Universiteit Leiden
The Netherlands

Opleiding Informatica

Automated Dynamic Analysis
of Java Exploit Proof-of-Concepts

Jimmy Oei

Supervisors:
Soufian El Yadmani & Olga Gadyatskaya

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

25/08/2023

Abstract

Exploit Proof-of-Concepts (PoCs) play a crucial role in understanding attacks on system vulnerabilities and testing the security of systems. While PoCs are not supposed to be harmful, malicious entities can use them to easily distribute their malware. This is especially true on archive websites that do not analyze the published PoCs, such as GitHub. In the past research has primarily focused on analyzing conventional programs, which leaves a gap in our understanding of PoCs. In this research we have created an automated dynamic analysis system for Java PoCs to close this gap. This system will reduce the manual effort required when performing dynamic analysis, thus increasing the number of PoCs one can analyze in a given time period. We have created a tool, named Automated Black-box Executer (ABE), which automatically finds commands that build and execute the PoCs. The ABE was able to find 1595 commands in 57.33% of 1071 tested PoC repositories. Out of these, 549 executed a Java program, which covered 21.01% of the tested dataset. By leveraging this tool we were able to automate the dynamic analysis process of Java PoCs. In a test of the automated dynamic analysis system we were able to automatically get monitoring reports for six out of ten PoC repositories. With this research we have laid the groundwork for automating the dynamic analysis process and automated black-box execution.

Contents

1 Introduction

Cyber security has become paramount in today's modern society, where data stored online and technology-driven services have increased significantly. This digital advancement brings along unexpected, but inevitable vulnerabilities. These vulnerabilities are prone to attacks from malicious entities for nefarious purposes. Pentesters try to find unfixed vulnerabilities before attackers can exploit them. In this race against time, exploit proof-of-concepts (PoCs) play a vital role. A PoC is a proof that shows that an exploit concept is able to achieve its goal of compromising a system through a certain vulnerability. PoCs help security analysts and researchers to test system for certain vulnerabilities and to learn how specific attacks work. PoCs are a way to share knowledge on vulnerability attacks within the cyber security community. There are many archive websites that allow for distributing PoCs to the public, such as Exploit Database[1] and Metasploit[2]. The PoCs are, by convention, identified by the vulnerability that it attacks. The Common Vulnerabilities and Exposures (CVE)[3] system, maintained by MITRE[4], stores the vulnerabilities that are publicly known, with a CVE ID. A PoC is distinguished by the CVE ID it attacks. An example of a vulnerability is `CVE-2022-22965`[5], which can be exploited to gain remote code execution (RCE) in Spring MVC or Spring WebFlux applications running on JDK 9 or higher. A PoC for `CVE-2022-22965` would prove that this is possible by attacking it without doing any harm. A PoC, by its definition, is a non-harmful attack, which makes it very easy for malicious entities to label their attack as a PoC since the only real difference is whether harm is done or not. Archive websites for distributing PoCs generally analyze the PoCs before making them publicly available. While many PoCs are available on such websites, not all of the PoCs are. This holds specifically for newer PoCs, since the internal analysis of the uploaded PoCs can be extremely time consuming. This is why security analysts may find themselves looking for PoCs on other websites, such as GitHub[6]. GitHub is a popular code repository website for collaboration and version control, but also for making code publicly available. Distributing PoCs through GitHub is much easier, because GitHub does not perform any comprehensive analysis, although they are strictly against it in their Acceptable Use Policies[7]. GitHub mainly relies on their reporting system to remove harmful code, which is much less efficient. This is why PoCs from GitHub pose a higher risk of being malicious.

In the past research has primarily focused on analyzing conventional programs, leaving a gap in our understanding of PoCs and especially those on GitHub. The lack of insights on GitHub PoCs is concerning as these are potentially more dangerous given that they are distributed without any examination. One study, done by El Yadmani, The, and Gadyatskaya, has done static analysis of PoCs from GitHub [YTG23], which consists of performing analysis on the code. However, the behaviour of the PoCs under execution remains uninvestigated. Furthermore, the PoCs outside the analyzed dataset remain untouched. It is clear that more insights on GitHub PoCs are needed. The purpose of this study is to fill this gap by introducing an automated dynamic analysis system for

[1] https://www.exploit-db.com/
[2] https://www.metasploit.com/
[3] https://www.cve.org/
[4] https://www.mitre.org/
[5] https://nvd.nist.gov/vuln/detail/CVE-2022-22965
[6] https://GitHub.com/
[7] https://docs.GitHub.com/en/site-policy/acceptable-use-policies

Java[8] PoCs. This system eliminates the time-consuming need to manually perform the dynamic analysis by automating this process. The only thing the security analyst has to do is evaluate the monitoring results the dynamic analysis system produces. To build the system we have investigated a dataset of 1071 Java PoCs and related work on automated black-box testing and automated building using readme mining. It became evident that creating a new tool for automatically building the PoCs and finding commands that run it was required. We called this automated black-box execution and used the concept of readme mining to enhance the search for finding valid commands by looking in the documentation files written by the developers of the PoCs, given the promising results for commands in readme files that we found in our data analysis of the dataset. We have tested the performance of this new tool, which we named Automated Black-box Executer (ABE), in an experiment where we ran the ABE on the dataset we analyzed. We found that the ABE was able to find 1595 commands for 57.33% of the 1071 PoCs in the dataset. Out of these, 549 were Java commands, which covered 21.01% of the dataset. These are promising results and show that the ABE can automatically find commands such that dynamic analysis can be automated. We also tested the automated dynamic analysis system on ten random PoC repositories from the dataset and found useful monitoring reports for six PoC repositories. With this research we show that the dynamic analysis process can be automated by using automated black-box execution to find commands. To the best of our knowledge we are the first to research automated dynamic analysis of Java PoCs and automated black-box execution.

[8]Java is a programming language and computing platform first released by Sun Microsystems in 1995. (https://www.java.com/en/)

2 Background

In this section we will provide information on subjects discussed in this thesis. Subjects that are relevant and require more explanation are explained in detail here. Specifically, we will discuss isolated environments, automated black-box execution, Java command-line arguments, and Markdown code blocks.

2.1 Isolated Environment

Dynamic analysis of the PoCs will be performed to investigate the behaviour of the PoCs and uncover malicious ones. Dynamic analysis involves analyzing the properties of a running program, unlike static analysis which is performing analysis without any execution, i.e. looking at the code [Bal99]. Executing the PoCs is dangerous, which is why it should be done securely in an isolated environment to prevent any harm to production systems, external computers or personal computers. Such an isolated environment used is often called a sandbox. Sandboxes are used by software developers to test their code and by security analysts to perform malware analysis. Sandboxes offer a jail-like environment to execute programs in without them having any direct access to the resources of the host machine [JNR+18]. The host machine is the machine in which the sandbox will run [VGT14]. The sandbox is essentially a virtual machine that runs on the host machine, and simulates a live system. The sandbox will only use part of the resources of the host machine, and this is done in complete isolation. From the perspective of the sandbox it thinks and acts like a real machine with its own resources, processes, and operating system. The sandbox provides protection against potential harm or damage to the host machine and other machines when running malicious or faulty software. Dynamic analysis can be performed by using monitoring tools in the sandbox to analyze the behaviour of the program under analysis.

2.2 Automated Black-box Execution

With automated black-box execution we refer to the concept of automatically executing a program without knowing the internal workings of the program. In this research we will need to automatically execute the Java PoCs in the isolated environment to automate the dynamic analysis process. The internal workings of the PoCs are unknown to us, i.e. we are dealing with black-boxes, thus to automatically execute the PoCs we have automatically find the inputs that lead to the intended outputs of the PoCs. The process of finding these inputs is what we call automated black-box execution. In this thesis we have created a form of doing this process, which is, to the best of our knowledge, the first form of automated black-box execution. In this research we will simply call the program that performs automated black-box execution the Automated Black-box Executer (ABE).

To our knowledge, there is very limited research and information available on the specific topic of automatically executing a program. A few potential reasons may be that there is a lack of practical application, the automation trade-offs are not worth exploring, and automating programs that are created for manual interactions serve no purpose. However, there does exist a lot of research on the topic of testing, specifically similar to automated black-box execution: automated black-box testing. Automated black-box testing is automatically testing a program without knowing the internal workings, with the purpose of finding defects and errors. Although the goals of automated

black-box execution and automated black-box testing are different, they have several similarities. For instance, both concepts are trying to automatically achieve extensive code coverage. However, when looking at their objectives they can be seen as complete opposites: automated black-box execution is trying to run the program and automated black-box testing is trying to crash it. Nevertheless, we believe that the techniques that exist for automated black-box testing can be flipped and used for automated black-box execution. The form of technique is like a coin, and automated black-box testing is only one side of it. Unfortunately, the other side of the coin has yet to be well-establishment in the research community. We will therefore use automated black-box testing techniques as foundation building-blocks for our Automated Black-box Executer. In section 3.3 we will go over the literature on automated black-box testing techniques and evaluate the best forms we can adapt for automated black-box execution.

In this research we will also include the automation building the PoCs in the ABE, since Java projects from GitHub must generally be built before it can be run. Without an executable to run, the automated black-box execution cannot even begin, which is why we have to include the building in the ABE. In contrast to automated black-box execution, there has been some research in automatically building projects, which we will discuss in section 3.1. The process of automatically building for Java projects is easier as the domain of inputs are limited by the building software used, since the building software has generally a finite number of options. Furthermore, there exist default build commands, which are the commands that are conventionally used for building a project with the respective build software. In the context of this research, the main focus is on automatically executing the PoCs for dynamic analysis. Building the PoCs is a step towards this goal, therefore we will see this as a phase of the automated black-box execution.

2.3 Java Command-line Arguments

Command-line arguments are parameters passed to a program when it is executed, which is often through the command-line or terminal. They provide a way to customize a program without changing the internal code, hence allowing more dynamic programs. Command-line arguments are a form of input to a program. Several other important forms of inputs are `stdin`[9], files, and graphical user interfaces (GUIs). A distinction can be made between the inputs during execution and the inputs used to trigger execution. Command-line arguments are of the latter category. They are used to initiate the execution of the program, hence crucial for correct program execution, since an incorrect input of this category may lead to a failed execution attempt. In this research the focus will be on the form of input of command-line arguments, such that the executions of the Java PoCs are triggered. Without an execution there is no dynamic analysis possible, therefore it is essential to find the intended command-line arguments of the PoCs.

Command-line arguments in Java are passed through the array parameter in the main method of the main class of a project[10]. In listing 1 an example Java program that uses command-line arguments is shown using the conventionally used names of the array parameter (`args`), main

[9] `stdin` (Standard Input) is the standard input stream of a program used to interact with the program during execution. (https://man7.org/linux/man-pages/man3/stdio.3.html)

[10] For information on command-line arguments see https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html.

method (`main`), and main class (`Main`). This program, if given the correct command-line arguments, will print "Hello World", then the first command-line argument, and then the second command-line argument. The command-line arguments will be passed to the array parameter in the main method, in listing 1 called `args`. Distinction between multiple command-line arguments is made by the space between them. The type of this `args` array must be a string array, otherwise the Java program will crash. When an argument is supposed to be used but missing in the array, the program will output an array index out of bounds exception unless the exception is captured. For instance, if we only pass one command-line argument to the example program in listing 1, the program will print "Hello World", then the first command-line argument, and then it will crash with a `returncode` of 1 and the "array index out of bounds" exception (`java.lang.ArrayIndexOutOfBoundsException`) in `stderr`, because it tried to get the second element in the `args` array (index 1), but the array only contains one element. This information is fundamental for our implementation of automated black-box execution of Java PoCs.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

Listing 1: Example of a Java program that uses command line arguments.

2.4 Markdown Code Blocks

In Markdown[11] a code block is a way to represent code or anything related to programming. It is a different way of formatting text in Markdown, such that raw text and code related content can be easily distinguished from each other. Code blocks are commonly used for giving code examples or instructions, such as command-line snippets on how to run or install a program. Given that the purpose of a code block is to highlight code related content it is of high interest to us for automated black-box execution, because the chances are higher that useful instructions on running and building the PoC is given in a code block. In Markdown there are different ways to create a code block. A basic code block can be created by indenting with four spaces or one tab after an empty line. An example of a basic code block can be seen in listing 2, where line three is empty and line four is indented with a tab (or four spaces), hence the command-line snippet "apt install project" will be formatted as a code block in Markdown. The resulting formatted text can be seen in figure 1, where indeed the command-line snippet is in a code block. This formatted text and the one seen in figure 2 is obtained using the *Live (GitHub-Flavored) Markdown Editor*[12] developed by James Taylor released under the ISC License[13]. To create code blocks with this syntax that take up multiple lines every next line should also be indented with four spaces or a tab.

[11] See Markdown Guide for documentation on Markdown. (https://www.Markdownguide.org/)
[12] https://jbt.GitHub.io/Markdown-editor/
[13] https://GitHub.com/jbt/Markdown-editor

```
# This is a title
If you want to install this project, do the following command:

    apt install project

That is all.
```

Listing 2: Example of a basic code block in Markdown.

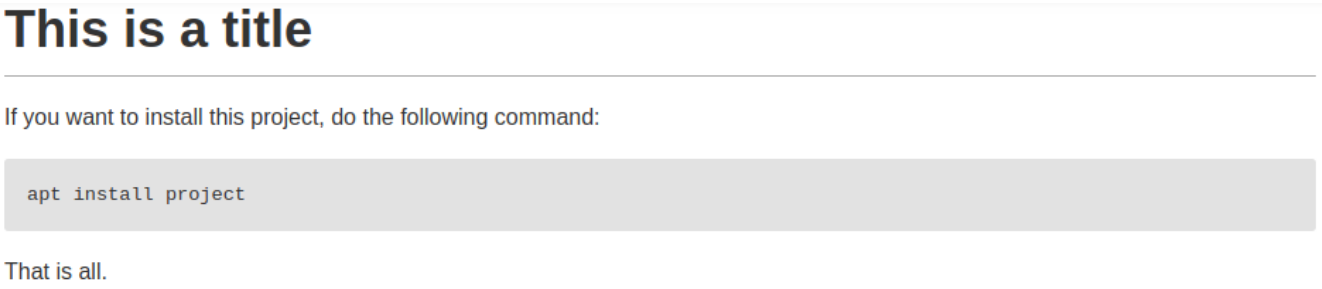

Figure 1: The Markdown formatted text of the example of a basic code block presented in listing 2.

