands within the Docker RUN command. Furthermore, we plan to consider further analyze the permission of path variables to detect permission denial errors.

BIBLIOGRAPHY

[1] bashlint.

[2] Dex2jar, `http://developer.android.com/design/patterns/app-structure.html`.

[3] Findbugs.

[4] Lint.

[5] shellcheck.

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

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

[8] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. Fault localization for build code errors in makefiles. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 600–601, 2014.

[9] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. Fault localization for make-based build crashes. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 526–530. IEEE, 2014.

[10] Jafar Al-Kofahi, Tien N Nguyen, and Christian Kästner. Escaping autohell: a vision for automated analysis and migration of autotools build systems. In *Release Engineering (RELENG), 4th International Workshop on*, pages 12–15, 2016.

[11] J.M. Al-Kofahi, Hung Viet Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, and T.N. Nguyen. Detecting semantic changes in makefile build code. In *Proceedings of ICSM*, pages 150–159, 2012.

[12] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[13] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *SIGCOMM Comput. Commun. Rev.*, 36:23–26, April 2006.

[14] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[15] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 225–238, New York, NY, USA, 2008. ACM.

[16] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.

[17] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS*, pages 1–18, 2003.

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

[19] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[20] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on European Software Conference and Foundations of Software Engineering*, pages 393–403, 2015.

[21] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the docker container ecosystem on github. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 323–333. IEEE, 2017.

[22] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

[23] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[24] Shuaifu Dai, A. Tongaonkar, Xiaoyin Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*, pages 809–817, April 2013.

[25] John Downs, Beryl Plimmer, and John G. Hosking. Ambient awareness of build status in collocated software teams. In *Proceedings of ICSE*, pages 507–517, 2012.

[26] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[27] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[28] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[29] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. On lightweight mobile phone application certification. In *ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[30] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[31] Bolei Guo, Neil Vachharajani, and David I August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, 2007.

[32] Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. Acas: automated construction of application signatures. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, MineNet '05, pages 197–202, New York, NY, USA, 2005. ACM.

[33] William G. J. Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, page 174â183, 2005.

[34] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *ACM/IEEE Symposium on Empirical Software Engineering and Measurement*, pages 38–47, 2017.

[35] Foyzul Hassan and Xiaoyin Wang. Mining readme files to support automatic building of java projects in software repositories: Poster. In *International Conference on Software Engineering, Poster*, pages 277–279, 2017.

[36] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *International Conference on Software Engineering*, pages 1078–1089, 2018.

[37] Eric Horton and Chris Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *Proceedings of the 41st International Conference on Software Engineering*, page 328â338, 2019.

[38] Newso James, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Muhamad Fitra Kacamarga, Bens Pardamean, and Hari Wijaya. Lightweight virtualization in cloud computing for research. In Rolly Intan, Chi-Hung Chi, Henry N. Palit, and Leo W. Santoso, editors, *Intelligence in the Era of Big Data*, pages 439–445, 2015.

[40] John Kam and Jeffrey Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, January 1976.

[41] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proc. ICSE*, pages 199–209, 2009.

[42] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, page 105â116, 2009.

[43] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[44] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of USENIX Security*, 2005.

[45] S. McIntosh, B. Adams, T.H.D. Nguyen, Y. Kamei, and AE. Hassan. An empirical study of build maintenance effort. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 141–150, May 2011.

[46] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proc. WWW*, pages 432–441, 2005.

[47] Antoine Miné et al. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages*, 4(3-4):120–372, 2017.

[48] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In *International Static Analysis Symposium*, pages 361–382. Springer, 2016.

[49] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 50–60, New York, NY, USA, 2005. ACM.

[50] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: A study on behavioral backward incompatibilities of java software libraries. In *International Symposium on Software Testing and Analysis*, pages 215–225, 2017.

[51] Shaikh Mostafa and Xiaoyin Wang. An empirical study on the usage of mocking frameworks in software testing. In *Quality Software (QSIC), 2014 14th International Conference on*, 2014.

[52] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[53] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *International Conference on Software Engineering*, pages 77–88, 2015.

[54] Byung-Chul Park, Young J. Won, Myung-Sup Kim, and James W. Hong. Towards automated application signature generation for traffic identification. In *NOMS*, pages 160–167, 2008.

[55] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.

[56] Akond Rahman and Laurie Williams. Characterizing defective configuration scripts used for continuous deployment. In *International Conference on Software Testing*, pages 34–45, 2018.

[57] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[58] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *DIMVA*, pages 108–125, 2008.

[59] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *WWW2004*, May 2004.

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

[61] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 416â430, 2016.

[62] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 189–200. IEEE, 2016.

[63] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[64] A Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and T.N. Nguyen. Build code analysis with symbolic evaluation. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 650–660, June 2012.

[65] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering*, page 650â660, 2012.

[66] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Symake: A build code analysis and refactoring tool for makefiles. In *ACM/IEEE Conference on Automated Software Engineering*, pages 366–369, 2012.

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

[68] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 13–, 1999.

[69] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis. *IEEE Transactions on Software Engineering*, 39(4):516–536, 2013.

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

[71] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-translate constant strings in web applications. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 87–96. ACM, 2010.

[72] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the 2015 10th Joint Meeting on European Softwar Engineering Conference and Foundations of Software Engineering*, page 16, 2012.

[73] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

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

[75] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proc. ICSE*, pages 171–180, 2008.

[76] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proc. ISSTA*, pages 249–260, 2008.

[77] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *International Conference on Compiler Construction*, pages 1–17. Springer, 2000.

[78] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of ICSE*, pages 1–11, 2009.

[79] X. Xia, X. Zhou, D. Lo, and X. Zhao. An empirical study of bugs in software build systems. In *2013 13th International Conference on Quality Software*, pages 200–203, 2013.

[80] Kok-Kiong Yap, Te-Yuan Huang, Masayoshi Kobayashi, Yiannis Yiakoumis, Nick McKeown, Sachin Katti, and Guru Parulkar. Making use of all the networks around us: A case study in android. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, pages 19–24, 2012.

[81] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Form. Methods Syst. Des.*, 44(1):44–70, February 2014.

[82] Shurui Zhou, Jafar Al-Kofahi, Tien N Nguyen, Christian Kästner, and Sarah Nadi. Extracting configuration knowledge from build files with symbolic analysis. In *Release Engineering (RELENG), 3rd International Workshop on*, pages 20–23, 2015.

[83] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.

VITA

Rodney Rodriguez is from San Antonio, TX. He earned a Bachelorâs and Master's degree in Computer Science from The University of Texas at San Antonio. He plans to work as a research and development engineer with a focus in DevOps.