An other way to create a code block is by fencing it with three backticks (```) or three tildes (~~~). These code blocks are called fenced code blocks and a way to make code blocks consisting of multiple lines without having to indent every line. Fenced code blocks allow you to also add color highlighting for a specific programming language. This is done by adding the name of the programming language after the first three backticks or tildes. An example of fenced code blocks in Markdown is given in listing 3, where the first code block is created using backticks and has the syntax highlighting for Bash[14], and the second code block is created using tildes. The resulting formatted text can be seen in figure 2. As you can see, the word "for" in the first code block is in bold because of the highlighting for Bash.

At last, it is possible to create a code snippet within a text in Markdown. The difference between a code snippet and a code block here is that a code block takes up the whole line, while a code snippet is a block only around the text within the snippet. This allows to have the same format for a code block, which is generally a box with a gray background as seen in the previous examples, but without it formatting the whole line. A code snippet is created by encompassing a text with single backticks (`). The backticks and the texts must not take up multiple lines. Additionally, it is also possible to use the three backticks for a fenced code block to create a code snippet if it is in a single line, however this does not work for three tildes. The example given in listing 4 shows these three possible ways to create a code snippet. The corresponding formatted output can be seen in figure 3.

[14]Bash is an sh-compatible shell developed by GNU. It is commonly used to run shell (`.sh`) scripts. (https://www.gnu.org/software/bash/)

````
# This is a title
If you want to install this project, do the following command:
```bash
This is a code block of multiple lines with highlighting for Bash.
The instruction is:
apt install project
```

If you want to run this project, do the following:
~~~
Code block with default highlighting.
cd /home/user/projects
run project
~~~

That is all.
````

Listing 3: Example of fenced code blocks in Markdown.

This is a title

If you want to install this project, do the following command:

```
This is a code block of multiple lines with highlighting for Bash.
The instruction is:
apt install project
```

If you want to run this project, do the following:

```
Code block with default highlighting.
cd /home/user/projects
run project
```

That is all.

Figure 2: The Markdown formatted text of the example of fenced code blocks presented in listing 3.

This is a title

If you want to install this project, do the following:

- First do `apt install project`.
- Then to run it you have to do `run project`

That is all.

Figure 3: The Markdown formatted text of the example of code snippets presented in listing 4.

```
# This is a title
If you want to install this project, do the following:
- First do ‘apt install project‘.
- Then to run it you have to do ‘‘‘run project‘‘‘

That is all.
```

Listing 4: Example of code snippets in Markdown.

3 Related Work

In this section we will explore previous research related to our objective of automating the dynamic analysis process for Java PoCs. These related works have shaped our approach of creating an automated dynamic analysis system. We will examine their contributions and novel ideas. Specifically, we will delve into the literature on automated Java project building, readme mining, automated black-box testing, and dynamic analysis of malicious PoCs.

3.1 Automated Java Project Building

Automating the building phase of the Java PoCs is required for the automated black-box execution in this research. To automate the Java project building, we build upon and integrate insights from various previous studies, which will be discussed in this section.

Sulír, et al. analyzed the build logs of a large number of Java projects from GitHub that used Maven, Gradle, and Ant [SBM+20]. They ran 7233 Java projects, spanning from 2008 to 2020, with their respective default build commands in a Docker container that simulated a standard programmer's environment. The findings showed that 59% of the builds failed, which is a concerning amount as this would potentially limit the coverage of our automated black-box execution. Furthermore, Sulír, et al., divided the failed executions into error categories. This revealed that most of the build failures were due to an error in Java compilation (36.23%) or issues with dependencies (26.94%). Unfortunately, these errors are difficult to fix automatically, since they require complex debugging and cover a wide range of unique issues. Additionally, Sulír, et al., compared the findings with a previous similar study they conducted in 2016, in which they found 38% of 7264 Java projects, spanning from 2008 to 2016, failed their respective default build command [SP16]. They also found that larger, older, and less recently updated projects failed more frequently. Interestingly, in the study conducted in 2016 they briefly discussed using the instructions given in readme files to build the Java projects, which they did manually for three random failing projects and managed to build them successfully after manually fixing the problems. However, they argue that automated builds are not fully automated if they require fixes to be made manually. This is where our research, and other research [HMLW17], proposes the use of readme mining to achieve full automation. In section 3.2, we will delve deeper into the concept of readme mining and its potential.

Another relevant study was conducted by Hassan et al., focusing on the feasibility of automatic software building [HMLW17]. They first executed the respective default build commands on top 200 Java projects from GitHub and discovered 91 build failures in 86 (43%) of the Java projects. Next, they manually analyzed these 86 projects that failed to determine the root causes of these failures

and how to fix them. They show that 52 (57%) of the build failures can be resolved automatically. What is more, 33 of the build failures were caused by a non-default build command of which 10 had the correct build command mentioned in the readme file in their project. After performing extraction of the commands from the readme files they were able to successfully build 5 of the 10 projects. This again emphasizes the potential of extracting build instructions from readme files to automate the building process. However, the relatively high proportion of build failures remains a concern.

These studies show that build failures will limit the coverage of the automated black-box execution and therefore the number of PoCs that will be analyzed. To improve the efficacy of automatically building the PoCs we will leverage the concept of readme mining with the aim of improving the overall coverage of the automated black-box execution process.

3.2 Readme Mining

Readme mining involves extracting certain information from the readme files in projects, which are often used to give information and instructions about the project to the user. A readme file is a Markdown file with a typical name of "*README.md*". Given that the PoCs are a black-box for us, we can leverage the potential information and instructions in the readme files to streamline the automated black-box execution of the PoCs.

Both the studies on automated Java project building discussed in section 3.1, have indicated that readme files can contain valuable information to improve the building process. Prana, et al. researched this more extensively [PTT+18]. They conducted a qualitative study where they categorized 4,226 file section from 393 readme files from GitHub through manual annotation [PTT+18]. They identified eight different kinds of content, and found that 88.5% of the readme files contained *'How'* content, which is the category they used for instructions and technical information about the projects. This is the content that will be particularly useful for the facilitating the automation of the building and execution of the projects. Additionally, the study designed a classifier to automatically predict the categories of sections in the readme files, which achieved an F1 score of 0.746.

We already saw in the study done by Hassan, et al. when automating Java project building that readme files could be used to increase the number of successful builds [HMLW17]. Hassan and Wang built upon their findings in an other study in which they proposed the first technique to automatically extract build commands from readme files and Wiki pages to support automatic building of Java projects [HW17]. Their technique leverages the Named Entity Recognition (NER) technique for extracting build commands. NET is a Natural Language Processing (NLP) technique that is used to identify and classify named entities within a text [BRN+09]. With their technique, they mined the readme files of top 1,500 Java projects on GitHub and managed to find 857 projects (57%) containing build commands in their readme files.

Apart from automatic project building, the concept of extracting valuable content from readme files has been utilized for various other purposes. For instance, Greene, et al. extracted potential skills of developer candidates from the readme files of their open-source contributions to identify suitable candidates for specific development roles [GF16]. They build on the notion that readme files

contain information about the technologies used in a project, because it contains information about installation instructions and project's dependencies. Similarly, Carvalho, et al. created DMOSS which mines readme files and other non-source code to assess the quality of non-source code text found in software packages [CSA14].

In our research, we will not confine the mining process to readme files alone, as other Markdown files may also contain valuable information and instructions about the projects. By leveraging readme mining and exploring additional Markdown files, we aim to increase the coverage of the automated black-box execution process.

3.3 Automated Black-box Testing

As discussed in section 2.2 automated black-box execution has a lot of similarities with the concept of automated black-box testing. Given these similarities we have analyzed literature on automated black-box testing techniques and system to explore techniques and systems we may potentially use to perform automated black-box execution of Java PoCs.

Mariani, et al. have surveyed the recent advances in automatic black-box testing in 2015, where they cover contributions from 2010 to 2014 [MPZ15]. In their paper they provided a comprehensive overview of four main areas with recent advances within automatic black-box testing: random testing (RT), models-based testing (MBT), testing with complex inputs, combinatorial interaction testing (CIT). Most interesting for automated black-box execution in this research is RT. "Random testing (RT) is a black-box software testing technique, where the core idea is to test a program by executing a set of randomly generated inputs." [MPZ15]. RT has commonly been used as a comparison baseline, however it has lately been increasingly seen as a good alternative testing technique [AIB12]. Since we do not know the required inputs of the PoCs or any information about it, trying random inputs to find a valid input seems like a good option. The paper discussed four different advances in RT: pure RT, adaptive RT, guided RT, and fuzz testing. Of these RT techniques guided RT and fuzz testing (also called fuzzing) look most promising for automatically finding valid inputs. Guided RT is a form of RT that uses information about the software under test (SUT) to guide the testing process. In a similar way one may guide the search for valid inputs of a program using known information of the program, in this case we know that the program we will run are Java PoC exploits. We thus can make better predictions on what the arguments of the SUT could be. Furthermore, we can use the output of the SUT to guide the process, such as the `returncode`[15], `stderr`[16], and `stdout`[17]. The `returncode` directly communicates feedback on the behaviour of the program, while `stderr` can provide causes of unintended behaviour, such as errors and exceptions. Additionally, `stdout` can be used by the developer of the program to give additional feedback and instructions. Fuzz testing is a technique of RT that tries to find defects or vulnerabilities by providing random unexpected or semi-valid inputs to the SUT, based on

[15] `returncode` is the exit status code returned by a command or a process. It indicates how the process or command returned, where a `returncode` of 0 means that the execution was successful and without any errors.

[16] `stderr` (Standard Error) is the standard error stream of a program used to output error messages. (https://man7.org/linux/man-pages/man3/stdio.3.html)

[17] `stdout` (Standard Output) is the standard output stream of a program used to output regular program output. (https://man7.org/linux/man-pages/man3/stdio.3.html)

collecting valid inputs and mutating them to obtain novel inputs [MPZ15][ZWCX22]. Similar to fuzz testing we can collect valid inputs and mutate them to steer the process towards its goal. With the guidance from the outputs of the PoCs we can see which arguments were valid. By the principles of fuzz testing we can then keep these arguments and build on this input combination by mutating it, such as adding a new random argument to potentially get more coverage.

One interesting system we encountered that uses guided RT is DART (Directed Automated Random Testing) by Godefroid et al. [GKS05], designed for automatic testing. DART combines three main techniques to automate unit testing: automated extraction of the interface of a program, automatic generation of a test driver for this interface, and dynamic analysis of how the program behaves under random testing. These techniques allows testing to be performed completely automatically, as there is no need to write any test driver. DART generates random test drivers and guides the testing along alternative paths using dynamic analysis of the program under testing. Although DART employs similar concepts, its primary focus is on the C programming language and unit testing, making it less suitable for our purpose of automated black-box execution on complete programs.

An other guided random testing technique that does primarily focus on Java is Randoop[18]: "Randoop is a well-known and successful test case generation technique that dynamically adapts the test case generation process according to the results produced by the previously executed test cases" [MPZ15] [Bal05]. Randoop dynamically adapts the test case generation process based on the results produced by previously executed test cases. While this guidance mechanism seems applicable to automate program execution, Randoop primarily targets unit testing and may not be directly adaptable for automated black-box execution.

Other interesting form of techniques that have been brought to attention in 2013 to seek vulnerability detection, exploitation, and patching in near real-time, are symbolic execution techniques [BKM14] [BCD+18]. By executing a program symbolically, using symbolic variables instead of concrete variables, all the possible execution paths can be explored [BCD+18]. During symbolic execution, constraints are collected and afterwards put in a SMT (Satisfiability Modulo Theories) solver to verify whether there are any instances that cause property violations. Although symbolic execution appears promising, path explosion is of high concern when performing symbolic execution on a complete program instead of a unit/function. Furthermore, symbolic execution requires that the internal workings of the program must be known, which makes it unsuitable for automated black-box execution.

We also considered existing Java fuzzers that could potentially be adapted for automated execution. Leveraging an existing fuzzer would provide state-of-the-art fuzzing techniques, but the necessary changes to make it suitable for automated black-box execution prove to be substantial. The requirements that should be implemented to make it work for automated black-box execution are: **1.** fuzz complete black-box programs; **2.** find valid executions instead of invalid ones. Unfortunately, we did not find a suitable Java fuzzer that could easily be changed to meet the criteria. Many popular fuzzers, like JQF [PLS19] and AFL [Zal16], are optimized for gray-box testing. Gray-box testing often uses instrumentation to track code coverage, which is inserting additional code into

[18]https://randoop.GitHub.io/randoop/

the program under test to gather information about the execution by tracking code coverage. For instance, JQF, developed by Padhye, et al., uses instrumentation on the JVM bytecode-level[19] to track code coverage and guide the fuzzing process to different code paths [PLS19]. Additionally, AFL (American Fuzzy Lop) and AFL++ (the successor of AFL) work by automatically discovering test cases that lead to new code paths using compile-time instrumentation and genetic algorithms [Zal16] [FMEH20]. Although, AFL focusses on testing C programs, we have looked at Kelinci, which allows AFL-style fuzzing to be applied on Java programs [KLP17]. Kelinci provides an interface to execute AFL on Java programs, by communicating the AFL-style instrumentation of Java programs to a simple C program that interfaces with the AFL fuzzer. Unfortunately, given the complexities we have found when instrumenting black-box programs make these popular fuzzers unsuitable for automated black-box execution.

At last we have looked at some less popular, but black-box, fuzzers. However, we have found that these black-box fuzzers are generally for testing web applications. This is probably because interfaces over HTTP or HTTPS tend to be more like a black-box from the perspective of a fuzzer, which makes it a better use case.

Although black-box testing and fuzzing is relevant to automated black-box execution, they are not the same and have very different use cases. Existing testing tools typically operate on the unit level, whereas automated execution involves the entire program. Furthermore, testing aims to discover vulnerabilities and bugs, whereas the goal of executing is to run the program correctly. Because of these differences altering existing testing tools for automated black-box execution becomes very complex. It is therefore evident that creating a new tool designed specifically for automated black-box execution may be more efficient and effective. We have seen that the techniques of guided random testing and fuzzing look promising for automatically finding valid inputs of a program when adapted. We can use these techniques as foundations for building the Automated Black-box Executer. To the best of our knowledge we are the first to research the automated execution of black-box programs.

3.4 Malicious Exploit Proof-of-Concepts Analysis

El Yadmani, The, and Gadyatskaya have done static analysis on malicious PoCs on GitHub [YTG23]. They collected 47,285 GitHub repositories that contained PoCs for CVE's discovered in 2017 to 2021, which they performed static analysis on to discover malicious PoCs. In their static analysis they looked at the following indicators: IP addresses, binaries, and IPS inside base64 or hexadecimal obfuscated payloads. The found IP addresses were then checked in VirusTotal[20] and AbuseIPDB[21], and the binaries were checked in VirusTotal. We will build upon the findings of the static analysis research by performing dynamic analysis of malicious PoCs on GitHub. Although there has been research on automatically generating proof-of-concepts exploits for vulnerabilities

[19] JVM bytecode is the low-level representation of Java source code which can be run using the Java Virtual Machine (JVM).

[20] VirusTotal is an online tool to analyse suspicious files, domains, IPs, and URLS to detect malware. https://www.virustotal.com/

[21] AbuseIPDB is a community-driven online tool for finding and reporting malicious IP addresses. https://www.abuseipdb.com/

[YZC+17] [BPSZ08], there has not been any research on dynamic analysis of PoCs to the best of our knowledge.

4 Dataset and Data Analysis

To create the automatic dynamic analysis system we will first analyze a dataset of Java PoCs on GitHub to get a better understanding of the PoCs on GitHub and extract potential requirements for the automated black-box execution and dynamic analysis process. This dataset will be analyzed and used to build the dynamic analysis system, specifically the component for the automated black-box execution. After building the dynamic analysis system we will perform an experiment to evaluate its performance and limitations, which will also be done with this dataset.

4.1 Dataset

The dataset we will use is obtained in a previous study done by El Yadmani, The, and Gadyatskaya that performed static analysis of PoCs from GitHub [YTG23]. The previous research collected data from 47,285 repositories on GitHub with PoCs for known vulnerabilities discovered in the years 2017 to 2021, which contained in total 72,580 PoCs written in various programming languages. The study distinguished 1071 Java PoC repositories, using the GitHub language labels, from the total dataset, which we will use as our first dataset. This dataset contains repositories created in the years 2017 to 2022. Furthermore, the PoCs in the dataset target in total 1191 CVE's, of which 170 are unique. The previous research found only one malicious PoC in the Java dataset with their static analysis. This repository targets the CVE's `CVE-2021-44228`[22], `CVE-2021-45046`[23], and `CVE-2021-45105`[24], which are Apache Log4j[25] vulnerabilities. The static analysis found that this PoC repository contained a malicious PoC, because it has a binary (a `.exe` file) that is flagged as malicious by VirusTotal[26]. This is the only binary the previous research found in the Java dataset. The lack of binaries along with the higher complexity of Java projects, makes it more difficult to find malicious PoCs in the Java dataset using static analysis. This is highlighted by the findings of the static analysis of merely 0.09% repositories of the Java dataset being malicious, which is much less than the total findings of 1.90% malicious repositories in the static analysis research. This highlights the necessity for further exploration of Java PoCs by performing dynamic analysis.

4.2 Data Analysis

To facilitate the automated black-box execution and dynamic analysis process, it is crucial to understand the composition of the data in the Java PoC repositories and identify common patterns. This will give us insights on the requirements for our automated dynamic analysis system, particularly the component that will perform the automated black-box execution, and hopefully future research. We have analyzed the PoC repositories in the dataset on several aspects, which we differentiate as

[22] https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[23] https://nvd.nist.gov/vuln/detail/CVE-2021-45046
[24] https://nvd.nist.gov/vuln/detail/CVE-2021-45105
[25] https://logging.apache.org/log4j/2.x/
[26] https://www.virustotal.com/gui/file/10b041dcb1c32b783122687723e02b6708852e8fec919fd4122cfea4c94da482/community

follows: file extensions, build files, and keywords analysis. This data analysis is performed by simply checking every file and directory of the PoC repositories and counting their occurrences. In this data analysis we have excluded several files and directories that we consider unimportant to our research and thus unwanted. For instance, we did not continue the traversal of the file structure when the directory or file started with a dot, e.g. the *.git* directory. This exclusion allows us to focus on the important characteristics of the PoC projects, thereby generating more meaningful results for the dynamic analysis process. The results were exported to a comma-separated values (CSV) file (`.csv`)[27], which we then used to get certain statistics and data shapes.

4.2.1 File Extensions

In the file extensions analysis we analyzed the occurrences of certain file extensions in the PoC repositories, along with their respective depths, which represents the number of directories one must traverse from the root of the repository directory to locate the file. The analysis concerned the following file extensions: `.java` (Java files), `.md` (Markdown files)[28], `.jar` (JAR files)[29], `.txt` (Text files)[30], `.py` (Python files)[31], and `.sh` (Shell files)[32]. The file extension `.exe`[33] was already analyzed in the static analysis study [YTG23] (see section 4.1), where only one file with the `.exe` file extension was found. We have therefore not included `.exe` in this analysis and also not implemented functionality for it in the dynamic analysis system, because of the low presence in the dataset. The findings of the file extensions analysis can be found in table 1, figure 4, and figure 5.

	Number of repositories	Percentage of dataset
`.java`	1047	97.76%
`.md`	969	90.48%
`.jar`	468	43.70%
`.txt`	316	29.51%
`.py`	221	20.63%
`.sh`	130	12.14%

Table 1: Number and percentage of repositories of the dataset that contain at least one file with the respective file extension.

Table 1 shows the number and percentage of repositories of the dataset that contain at least one file with the respective file extensions, i.e. the file extension is present in the repository. We see that the file extensions `.java` is present in 97.67% of the repositories in the dataset, confirming that these are indeed Java projects. After manual inspecting we found that the 24 repositories that

[27] A CSV (Comma-Separated Values) file is a delimited text file that allows data to be saved in a table structured format. (https://docs.python.org/3/library/csv.html?highlight=csv)

[28] Markdown is a lightweight markup language for creating formatted text, developed by John Gruber. (https://daringfireball.net/projects/Markdown/)

[29] Java archive (JAR) file is a package file format used for aggregating many files into one, which enables easy execution of Java projects. (https://www.java.com/en/)

[30] `.txt` files are files consisting of plain text, without any special formatting.

[31] Python is a high-level programming and scripting language. (https://www.python.org/)

[32] `.sh` files are Unix shell executable scripts.

[33] A file with file extension `.exe` is an executable for Microsoft Windows.

did not contain a file with a `.java` extension had at least a JAR file (`.jar`) or a JSP file (`.jsp`)[34]. Interestingly for our dynamic analysis, we have found that Markdown files (`.md`) are present in 90.48% of the repositories. These can be mined for execution and building commands to increase the coverage of the automated black-box execution component in the dynamic analysis system, as we have seen promising results in previous studies [PTT+18][HW17] (see section 3.2). Furthermore, we see that 43.70% of the repositories already have a JAR file (`.jar`) built. A JAR file is an executable of a Java project and can be run directly, thus in these repositories the project has been pre-built. Automated black-box execution for such projects therefore do not need to have any logic on the building phase, which leads to higher chances of coverage as it cannot fail the building phase. We have also captured the occurrences of the file extension for Text files (`.txt`), as they are fairly similar to Markdown files in their use for conveying notes and information. We have found that 29.51% of the repositories contained a Text file, which is almost one third less than the presence of Markdown files in the dataset. This makes it less interesting to mine for automated black-box execution than the Markdown files, however we will further investigate this in the keywords analysis in section 4.2.3. Additionally, we observed that the file extensions for Python (`.py`) and Shell (`.sh`) occur in 20.63% and 12.14% of the repositories respectively. This moderate presence of file extensions for scripting languages hints that projects may contain scripts for building, testing, and executing the PoCs. Such scripts can be very helpful for automated black-box execution since the script already automated the process or part of it.

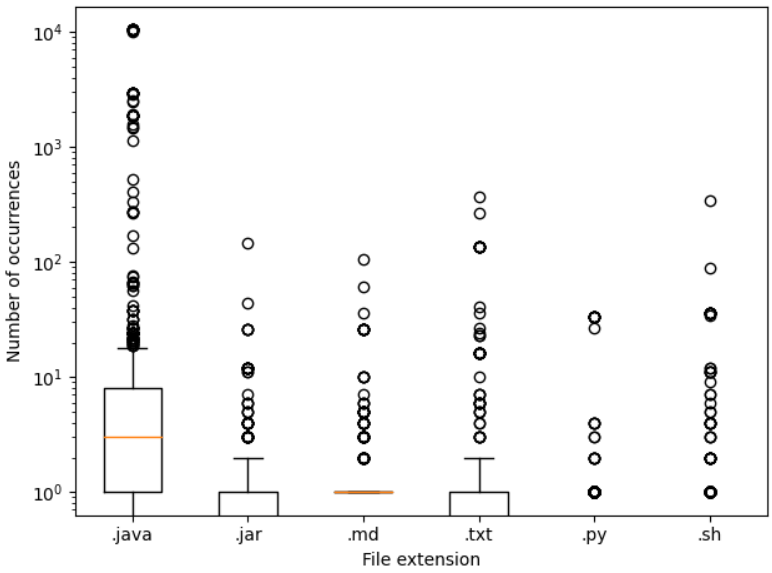

Figure 4: Boxplot of the number of occurrences of file extensions per repository in the dataset.

Figure 4 presents a boxplot of the number of occurrences of files with the analyzed file extensions per repository in the dataset. Each data point in the figure is a repository from the dataset with their corresponding number of occurrences of a file with the respective file extension. In the figure we can see that there are many outliers for the file extension `.java`. The foremost

[34]A Jakarta Server Pages (JSP) file is a text document that is used to create dynamic web pages. (https://docs.oracle.com/en/java/)

outlier is a repository containing 10,419 files with a `.java` extension. This is a repository called "*CVE-2020-0097-frameworks_base_afterfix*". The repository contains the source code of the Android platform frameworks base[35]. The repository added a fix for the CVE `CVE-2020-0097`[36], which is a vulnerability in various methods of the file `PackageManagerService.java` in the source code. This repository is actually not a PoC, but because it concerns a fix for a CVE it is included in the dataset. Several similar repositories were found in the dataset, which likely make up most of the outliers in the boxplot for the `.java` files. This implies that exploring every file in a repository will likely be a waste of resources for the dynamic analysis system. In the dynamic analysis and automated black-box execution we will therefore include measures that prevent wasting too many resources on irrelevant files and directories. Interestingly, the data on the number of occurrences of the file extension `.md` per repository is extremely homogeneous, with the majority of the dataset containing one Markdown file per repository and only a few outliers. This single Markdown file is likely the readme file (`README.md`), which is a very common file to have in projects on GitHub (see section 3.2). The file will be shown at the GitHub repository web page if it is at the root directory of the repository, and it often contains useful information about the project.

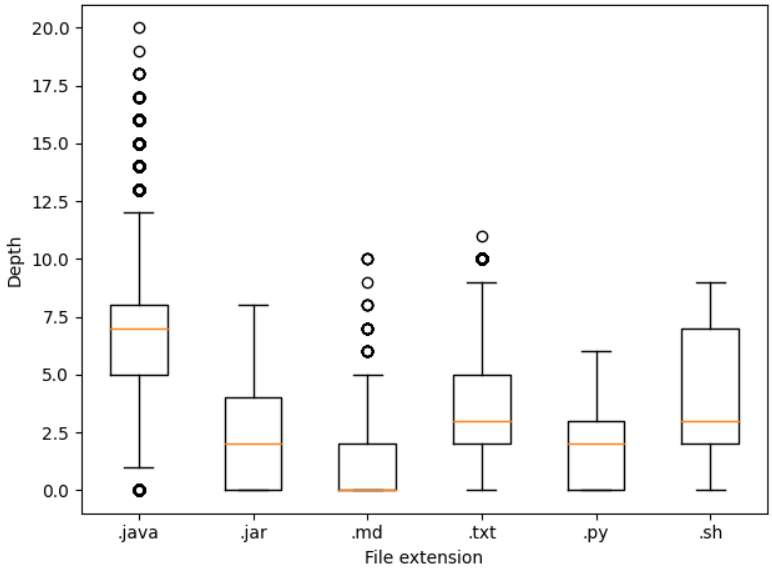

Figure 5: Boxplot of the depths of all the occurrences of file extensions in the dataset.

To further investigate the importance of the files with certain file extensions for the dynamic analysis system we have analyzed the depths of the occurrences of the file extensions. As we have mentioned, readme files that are in the root directory of a repository will be displayed on the web page of the repository in GitHub. Derived from logical sense, we can say that the closer a file is to the root directory of a repository, the higher the chance it contains instructions or other valuable information for the user. The root directory will be seen first, so it makes sense that important information for the user is close to the root, such as the readme file. This is the notion behind analysing the depths of the file extensions in the dataset. The results are represented in the boxplot

[35] https://GitHub.com/aosp-mirror/platform_frameworks_base
[36] https://nvd.nist.gov/vuln/detail/CVE-2020-0097

in figure 5. The Markdown files are indeed close to the root directory of the repositories, which further indicate that many of them are readme files. The Text files, however, are not close to the root directory. More than 50% of the Text files in the dataset are located more than 2 directories deep, meaning they likely do not contain instructions on the project. Furthermore, the Python and JAR files have an overall low depth, with 75% of Python files between a depth of 0 and 3, and 75% of JAR files between a depth of 0 and 4. This is in line with having important information for the user closer to the root, because the JAR files are the executables of the projects and, usually, want the user to easily be able to execute the project. The Python files are likely scripts that execute, build, or test the project, hence also an executable of the project. One would assume the same with Shell files, however they have a median depth of 3 and are spread over more depths, with 50% of the Shell files between the depths 2 and 7. This indicates that most of these are likely not scripts made for the user to use, but rather for development or other purposes than executing, building, and testing the project. Additionally, we see that the Java files are spread moderately uniform over a wide spectrum of depths, with 50% of the data located between depths of 5 and 8. This is in line with the conventional Java projects. The standard directory layout given by Apache Maven[37] and in the Oracle Java tutorials[38] have the `.java` files already three layers of directories deep. This can easily add up when developers want to differentiate between certain code as the projects get larger by dispersing the files into more directories, and we have seen in figure 4 that the projects are fairly large given the data shape of the number of occurrences of Java files.

4.2.2 Build Files

There are several different ways to compile and build a Java project. Given that Java projects are often very tedious to compile by hand, there exist many build automation tools, such as Apache Maven[39], that simplify and automate the build process. They are widely utilized and present in nearly every Java project. There is, however, not one universally used build automation tool. To automate the execution of the PoCs we must also automate the building of the PoCs, thus it is essential to identify what build automation tools are used in the PoC repositories. In the data analysis we captured the presence of build tools by counting the occurrences of the characteristic build file of ten popular build tools commonly used in Java projects. These build files are often unique for each build tool, which makes it easy to differentiate the presence of different build tools. The ten analyzed build automation tools with their corresponding build file are: Apache Maven (also called Maven) (`pom.xml`), Docker (`Dockerfile`)[40], Gradle (`build.gradle`)[41], Apache Ant (also

[37] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[38] https://docs.oracle.com/javase/tutorial/

[39] Apache Maven is a build automation tool developed by the Apache Software Foundation. (https://maven.apache.org/)

[40] Docker is a platform for developing, shipping, and running applications developed by Docker, Inc. (https://www.docker.com/)

[41] Gradle is a build automation tool developed by Gradle, Inc. (https://gradle.com/)

called Ant) (`build.xml`)[42], GNU Make (`Makefile`)[43], sbt (`build.sbt`)[44], Bazel (`BUILD(.bazel)`)[45], SCons (`SConstruct`)[46], Leiningen (`project.clj`)[47], and Buck (`BUCK`)[48]. It is important to mention that not all build files are completely unique. For example, the build file for Bazel (`BUILD`) has the same name as the build file for Pants[49]. This is the only exception we managed to find for the ten analyzed build tools. We would also like to note that this analysis does not take into account all of the possible build tools that can be used for Java projects, there are way more than just ten build tools, simply too many to include all of them in this analysis. We have therefore included the most popular ones we could find. The results of the analysis can be seen in table 2, and figure 7.

	Number of repositories	Percentage of dataset
pom.xml (Maven)	525	49.02%
Dockerfile (Docker)	227	21.20%
build.gradle (Gradle)	225	21.01%
build.xml (Ant)	51	4.76%
Makefile (GNU Make)	21	1.96%
build.sbt (sbt)	1	0.09%
BUILD(.bazel) (Bazel)	0	0.00%
SConstruct (SCons)	0	0.00%
project.clj (Leiningen)	0	0.00%
BUCK (Buck)	0	0.00%

Table 2: Number and percentage of repositories in the dataset that have the respective build file present.

Table 2 represents the number and percentage of repositories in the dataset with at least one file in the repository with the respective analyzed build file. Maven, Gradle, and Ant are at the top, which is in line with the three current most popular Java build automation tools [HMLW17] [BGZ17]. Here we exclude Docker, since it is primarily used for deployment, containerization and packaging applications, and not for building. Furthermore, Docker is likely used combined with a build automation tools, such that the build automation tool does the building and Docker does the deployment. This is why Docker is a special case, but we still included it in the analysis as it shows specific building characteristics. Utilizing Docker for building and for running PoCs introduces complexities to our dynamic analysis system, as the application runs in its own isolated environment. Monitoring the application's behavior becomes more challenging compared to a traditional application running in a local environment. Considering these additional complexities and based on the relatively moderate prevalence of Docker (21.20%) opposed to traditional build

[42]Apache Ant is a build automation tool developed by the Apache Software Foundation. (https://ant.apache.org/)
[43]GNU Make is a build automation tool developed by the Free Software Foundation. (https://www.gnu.org/software/make/)
[44]sbt is an open-source build tool created by Mark Harrah. (https://www.scala-sbt.org/)
[45]Bazel is an open-source build and test tool developed by Google LLC. (https://bazel.build/)
[46]SCons is an open-source build tool created by Steven Knight. (https://scons.org/)
[47]Leiningen is an open-source build automation tool created by Phil Hagelberg. (https://leiningen.org/)
[48]Buck is a build tool developed by Meta Platforms, Inc. (https://buck.build/)
[49]Pants is an open-source build system developed by Twitter, Inc. (https://www.pantsbuild.org/)

automation tools (Maven, Gradle, Ant, GNU Make), we have decided not to implement specific functionality for Docker in our dynamic analysis system. At last, we found that the build files for Bazel, SCons, Leiningen, and Buck are not present in any of the repositories. In total, when we exclude the occurrences of Docker files, we have found a build file in 823 PoC repositories, which is 76.84% of the dataset, without considering that some build files may be in the same PoC repository. We have taken all these results into account when building the dynamic analysis system by prioritizing functionality for the most used build automation tools, with the purpose of getting a higher coverage of analyzed PoC repositories.

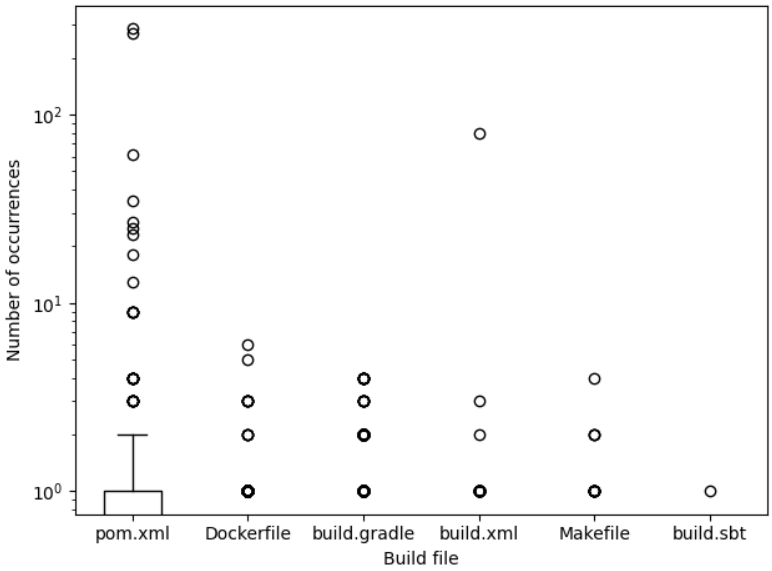

Figure 6: Boxplot of the number of occurrences of build files per repository in the dataset.

Figure 6 shows a boxplot of number of occurrences of build files per repository in the dataset, excluding the build files that we did not find any occurrences of (see table 2). Each data point is a repository in the dataset with its corresponding number of occurrences of the respective build file. In line with the findings in table 2, most build files are not noticeably present in the dataset, which is why most of the data points are at y-axis 0 and no box is present, except for the Maven build file `pom.xml`. 75% of the repositories in the dataset have either one or zero Maven build files, while 25% repositories contain multiple Maven build files. Additionally, we see relatively far outliers for the build files `pom.xml` and `build.xml`.
Figure 7 presents a boxplot of the depths of all the occurrences of the build files in the dataset, excluding the build files that we did not find any occurrences of (see table 2). Each data point is a build file with its corresponding depth that it is located at from the root directory of its repository. A build file is generally at the start of a project, so it makes sense that the build files are located close to the root directory of the repositores. We can see in the boxplot that the build files tend to be close to the root as all of the lower quartiles are (LQ or Q1) at depth 0, excluding the single occurrence of `build.sbt`, however some build files are still notably spread far from the root. Especially, `build.xml`, the build file for Ant, is spread over many further depths, with 75% of the depths between 0 and 5, and has a median of depth 3. The build file for Maven, `pom.xml`, is also moderately spread, with a median of 2, and 75% of the depths between 0 and 3, however this

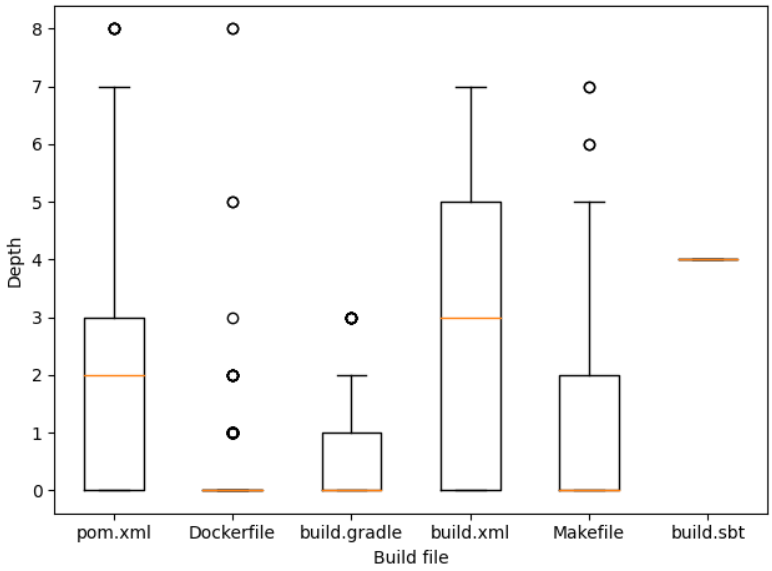

Figure 7: Boxplot of the depths of all the occurrences of the build files in the dataset.

tend to be much closer to the root than is the case with `build.xml`. The build files `Dockerfile`, `build.gradle`, and `Makefile` are overall more closely located at the root, with all of them having 50% of their depths being 0. A reason for the spread over higher depths for the build files for Maven and Ant might be that they are present in larger repositories. We already know that the dataset contains repositories with more than one PoC from the study that collected the dataset [YTG23] (see section 4.1), and in the boxplot in figure 6 we have seen that there are far outliers of repositories containing many Maven and Ant build files. We have manually inspected some of the PoC repositories and saw that these outliers often contained deep `pom.xml` files, for example we found that the two foremost outliers for `pom.xml` with in total 286 and 269 `pom.xml` files have the build files mostly located in depths between 3 and 5. We also observed that most of these deeper build files are likely unimportant sub-projects, therefore it is reasonable to implement a limit of the discovery of build files to a certain depth for the automated black-box execution. From the data shapes in the boxplot we must also derive that this discovery of build files should not be restricted to a depth 0, because the build files are still spread moderately over other depths close to the root.

4.2.3 Keywords

In the data analysis of file extensions (see 4.2.1) we mentioned the use of Markdown (`.md`) files, specifically readme files, to obtain information about executing and building the project. Mining readme files is of high interests to us, since we are trying to automatically run the PoCs without knowing the internal workings of the PoC (see related work on readme mining in section 3.2). A readme file containing instructions on the PoC is therefore to great value for the automated black-box execution process. To gain more insight on the Markdown (`.md`) and text (`.txt`) files we have checked them in this data analysis. In this analysis the analysis script reads all the Markdown

and text files in the PoC repositories and counts the occurrences of the keywords *"java"*, *"javac"*[50], *"python"*, *"python3"*[51], *"bash"*, *"mvn"*, *"gradle"*, and *"ant"*. The script does not count sub-strings, for example *"important"* is not counted as an occurrence for the keyword *"ant"*. The occurrence of these keywords may indicate an instruction on how to execute the project considering that these words are the commands needed to use their respective software. We have only considered the Markdown and text files that were in UTF-8[52] format for simplicity of implementation. Consequently, this meant that we did not take into account seven text files from in total five different PoC repositories in this analysis. In our results in table 4 we have therefore not included those five repositories to obtain the percentage of the repositories with a text file that contains an execution, i.e. we divided by 311 instead of 316. The results of this analysis can be seen in table 3 and table 4.

	Number of repositories	Percentage of repositories with `.md` files
"java"	485	50.05%
"python"	243	25.08%
"mvn"	153	15.79%
"bash"	140	14.45%
"javac"	72	7.43%
"python3"	43	4.44%
"gradle"	15	1.55%
"ant"	3	0.31%

Table 3: Number of repositories containing a Markdown file with the respective keyword and the corresponding percentage of the found repositories with a Markdown file.

We have found that 50.05% of the found Markdown files (`.md`) contain the keyword *"java"*, which indicates a Java command. This is as expected as these are Java projects. In line with the findings of the file extension analysis (see 1) we see that the scripting languages Python and Bash are moderately present in the Markdown files, with a presence of 14.45% for the keyword *"bash"*, 25.08% for *"python"*, and 4.44% for *"python3"*. Furthermore, the keywords for the build automation tools (*"mvn"*, *"ant"*, *"gradle"*, and *"javac"*) are also moderately present, with a combined presence in 243 repositories, which is 25.08% of all the Markdown files, if we do not account for the possibility of multiple keywords that occurred in the same repository. In the text files there are very few occurrences for all the keywords that we have analyzed, with the highest occurring keyword *"python"* only occurring in 2.89% of all the text files. This implies that the text files are much less used to communicate commands and instructions on the project, which is in line with the data shape of the depths of the found text files (see figure 5), which is farther from the root directory than for the Markdown files. In both the Markdown and Text files, we have found that the keyword for Python is used more often than the keyword for Python3. Overall, the findings show promising indication

[50]Javac is the primary Java compiler included in the Java Development Kit (JDK) from Oracle Corporation. (https://www.oracle.com/java/)

[51]Python 3 is the latest major version of the Python programming language. It is a revision of the previous major version Python 2 and contains backward-incompatible changes. (https://www.python.org/)

[52]UTF-8 (Unicode Transformation Format, 8-bit) is an encoding standard for representing Unicode characters. (https://home.unicode.org/)

	Number of repositories	Percentage of repositories with `.txt` files
"python"	9	2.89%
"mvn"	6	1.19%
"java"	5	1.61%
"ant"	2	0.64%
"gradle"	1	0.32%
"javac"	0	0.00%
"python3"	0	0.00%
"bash"	0	0.00%

Table 4: Number of repositories containing a Text file with the respective keyword and the corresponding percentage of the found repositories with a Text file.

of potential meaningful instructions on how to build and execute the project in the Markdown files, but not in the text files.

To increase the accuracy of the analysis we have also counted the occurrences of the strings inside a code block in the Markdown files. In this analysis we made a distinction between indented code blocks, code snippets using single backticks (`), fenced code blocks using three backticks (```), and fenced code blocks using three tildes (~~~). For more information on code blocks see section 2.4. We have separated the different forms to create a code block in Markdown in this analysis to see which form of code block is used most frequently. The combined results of code block usage can be seen in table 5. In table 6 we show the findings of the differences between the four code block forms.

	Number of repositories	Percentage of repositories with respective keyword
"java"	417	85.98%
"python"	127	52.26%
"mvn"	135	88.24%
"bash"	138	98.57%
"javac"	50	69.44%
"python3"	42	97.67%
"gradle"	7	46.67%
"ant"	3	100%

Table 5: Number of repositories containing a Markdown file with the respective keyword in a code block and the corresponding percentage of the found repositories with a Markdown file containing the respective keyword.

We have found that the keywords are used significantly more inside code blocks than outside code blocks, which indicates that many of the occurrences of the keywords are potential commands. In total we have found that 79.64% of the examined keywords are inside a code block. Based on these

findings along with the high presence of 90.48% (see table 1) of Markdown files in the repositories we have reason to pursue the mining of the Markdown files on potential commands for executing and building the project.

	Distribution
Three backticks (```)	731 (79.54%)
Single backtick (`)	178 (19.37%)
Indentation (four spaces or one tab)	8 (0.87%)
Three tildes (~~~)	2 (0.22%)

Table 6: The distribution of the four different code block forms that was found to contain a keyword in the Markdown files.

As we mentioned there are different ways to create a code block, but to know which one(s) we should focus on when mining code blocks in the Markdown files we have analyzed the distribution of the four different code blocks that contained a keyword in the Markdown files, i.e. the distribution of the code block forms of the data in table 5. This distribution is presented in table 6. We see that the code blocks created with three backticks (```) and with single backticks (`) are found to be used most frequently for the analyzed keywords, with a significantly high presence of 79.54% for three backticks and 19.37% for single backticks. Combined, they make up 98.91% of the found code blocks with a keyword. Based on these findings the focus should be directed towards extracting the keywords from these two forms of code blocks, rather than the code blocks created with indentation or three tildes (~~~).

5 Architecture

In this section we will discuss the architecture of the automatic dynamic analysis system that we have created. The architecture is divided into two core components: the Automated Black-box Executer and the sandbox. The Automated Black-box Executer is a tool that will automatically build the PoCs and find commands that run the PoCs. The sandbox is the isolated environment on which the PoCs and the Automated Black-box Executer will by run securely while monitoring the activity. By leveraging the Automated Black-box Executer we are able to fully automate the dynamic analysis process. In this section we will demonstrate how the dynamic analysis of PoCs has been automated by diving into the architecture.

5.1 Automated Black-box Executer

To automate the dynamic analysis process we must automate the execution of the PoCs. To automatically execute the Java PoCs in the Sandbox we have made a Python3 program that tries to find inputs that run the PoC as intended. This program relies on examining the inputs and outputs of example executions to find correct inputs. In section 2.2 we called this automated black-box execution, and we will now call the program that performs this Automated Black-box Executer (ABE). Automating this process will likely mean that only a proportion of the PoCs will be analyzed, since the ABE will likely not be able to find valid inputs for all the PoCs in the data-set. This is an inherent trade-off when opting for automation. However, the ABE will allow us to create a dynamic analysis system that can automatically analyze many PoCs with minimal manual effort, thus allowing for easy re-usage and distribution.

We have developed the ABE using the findings of the data analysis (see section 4.2) and previous research in automated black-box testing [MPZ15] [ZWCX22] (see sections 2.2 and 3.3), and in readme mining [PTT+18] [HW17] (see section 3.2). The ABE can be divided into four different components, that can be seen in the diagram of the ABE in figure 8. The ABE takes as input a path to the repository containing the PoC as it is uploaded on GitHub. The ABE will then **(1)** build and compile the project (automated black-box building), **(2)** mine all the Markdown files (`.md`) for build or execution commands,**(3)** run all scripts (`.sh` and `.py`), and **(4)** fuzz the Java executables for valid inputs. The first component, which does the automated black-box building, may be run separately from the other components. This will be necessary for the dynamic analysis process where we will want to restrict network connection for the executions of the PoCs, but not for the building. Additionally, it is important to note that we excluded several files and directories from searching for files in the ABE, like with the data analysis (see section 4.2). We consider these files and directories unimportant to our research and unwanted. For instance, we did not continue the traversal of the file structure when the directory or file started with a dot, e.g. the *.git* directory. In the end, the ABE will have found valid build and execution commands that we can use in the sandbox to perform dynamic analysis on. The ABE will be also run in the sandbox for security. Once the ABE is done we will revert to a base-line snapshot to execute the found commands for dynamic analysis.

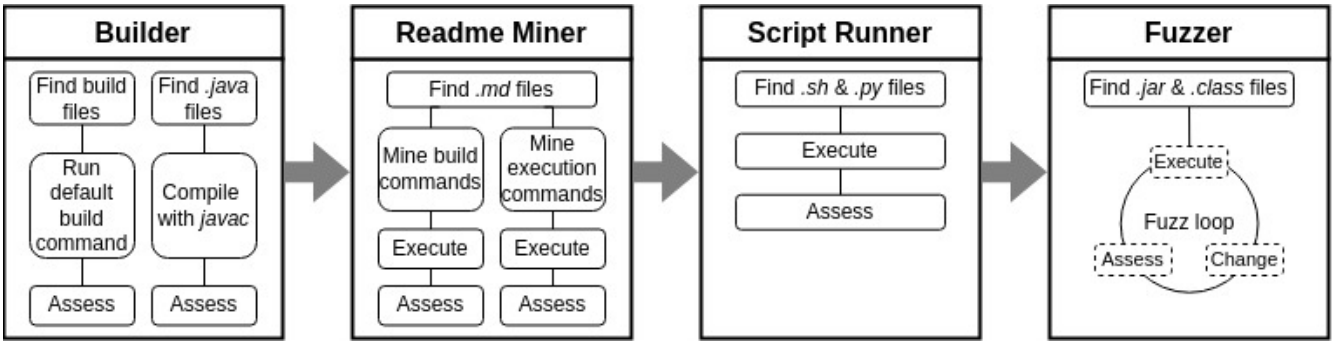

Figure 8: Diagram of the Automated Black-box Executer.

5.1.1 Builder

To execute a PoC it must first be built, which is the purpose of the Builder component. The Builder component will find all the build files within the repository and then run the default build commands for the respective build automation tools. Based on the occurrence rates of the build files from table 2 in section 4.2.2, we have chosen to implement the building of Maven, Gradle, Ant, and GNU Make in the repositories. We have excluded Docker for the reasons we mentioned there, in short: Docker uses containerization which adds complexities when dynamically analysing it compared to traditional programs that are run directly in the local environment.

We will not search the whole file structure of the PoC repositories to find the build files, because of the findings of the depths analysis of the build files (see section 4.2.2 and figure 7). Given these findings we have chosen to limit the search of the build files in the Builder to a depth of two, thus we will only look for build files in at most two directories away from the root of the repositories. After finding the build files the Builder will execute the default build commands according to their respective build tools. Table 7 contains the build commands that are used. After executing the default build commands for the found build files, we will assess if the build was successful by looking at the `returncode`. If a build failed we do not include it in the list of found build commands, hence it will not be executed in the sandbox. Unfortunately, it is not uncommon for a build to fail its default build command [SP16] [SBM+20] [HMLW17] (see section 3.1). To hopefully reduce the number of failed builds we have added the absolute path of the build file to the build commands (see table 7. This should reduce errors caused by not being able to find the build file. The Builder will already find these build files to know which build command to perform, so it is very easy to implement this. For further reduction of failed builds we rely on the Readme Miner (see 5.1.2) to find build commands given by the author of the project in the Markdown files.
Besides building with build automation tools, the Builder component will also perform the compilation of the Java files (`.java`) to a certain depth with Javac. We have limited the finding of the Java files to a depth of 1, because we only want to include the cases where a few Java files make up the PoC and with such small projects the Java files are likely not rooted deeply into the file structure. Here we want to include the analysis of such small projects that a build automation tool isn't used.

Compiling with Javac will create corresponding class files (`.class`)[53]. Building with a build

[53]A Java class file contains bytecode that is executable on the Java Virtual Machine (JVM). (https://docs.

Build tool	Used build command
Maven	`mvn clean package --batch-mode --file` *[path to build file]*
Ant	`ant clean ; ant -buildfile` *[path to build file]*
Gradle	`gradle clean assemble --build-file` *[path to build file]*
GNU Make	`make --file` *[path to build file]*
Javac	`javac` *[path to .java file]*

Table 7: The build tools and corresponding build commands that are executed by the Builder component.

automation tool will generally create at least one of three files: JAR files (`.jar`), WAR files (`.war`)[54], or EAR files (`.ear`)[55]. In this research we are interested only in JAR files, since these are standalone Java applications. WAR files need to be deployed and run on a web server, such as Apache Tomcat[56]. EAR files also need to be deployed. Deployment adds complexities to the dynamic analysis system, like with Docker, which is why we have decided to ignore these and focus only on standalone applications, such as JAR and class files. The Builder component does not have to pass the JAR and class files to the Fuzzer component in the ABE for fuzzing the executables, because the Fuzzer will search for the files in the repository. This is because not every file project will have the same file structure, and some developers may even have changed the default build directory of the build automation tools. For example, the build directory for Maven is called "target", but can easily be customized in the Maven build file (`pom.xml`). It is therefore unknown, without any added functionality, to confidentially know where the specific JAR files are. The Builder will therefore not pass the files directly to the Fuzzer, but it will be run before fuzzing, therefore indirectly passing the result files to the Fuzzer. Furthermore, this implementation allows for the builder component to be run separately from the other components in the ABE, which will come in handy when we want to have different network settings for the builder component than for the other components.

5.1.2 Readme Miner

Based on the findings of the data analysis (see section 4.2) and previous research on readme mining [PTT+18] [HW17] (see section 3.2), we have decided to mine the Markdown files in the PoC repositories to find build and execution commands. The Readme Miner component will look for Markdown files in the given PoC repository and mine potential build and execution commands, which it will then execute and assess, based on the `returncode`. The commands that were found and ran successfully will go to the output of the ABE. The search for the Markdown files will be limited to a depth of one directory from the root of the repository, since we saw that the Markdown files are generally close to the root directory (see 1 in section 4.2.1), and files with useful instructions about the project, such as readme files, are likely close to the root of the project.

`oracle.com/javaee/`)

[54]WAR (Web Application Archive) files are packaged web applications in Java. (`https://docs.oracle.com/en/`)

[55]EAR (Enterprise Archive) files are packaged enterprise-level Java applications. (`https://docs.oracle.com/javaee/`)

[56]Apache Tomcat is an open-source Java web server and Servlet container. (`https://tomcat.apache.org/`)

In section 4.2.3 we found that 79.64% of the examined keywords are inside a code block. Given the high presence and the notion that the potential value of the keywords inside a code block are higher than outside a code block, since the purpose and use case of code blocks are to give instructions and commands, we will mine the keywords inside the code blocks. Furthermore, we found that most of the found keywords in a code block were in a code block created with three backticks (“‘) or with single backticks (‘), which accounted for 98.91% of the results (see table 6). We will therefore only mine the potential commands from these two forms of code blocks. The mining is very simple and the exact similar way as we extracted them for the data analysis in section 4.2.3. The Readme Miner will find the code blocks by splitting the contents of the Markdown files on a single backtick (‘). This will result in a list where each element was between two single backticks in the Markdown file, except for the first element which is the content between the start of the file and the first single backtick in the file. If we then take every second element starting from the second element (so excluding that first bit of content before the first backtick) we will have all the code blocks. We can now check if a keyword is present in this code block. When we find a keyword we create a command starting with the keyword till the end of the line that contained the keyword. We do this such that we can keep potential arguments in the command. There is few risk of failing the command because of giving too many arguments, because these will usually be ignored by the program. For example, a program that needs only two arguments will just ignore all the arguments past the second one. It is reasonable to do this, since extracting just the command and the file it is supposed to run is simply not worth mining the Markdown files for. Besides, we will already run that in the Fuzzer component. Therefore it is better to extract the command with potential arguments and have the possibility of command failure caused by non-intended arguments. In table 8 the keywords that are mined from the Markdown files are listed.

Build commands	`javac`, `mvn`, `gradle`
Execution command	`java`, `python`, `python2`, `python3`, `bash`

Table 8: The build and execution commands that are extracted from the Markdown files by the Readme Miner.

After finding the commands from the Markdown files the commands will be refined by replacing every file name in the command with their corresponding absolute path that the ABE has found. This measure hopefully reduces the amount of failed commands by “file not found” errors, because some may need execution from a specific current working directory (cwd). The commands that are found will then be executed and assessed based on their `returncode`. If the `returncode` of the execution was 0, the program ran without errors and we will count it as a valid execution. At last, the commands with valid executions will go to the output of the ABE.

5.1.3 Script Runner

The Script Runner will find all the shell files (`.sh`) and Python files (`.py`), and execute them without any arguments. In the data analysis (see section 4.2.1) we found that there are 221 PoC repositories containing files with a Python extension (19.59% of total data-set) and 177 PoC repositories containing files with a shell extension (15.69% of total data-set). After some manual inspection we have seen that these are often scripts used for running, testing or building the project. This also makes sense since the main programming language of the projects are Java. We will search

for the Shell and Python files from the root of the repository to a depth of 2, given that we found that both file extensions had a median depth of 2. The average depth for the Python files is 1.92, which is also around 2, however for the shell files this is 3.72. The median depth of 2 shows that the average is skewed upwards, and given that scripts are probably closer to the root because that is where you would generally want your instructions and executables on the project, we will only be interested in the files with a lower depth. This is why we have chosen to limit the search depth of 2. After searching the shell and Python files they will each be executed and assessed. The shell scripts will be executed using Bash, and the Python scripts will be executed with Python3 ("python3") and then with Python2 ("python2") if it was unsuccessful. We have decided to only execute it once without any arguments. If the `returncode` of the execution is 0 we take the command to the output of the ABE. Our main focus is on running the Java PoC and not the scripts, that is why we did not implement any fuzzing or anything more than simply running the scripts once. For more complex script commands we rely on the Readme Miner to find it in the Markdown files.

5.1.4 Fuzzer

The Fuzzer, derived from automated black-box testing [MPZ15] (see section 2.2), will try to find commands that execute corresponding program correctly. In other words it will try to trigger the program, instead of finding defects or errors. This component has not all the characteristics of fuzzing that is known in the testing field, however we have chosen this name because of the characteristic similarities with fuzzing and random testing. We have described the similarities between the two concepts of automated black-box testing and automated black-box execution in section 2.2.

The Fuzzer will first find the executable files that were built in the previous components or were already in the repositories: the JAR files (`.jar`) and class files (`.class`). We have chosen to limit the search for the JAR and class files to a depth of 3, which, based on the findings of the data analysis (see figure 5 in section 4.2.1), should discover 66.92% of the pre-built JAR files, which is about 313 JAR files. Although JAR files are the executables of the projects, we still decided to implement a limit on the discovery of the JAR files, because we do not want to waste too much resources on executing irrelevant sub-projects. It makes logical sense that an important JAR that the developer wants the user to run is close to the root, therefore deeper JAR files are likely irrelevant and fuzzing them would waste too much resources as the Fuzzer component is likely the most exhaustive component.

After the executables are found, they will be fuzzed using a predefined list of constant arguments we have chosen ourselves. We have chosen arguments that are often used in PoC exploits, based on manually inspecting the dataset and discussions with an experienced pentester with sound knowledge of PoC exploits. Here we utilize the fact that our target program is a PoC, therefore we can guide the process by giving more related arguments. The arguments can be differentiated into categories to manage coverage of different domains of arguments. In table 9 the categories and their respective arguments constants that are used by the Fuzzer are shown. The file argument is not a constant, but randomly added to the arguments list out of the list of jar files when the Fuzzer is called, such that the Fuzzer is able to provide an argument of an existing file.
The Fuzzer will try to find inputs that successfully execute the programs with the Fuzz loop or

Category	Argument String
(Port) Number	`80`
URL	`http://localhost`
IP address + Port	`127.0.0.1:80`
IP address	`255.255.255.255`
Host name	`google.com`
Command	`whoami`
Path	`./`
String	`user`
String	`pass`
String	`admin`
File	*`[path to random jar file]`*

Table 9: Arguments list and their respective categories used by the Fuzzer.

trial-and-error cycle that can be seen in figure 9. This cycle consists of three distinguishable parts: **(1)** Execute, **(2)** Assess, and **(3)** Change. We call it trial-and-error, because the Fuzzer is essentially trying out different inputs/commands and it keeps trying until it has found an input that works. The Execution part will simply run the command, then the Assess part will assess the `returncode`, `stderr`, and `stdout` of the execution. If the `returncode` is 0, we have found a successful run, so we stop with fuzzing and exit the loop, otherwise we continue and change the command based on the information in `stderr` and `stdout`. This information is assessed in the Assess part and will tell the Change part what kind of changes to make to the command. If `stderr` contains an exception for an array index out of bounds (`java.lang.ArrayIndexOutOfBoundsException`), a random argument from the arguments list will be added to the command. This is because such an exception indicates that more arguments are required by the program, see section 2.3 for more information. If the particular exception is not in `stderr` the Fuzzer will change the current last argument using a new argument from the arguments list (see table 9). After the change the command will start the loop again at the Execution part. This approach indirectly discovers whether a particular argument was valid. For instance, if we previously encountered an error and now encounter an "array index out of bounds" exception, we can deduce that the previously used argument was valid. This is because the program now reports an error that it requires a subsequent argument. The Fuzzer therefore contains implementation for indirectly determining the validity of an argument, simply by looking at a single error output. Assuming the programmer has not modified the handling of the "array index out of bounds" exception, the Fuzzer is guiding the fuzzing process by determining whether an argument is required or whether the last argument should be replaced. The Fuzzer will initially run the program without any arguments, then the Fuzzer continue looping over the Fuzz loop until a particular argument position in the command has tried all the arguments in the arguments list (see table 9).

There is one deviating path that can also exit the loop, as you can see in figure 9, which is when a command is found in `stdout`. When inspecting some PoCs we have seen that some PoCs will give an example execution in `stdout` when the input was not correct to instruct the user on how to execute the program, which is a common thing also with other programs. This is why the Fuzzer will look at the `stdout` and basically mine Java commands from it in a similar way the Readme Miner mined commands from a Markdown file. When a Java command is found in `stdout` it will

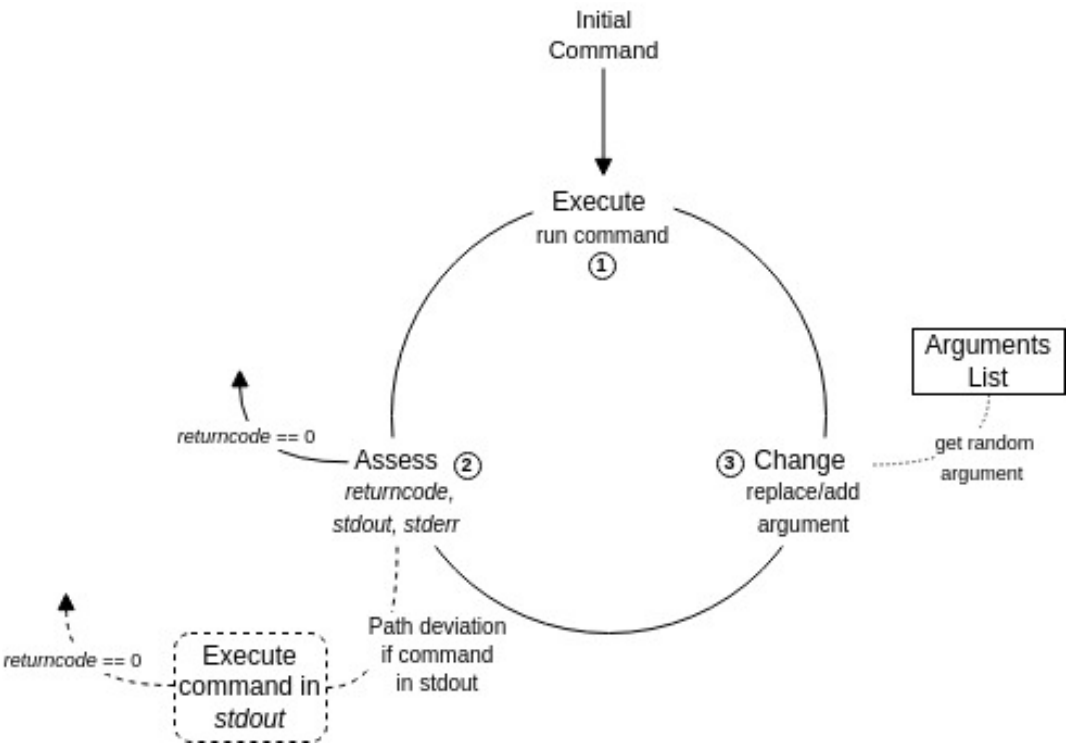

Figure 9: The Fuzz loop.

be executed and if it has a `returncode` of 0 the Fuzzer will exit the Fuzz loop, otherwise it will just continue fuzzing with the initial input/command and disregard the command found in `stdout`.

To prevent the Fuzzer from running infinitely we have implemented a couple measures. First of all, time-outs are used when executing the commands to make sure they do not run too long, see 5.1.5. Secondly, we have limited the amount of cycles the Fuzzer will do. This is by default set to ten cycles, but can be changed through a command-line argument when running the ABE. When the Fuzzer uses up all of the cycles and has not found a successful execution yet, it will stop fuzzing. Eventually the commands with a successful execution will go to the output of the ABE.

5.1.5 Execution Time-outs

We have implemented time-outs when executing the commands as subprocesses as we do not want them to run too long or stall the analysis. The Python module subprocess for subprocess management[57] offers time-out functionality, however this implementations is too soft when dealing with programs that may ignore or interfere termination requests, furthermore spawned subprocesses of the subprocess also must be terminated. We have therefore implemented our own implementation for time-outs, using the threading module[58] and psutil module[59]. In this implementation we force the termination of the child processes, instead of requesting them to finish and terminate themselves. This implementation can be seen in listing 5. We trimmed the function to only include

[57] https://docs.python.org/3/library/subprocess.html
[58] https://docs.python.org/3/library/threading.html?highlight=threading#module-threading
[59] https://psutil.readthedocs.io/en/latest/

the implementation of the time-out and execution of the commands, but note that in the actual code the `returncode`, `stderr`, and `stdout` of the child process are returned, and logging is performed. First we open the child process for the command, then we activate a timer from the threading module with our `'timeout_callback'` function and a time period. This timer will run in a separate thread and call the `'timeout_callback'` function when the given time period is reached. This `'timeout_callback'` function will kill the child process and all the spawned sub processes. We have a loop that captures the time-out event or the completion of the child process. The ensure robustness of this loop and the time-out implementation we have added a maximum number of iterations the loop can perform, such that in the case that the timer did not work properly we still ensure a limited run-time of the child process. When the loop reached the maximum number of iterations it will exit the loop and call the `'timeout_callback'` function, which will kill the subprocesses. This time-out implementation should prevent stalling of the ABE when running subprocesses. The implementation will terminate the child process and its spawned subprocesses, except for one situation where the subprocess is a background process which is unrelated to the child process. In that case the subprocess will keep running after the child process is terminated and it will not be a subprocess of the child process, which means it is not killed by our implementation. However the main goal of not stalling the execution of the ABE will be accomplished, and given the difficulty of including this situation in the implementation we will keep it as is. We make a distinction between building and execution commands, since building may required a higher up-time for downloading dependencies, therefore we will use different time periods for the time-outs. By default we use 30 seconds for a non-build command, and 90 seconds for a build command. The maximum amount of iterations the internal loop can do is, by default, set to the time period of the respective command plus 5 (i.e. 35 iterations for non-build commands, and 95 for build commands). Furthermore, we make a distinction between time-outs of the two commands, because a time-out of an execution command likely means it worked, since it did not terminate in the for the time-out time period. It is the generally expected behaviour of a successfully run of a program, since termination is generally done through user-input rather than automatically. A build automation tool should generally terminate when it is done building. We therefore return an `returncode` of -1 for a time-out of build commands, and a `returncode` of 0 for time-outs of a non-build command. Note that this implementation will therefore also count non-build executions that timed out because of other reasons as successful.

```
import subprocess
import time
from threading import Timer, Event
import psutil

def execute_cmd(cmd):
    try:
        child = subprocess.Popen(cmd,
                                 stdout=subprocess.PIPE,
                                 stderr=subprocess.PIPE,
                                 text=True,
                                 shell=False)
    except Exception as e:
        # Failed to open child process with command
        ...

```

```
    timeout_occurred = Event()

    def timeout_callback():
        # Kill the child process and all its sub processes
        try:
            parent = psutil.Process(child.pid)
            children = parent.children(recursive=True)
            for child_proc in children:
                child_proc.kill()
        except psutil.NoSuchProcess:
            pass
        nonlocal timeout_occurred
        timeout_occurred.set()

    # Starting timer
    timer = Timer(TIMEOUT_TIME, timeout_callback)
    timer.start()

    loop_iterations = 0
    while loop_iterations < MAX_LOOP_ITERATIONS:
        if timeout_occurred.is_set():
            # The child process timed out
            ...

        if child.poll() is not None:
            # The child process completed
            timer.cancel()
            ...

        time.sleep(1)
        loop_iterations += 1
    # The child process reached maximum loop iterations
    ...
```

Listing 5: Implementation of executing execution commands.

Additionally, we have implemented time-outs when reading the `stdout` and `stderr` of the child process to prevent blocking. It is possible for the whole ABE to block when reading the output of `stdout` or `stderr` with the `communicate`[60] function from the subprocess module, as it will read until an end-of-file (EOF) is found, however when this is not given it will block the whole program. The `stdout` and `stderr` will be used by the Fuzzer and for logging, therefore we have used the `communicate` function with its implemented time-out functionality. This implementation can be seen in listing 6. The default time-out period for this is 10 seconds.

```
try:
    stdout, stderr = child.communicate(timeout=COMMUNICATE_TIMEOUT)
except subprocess.TimeoutExpired:
    # Timed out
    ...
```

Listing 6: Implementation of reading `stdout` and `stderr`.

[60] https://docs.python.org/3/library/subprocess.html#subprocess.Popen.communicate

5.1.6 Output

After all the components are done the successful build and execution commands are outputted. We have made a distinction between the different commands, because we cannot run an execution command of, for example, a JAR file (`.jar`) without first building it. The order of execution is of high importance for the dynamic analysis. Additionally, script commands are likely either scripts for building, executing or testing (although testing is irrelevant for the research), this is why these should also be executed before the actual PoC executions. In the output we will therefore make distinction between build commands (`mvn`, `gradle`, `ant`, `make`, and `javac`), Java commands (`java`), Python commands (`python2` and `python3`), and Bash commands (`bash`). The sandbox can then execute the commands in this order: **(1)** build commands, **(2)** Bash commands, **(3)** Python commands, **(4)** Java commands. The different command lists will be exported to a JSON file (`.json`)[61] target that is given through a command-line argument when executing the ABE. The sandbox can then read the commands from the JSON file and execute them in the mentioned order.

For performance information and research purposes the ABE will also output the commands to a CSV file (`.csv`) target, which is also given through a command-line argument. In this CSV file the distinction will be made between the commands found in the four different components: Builder commands, Readme Miner commands, script commands, and Fuzzer commands. This allows us to review the performance of the different components. Additionally, the run time of the ABE will also be stored here. This information is not required by the sandbox, therefore we made a distinction between the two files.

5.2 Sandbox

The dynamic analysis will be conducted in a sandbox, which is an isolated environment to prevent any harm or damage to production systems or other systems(see section 2.1). The sandbox will be used to run the ABE for finding the build and execution commands, which is followed by to monitoring the executions of the found commands. The sandbox is a virtual machine (VM) that emulates a standard user environment with monitoring tools installed for dynamic analysis. We have automated the dynamic analysis process using Shell scripts and the ABE for finding commands with valid executions to analyze. In this dynamic analysis system only manual interaction required is in the installation process of the VM and adapting the settings of the dynamic analysis system for your personal analysis and preferences. The VM created in this research has the operating system (OS) Ubuntu 20.04.6 LTS (Focal Fossa) Live Server[62]. The steps of the installation process have all been documented, which make it possible to recreate the dynamic analysis system. It should also be possible to create it with an other OS by applying the main steps relative to this OS if the versions of the software allow this, We have done this installation process with a Windows 10 Pro 22H2[63] too and documented the steps relative to this OS. In this section we will discuss the architecture of the sandbox we created and how the automation of the dynamic analysis process is achieved. Furthermore, we will discuss the ethical issues arising when developing a dynamic analysis system

[61] JSON (JavaScript Object Notation) is a lightweight data-interchange format. (https://www.json.org/json-en.html)

[62] https://releases.ubuntu.com/focal/

[63] https://www.microsoft.com/en-us/software-download/windows10ISO

and how they were mitigated in the design.

5.2.1 Installation

The VMs are created and run with QEMU[64], which is a full-system emulator. We have installed the operating system on the VM and set it up for dynamic analysis. The idea is that we create a basis VM state that we can revert to after each individual analysis, such that we do not have to recreate the complete VM again. With QEMU this is possible using the snapshots tool, which will store the current VM state and allow you to go back to it whenever required. The main steps of the installation process are: **(1)** creating the VM and installing the operating system, **(2)** installing and configuring SSH (OpenSSH[65]), **(3)** installing required software and dynamic analysis tools, **(4)** disabling any security measures that can interfere with or prevent the execution of malware. After these steps a sandbox state to revert to after for each analysis should be achieved. The first step of installing the operating system tends to be difficult to automate, which is why this is done manually in our case. The cause of complexities is that the installation software is not very robust for non-graphic installations and without it, it is hard to automate. This is the reason for manually installing the operating systems. We used a VNC[66] to do this in our case, since the VM is not on any local computer we can access directly. After installing the operating system, we also manually installed and configured the OpenSSH tool for SSH, this will allow us to automate further steps and the dynamic analysis process, as this enables automated communication from the host machine to the sandbox. Step 3, installing software and dynamic analysis tools on the sandbox, has been fully automated with a Shell script. This Shell script copies the ABE and necessary scripts, such as an "installer" script, to the sandbox using SCP[67]. It then tells the sandbox to run the "installer" script, which will install all the required tools and set up the necessary environment variables. This will be done using APT[68]. The versions of the tools are important, because they dependent on each other. The software installed and their corresponding version can be seen in table 10.

After the installation of the tools we move on to disabling security measures that are implement in the operating software by default. It is important to disable these as they can interfere with the behaviour of malware, possibly blocking certain paths of execution that show malicious behaviour or even preventing the execution of the malware. In a default user system this would be wanted, however in the case of dynamic analysis you want to see the malicious activity of the malware to gain insights on its behaviour and workings. Ubuntu does not have any hard security measures that can interfere with running malware, apart from user privileges. We can run the PoCs as the root user to give the PoCs all privileges to ensure it will take the execution path that it is intended to take. For other operating systems this step can look very different. For example, when we did the installation process for windows we had to disable Windows Defender, Windows Firewall, BitLocker, User Account Control (UAC), and Windows Update. Windows Defender was the most difficult to disable, as this will still run in the background even when you "disabled it" in the Windows

[64]QEMU (Quick Emulator) is a full-system emulator https://www.qemu.org/

[65]OpenSSH is a tool for remote login with the SSH protocol. https://www.openssh.com/

[66]Virtual Network Computing (VNC) is a graphical desktop-sharing system that allows you to access an other desktop graphically over the network.

[67]Secure Copy Protcol is a tool for transferring files from local host to a remote host using the SSH protocol.

[68]Advanced Package Tool is the main package management tool for Ubuntu and Debian-based systems. https://wiki.debian.org/Apt

Software	Version
Open-JDK	11.0.20
Maven	3.6.3
Ant	1.10.7
Gradle	7.4.2
GNU Make	4.2.1
Python3	3.8.10
Python2	2.7.18
Fakenet-Ng	3.0
ProcMon	1.0.1
SysMon	1.2.0

Table 10: The software installed along with their corresponding version.

settings. To completely disable Windows Defender, we have removed it using the Defender Remover tool developed by ionuttbara[69]. After this last step the sandbox is ready for dynamic analysis. This sandbox state will be stored and will serve as the basis for the dynamic analyses. In our installation process we have also stored the state of the sandbox after each main step as it is good practice to do for when something goes wrong or when changes are required.

5.2.2 Automated Dynamic Analysis

The dynamic analysis process is fully automated with a Shell script and the ABE. In total, an individual dynamic analysis will consist of a sequence of three runs of the sandbox that is used (Windows or Ubuntu). With a "run" we mean the session of the sandbox between booting up and shutting down. This means an individual analysis will boot up and shut down the sandbox that is used three times. The process of these runs are presented in figure 10. The first run of the sandbox will be for building the PoC. In the second run we try to find commands with valid executions. And in the third run we run the found commands while monitoring the activity. Everything is fully automated in this process and logs, including monitoring reports, are retrieved from the sandbox after each run to the host machine. We will discuss each run in more detail in the following subsections.

5.2.2.1 Run 1 - Builder

In the first run of the sandbox the PoC under analysis will be copied to the sandbox and the Builder component of the ABE (see section 5.1.1) will be run on it to find default build commands that worked. Internet is required in this run of the sandbox such that the build automation tools can download necessary dependencies. Even though, default build commands should not be harmful, it remains a risk. We do restrict the network of the sandbox in the second and third run, which involves running the actual commands of the PoCs. This is why running the Builder component is separated from running the other components of the ABE. The state of the built PoC after running the Builder component in run 1 will be stored temporarily for each individual analysis. This state is used as the start for run 2 and run 3, such that network can be restricted when running the

[69] https://github.com/ionuttbara/windows-defender-remover/

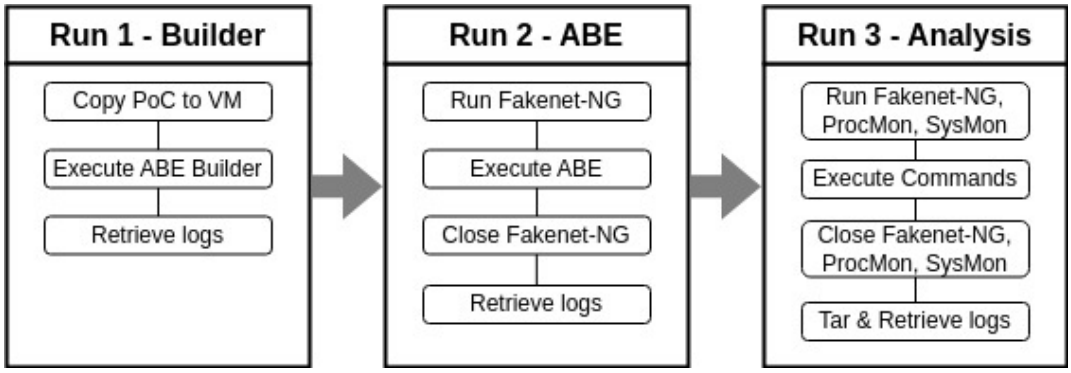

Figure 10: Dynamic Analysis Sandbox Runs.

executables obtained after building the PoC. This implementation isolates the PoCs as much as possible, as network is always restricted when running PoC specific commands.

5.2.2.2 Run 2 - ABE

In the second run, which runs the state of the built PoC after run 1 without internet access, the other three components of the ABE will be run to find commands with valid executions. A valid execution, as we discussed in section 5.1, is either an execution that returned a `returncode` of 0 or and execution that did not terminate for the time-out period, which by default is 30 seconds. For build commands a valid execution is only an execution that returned a `returncode` of 0, so if it timed out it does not count as valid. In this run the Readme Miner (section 5.1.2), Script Runner (section 5.1.3), and the Fuzzer (section 5.1.4) components will be run in this order. Internet access is restricted so build commands found by the Readme Miner or in scripts run by the Script Runner will not be able to download dependencies, therefore these are prone to fail if they require internet access. The same holds for PoC commands that require internet access, however to limit the failing of such commands we simulate a legitimate network on the sandbox using Fakenet-NG[70] to hopefully improve the execution path. It could be the case that the malware would crash if it notices that network is completely restricted, however by simulating a legitimate network where the malware can simply not get a connection going, it could potentially take an execution path that does not fail. This could therefore improve the coverage of the ABE. Apart from simulating a legitimate network, FakeNet-NG is a dynamic analysis tool for monitoring network traffic. It works by intercepting and redirecting network traffic, while simulating legitimate network services. This is one of the dynamic analysis tools we will also use in run 3 to monitor the activity of the PoC. In this run, it will only be used for simulating a legitimate network, however we do retrieve the monitoring logs that are created when capturing the network traffic while running the ABE. These monitoring logs will not be used, but, if curious, can still be analyzed. The ABE will export the results of commands that it found on the sandbox, which will then be retrieved to the host machine. The most important log file here is the JSON file, which will contain the found commands that had valid executions. The commands in this JSON file will be executed in run 3 and analyzed.

[70] https://github.com/mandiant/flare-fakenet-ng

5.2.2.3 Run 3 - Analysis

The third run will do perform the dynamic analysis on the commands that have been found during run 2. The run will start with the state of the built PoCs which was temporarily stored after the first run and run the found commands while dynamic analysis tools are monitoring the activity. Three dynamic analysis tools are used to monitor the activity of the PoCs: FakeNet-NG, Procmon (Process Monitor)[71], and Sysmon (System Monitor)[72]. FakeNet-NG has already been discussed in run 2 as it was used to simulate a legitimate network. In this run it will be used specifically to capture and analyze the network activity. Procmon will be used for monitoring the activity of processes on the sandbox, and Sysmon will be used to monitor system activity. Procmon and Sysmon are part of the Microsoft Windows Sysinternals Suite, which contain utilities for managing, troubleshooting and diagnosing Windows and Linux systems and applications. Procmon for Windows monitors the file system, Registry and process activity in real-time, and in Linux Procmon monitors the `syscall`[73] activity in real-time. Sysmon will monitor system events, such as process creations, network connections, file system writes, and process lifetime in both Windows and Linux. Before activating all the dynamic analysis tools, the JSON file containing all the found commands with valid executions obtained in run 2 will be copied to the sandbox. The execution of the commands will be done in the following sequence: **(1)** build commands, **(2)** Bash commands, **(3)** Python commands, **(4)** Java commands. The default build commands that worked will not be in this file as they were found in the first sandbox run, therefore it is also not needed to rerun it again as the starting state of run 3 is the ending state of run 1. This allows us to include builds that required internet access, while restricting the internet access in this sandbox run. The build commands in the JSON file will therefore only be build commands found in the Markdown files by the Readme Miner, which should not require internet as it was counted as valid when it was run in sandbox run 2, which also had restricted network. This order of commands execution is important as executables that require building are built before trying to run it. The running of these commands will be done by a script in the sandbox in the mentioned order. To prevent the running of commands for too long, this script has the same timeout implementation as the ABE (discussed in section 5.1.5). By default the build commands will run maximally 90 seconds, and the other commands will run maximally 30 seconds. During the running of the commands the dynamic analysis tools will monitor the activity of the executions and log this to their respective log files. Afterwards, all the log files are retrieved to the host machine and will be ready to be analyzed manually. Given the large sizes of these log files the logs will be compressed before transfer in this sandbox run. When the sandbox is closed the stored state of the built PoC of run 1 will be deleted and the sandbox state will be reverted to the basis state for the next dynamic analysis, and for security reasons, as we do not want to accidentally run the potentially infected sandbox again.

5.2.3 Setup

When performing malware analysis it is important to correctly isolate the execution of the malware and prevent any harm to production systems and infection of other systems. In our setup of the dynamic analysis system we have taken the dangers and ethical issues of running malware into

[71] https://learn.microsoft.com/en-us/sysinternals/downloads/procmon

[72] https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon

[73] syscall (system call) is a request to the operating system to securely perform a certain action.

account. The potential malware will only be run in an isolated and controlled virtual machine, that is separate from production systems and restricted from the network. This virtual machine, that is the sandbox, will only have communications with the host machine. Figure 11 presents the setup for the dynamic analysis system we created, which will is used for the experiment done in section 6. The host machine will be able to execute connect to the sandbox through SSH and execute commands, while the sandbox is unable to connect to the host machine. This one-directed SSH tunnel allows us to securely automate the dynamic analysis, as the sandbox remains isolated. This also allows the host machine to SCP files to and from the sandbox, while the sandbox cannot initiate SCP transfers.

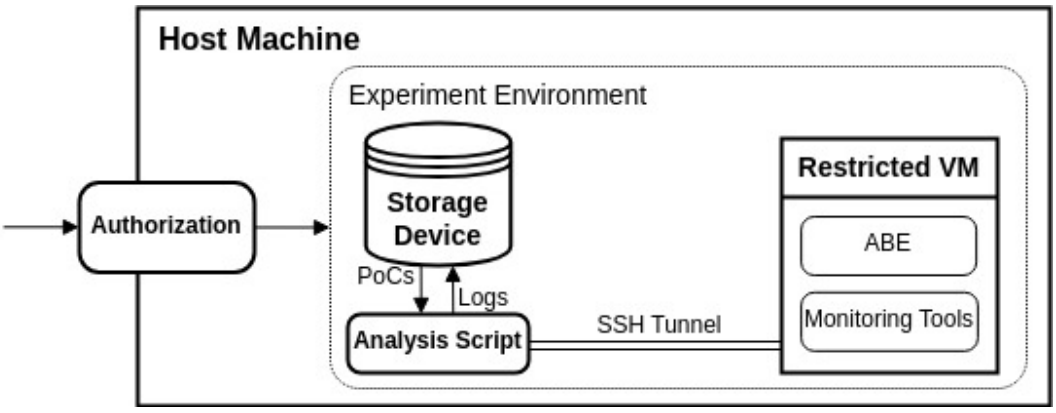

Figure 11: Analysis Setup.

Additionally, all the components of the host machine, including the server, network devices, and storage, are updated with the latest security patches, which will provide a robust and protected environment for conducting dynamic analyses. Furthermore, the access to the host machine is restricted to authorized researchers and personnel only, to reduce the risk of unauthorized use or data breaches. The host machine will contain the PoCs that will be analyzed on the sandbox, it is therefore paramount to restrict access to the machine as malware in the wrong hands can inflict a lot of harm. The PoCs will only be used for research purposes and handled strictly according to ethical guidelines. After performing analysis of each individual PoC on the sandbox the state will be immediately reverted to the basis state to prevent storing and accidentally opening an infected sandbox state. The logs and monitoring reports that are transferred to the host machine are specifically picked to ensure that they are from the ABE, the operating system logs, or from the dynamic analysis tools. Nothing else will be transferred to the host machine, and especially not any executables. The logs will only be stored on the host machine and will not contain any personal identifiable information (PII) or sensitive data that is not necessary for the research objectives. The sandbox will always be off when no analysis is being performed and it will not be running any longer than is necessary to prevent any mistakes with the handling of the sandbox and to keep the state of the sandbox known and controlled.

6 Experimental Results

We have done an experiment to test the performance of the automated dynamic analysis system and the Automated Black-box Executer. The main focus of this experiment is to show that automation of the dynamic analysis process can be achieved by leveraging the ABE that we have created. The ABE can be leveraged to automate the building of the PoCs and find commands to run them. To test the performance of the automated dynamic analysis system we therefore test the coverage of the ABE, as the more commands the ABE can find the more the dynamic analysis system can analyze automatically. We will test this by checking the coverage of the ABE of the complete dataset used for the data analysis in section 4.2. We then also show how the automated dynamic analysis system works by giving a couple examples.

6.1 Automated Black-box Executer Coverage

The complete dataset from section 4.1, which was also used in the data analysis (section 4.2), is ran with the ABE in the dynamic analysis system. This consists of only the first two runs of the sandbox as talked about in section 5.2.2. We have done this by implementing an option for only running the ABE, i.e. the first two runs, in the script that automates the dynamic analysis. The reason we are running the dataset with only the ABE is that we are mostly interested in the coverage of the ABE, given that this is the innovative tool we have created to enhance known standards for dynamic analysis and enable automation of dynamic analysis systems. Including the third run in this experiment would greatly increase the resources and time required for performing the experiment on the complete dataset of 1071 PoCs. We will therefore present the workings of the automated dynamic analysis system separately from this experiment in more detail on a couple randomly picked PoCs from the dataset. For this experiment of finding the coverage of the ABE on the dataset we have transferred the dataset to the host machine of the setup of the dynamic analysis system (see section 5.2.3) and performed the first two runs of the automated dynamic analysis, using a limit of 30 cycles for the Fuzzer, on each individual PoC repository in the dataset. The sandbox in the dynamic analysis process was given 8GB of RAM, 2 CPU's and 2 CPU threads. It was run with KVM[74] mode on, which should improve the performance of the VM. The experiment has been performed in 5 separate sequential runs on subsets based on the year of the PoC as classified in the study that obtained the dataset[YTG23] (see section 4.1). The years spanned from 2017 to 2021, with in total 1127 PoC repositories. The year 2020 contained nine PoC repositories with a size above 1GB, which we did in a separate run. The experiment ran in total for 85 hours and 46 minutes, which is 4.57 minutes per PoC repository. When we exclude the PoC repositories larger than 1GB, i.e. the second run of the 2020 subset which ran for 17 hours and 46 minutes, we get an average time of 3.64 minutes per PoC repository. The repositories in the dataset contained some duplicates that we filtered out after doing the experiment, which lead to in total 1071 distinct PoC repositories. These duplicates were not a mistake as the dataset of the previous study was counting PoCs and not repositories, therefore repositories which contained multiple PoCs from different years occurred in multiple year subsets. After filtering these out, we combined all the results of running the ABE, which included the exported CSV and JSON files

[74]KVM (Kernel-based Virtual Machine) is open-source virtualization software that turns a Linux Kernel into a hypervisor, allowing to run multiple VMs and can improving performance of VMs. https://www.linux-kvm.org/page/Main_Page

containing the found commands, into one CSV and one JSON file for the whole dataset. Table 11 presents the results of running the ABE on the entire dataset in the dynamic analysis system.

Commands	Commands found in dataset (distribution)	Repositories covered (pct. of dataset)
build	557 (34.92%)	258 (24.09%)
Java	549 (34.42%)	225 (21.01%)
Python	231 (14.48%)	209 (19.51%)
Bash	258 (16.18%)	89 (8.31%)
Total	1595 (100%)	614 (57.33%)

Table 11: Commands found in the dataset by the ABE with the dynamic analysis system.

The first column in table 11 shows the total number of commands found by the ABE for each respective commands class. It also shows the total commands found and the distribution of the commands that are found. The second column shows the number of PoC repositories with a command from the respective class, and in total the number of repositories in the dataset for which the ABE has found at least one command for one of the classes. In parentheses, the percentage of the total dataset of 1071 PoC repositories is given. We can see that 1595 commands are found in total, and 614 PoC repositories have at least a command found by the ABE, which is 57.33% of the dataset. Build commands are found the most, with 557 build commands found and 24.09% PoC repositories of the dataset covered. We then see that the ABE has found 549 Java commands that resulted in a valid execution, which means that it had a `returncode` of 0 or a time-out, i.e. no crash for the time period of 30 seconds. The ABE covered 21.01% of PoC repositories of the analyzed dataset with a Java command. Following this, Python and Bash commands have also been found, namely 231 Python commands and 258 Bash commands. Although the Bash commands have a higher presence, it covered only 8.31% of the dataset, while Python covered 19.51%. This suggests that the found Bash commands are less distributed over the PoC repositories. To get more insight on the coverage of the ABE components we have obtained the component specific results. Table 12 presents these results.

ABE Component	Commands found in dataset (distribution)	Repositories covered (pct. of dataset)
Builder	515 (32.29%)	249 (23.25%)
Readme Miner	159 (9.97%)	98 (9.15%)
Script Runner	455 (28.53%)	252 (23.53%)
Fuzzer	466 (29.22%)	214 (19.98%)
Total	1595 (100%)	614 (57.33%)

Table 12: Commands found in the dataset by each component of the ABE with the dynamic analysis system.

Table 12 shows the number of commands found and the number of PoC repositories covered in the dataset by each component of the ABE. The first column shows the total number of commands found in the dataset by the respective ABE component, along with the corresponding distribution. The second column shows the number of repositories with a command found by the respective

ABE component and the percentage of this from the complete dataset. We see that the Builder has found most of the commands, with 515 commands found. It covered 23.25% of the PoC repositories in the dataset. The Fuzzer has found 466 commands, which is 29.22% of the total commands found by the ABE and covers 19.98% of the dataset. The Script Runner has found 455 commands and covered 23.53% of the dataset, while the Readme Miner has found 159 commands, which is only 9.97% of the commands found by the ABE.

6.2 Testing the Automated Dynamic Analysis System

We tested the automated dynamic analysis on ten PoC repositories from the dataset from section 4.1. The subset is randomly obtained by drawing two random PoC repositories from each year of the dataset. The PoC repositories in the dataset were categorized based on the year the CVE that the PoCs targeted was found in, by the study[YTG23] that obtained the dataset. Some PoC repositories in the dataset had multiple PoCs targeting multiple CVE's, therefore it occurred in multiple years. To keep complete randomness we first randomly chose to which year the PoC repository belonged to. We then randomly picked two PoC repositories from each "year" subset to get ten PoC repositories. The ten PoC repositories were sequentially automatically analyzed by the dynamic analysis system. This time we also ran the third run of the sandbox, in contrast with the coverage experiment, such that we obtain the monitoring reports. This took in total 53 minutes. In the logs and monitoring reports stored on the host machine we can then analyze if the PoCs had any suspicious activity. We will first show the logs of the ABE and builder phase, and then delve into the monitoring reports obtained from the three monitoring tools: Fakenet-NG, ProcMon, and SysMon.

6.2.1 ABE and Builder Logs

The automated dynamic analysis system will create a directory for each individual analysis, thus for each PoC repository, with, by default, the same name as the PoC repository. In this directory there are three sub directories: "abe", "analysis", and "builder". There is also a file called `qemu.log`, which shows the output of running the VM with QEMU. The "builder" directory contains the logs from the Builder component of the ABE, which consists of a `builder-results.csv`, `builder.log`, and a `commands.json`. The log file contains the logging output of running the Builder component of the ABE. By giving the option "-d" or "–debug" to the ABE, informative debug logging will be done next to the normal logging. This can be handy for checking what the ABE is doing, such as seeing what commands it is trying to run and arguments the Fuzzer is trying. The `builder-results.csv` and `commands.json` files contain the commands that the ABE has found. In this case only the Builder component is run, so these files contain only build commands. The CSV file shows which command is found by which component and the JSON file lists all the commands found per category of build commands, Java commands, Python commands, and Bash commands. An example of the JSON file, obtained from one of the ten tested PoC repositories, is presented in figure 12. The "abe" directory will show these files too, but for the other components that were run in sandbox run 2. Additionally to these files, there will also be Fakenet-NG logs. These logs contain the output of running Fakenet-NG and there will also be a PCAP[75] file that Fakenet-NG exported. The PCAP file is the monitoring report that Fakenet-NG will generate when run and can be used to analyze

[75]A PCAP (Packet Capture) file contains network traffic data.

the network traffic that Fakenet-NG has captured. We will not analyze it in this case, because in sandbox run 2 it is run for the purpose of simulating network and not analyzing the workings of the ABE. However, it can be analyzed if interested. Nevertheless, we will analyze the PCAP generated in sandbox run 3 to analyze the network traffic of the executed commands.


```
[
  {
    "path": "/home/sandbox/b-abderrahmane_CVE-2021-44228-playground",
    "commands": {
      "java_cmds": [],
      "python_cmds": [
        [
          "python",
          "/home/sandbox/b-abderrahmane_CVE-2021-44228-playground/log4shell_validator.py",
          "--help"
        ],
        [
          "python3",
          "/home/sandbox/b-abderrahmane_CVE-2021-44228-playground/colored_formatted.py"
        ],
        [
          "python3",
          "/home/sandbox/b-abderrahmane_CVE-2021-44228-playground/constants.py"
        ]
      ],
      "bash_cmds": [
        [
          "bash",
          "/home/sandbox/b-abderrahmane_CVE-2021-44228-playground/exploits.sh"
        ],
        [
          "bash",
          "/home/sandbox/b-abderrahmane_CVE-2021-44228-playground/run_containers.sh"
        ]
      ],
      "build_cmds": []
    }
  }
]
```


Figure 12: Example of the JSON file exported by the ABE.

6.2.2 Analysis Logs

The *"analysis"* directory will contain the reports and logs obtained from the third sandbox run. There will also be a TAR[76] file containing all the *".log"* files from *"/var/log"* on the sandbox, which is where we also put our log files, along with the monitoring reports. The logs will therefore also include logs from the Ubuntu operating system. The monitoring file from Fakenet-NG has already been discussed in the previous section, section 6.2.1, which is a PCAP file containing network traffic data. The dynamic analysis tool ProcMon will export a database file containing data about all the `syscalls` made during its run. This trace file can be viewed within ProcMon's built-in terminal user interface (TUI), or with database software. Figure 13 presents an example of the trace file from one of the ten PoC repositories. The highlighted part is the `syscall` for running the commands runner program, which runs the found commands on the sandbox. In this example

[76] An archive file created with the tool TAR.

we can see that ProcMon captured 720050 events, and ProcMon shows all sorts of details, such as the timestamp, PID[77], Process name, etc... ProcMon can be overwhelming given the number of `syscalls`, therefore it is most useful when combined with the other dynamic analysis tools to know what you are looking for exactly.

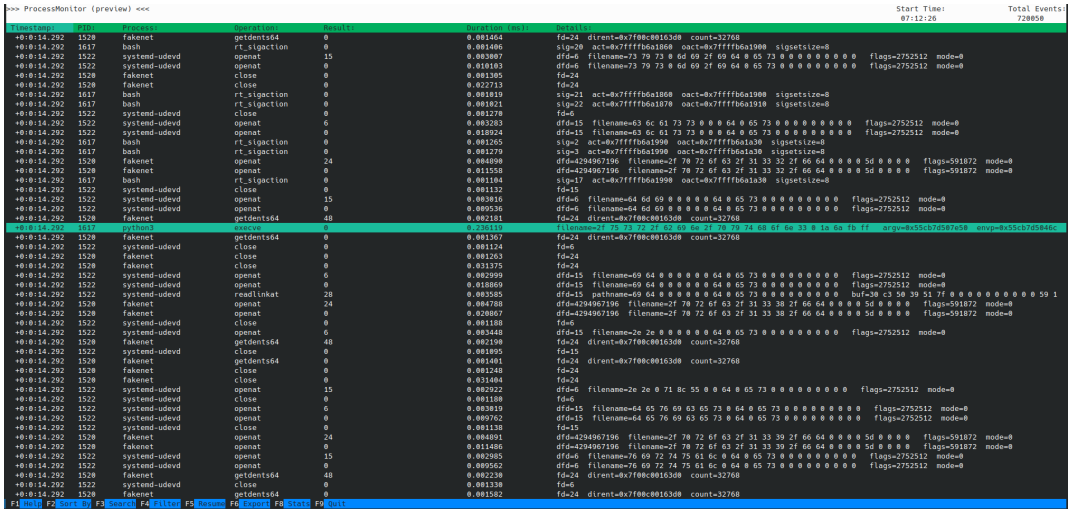

Figure 13: Example of the trace file exported by ProcMon.

SysMon helps with getting an oversight, as it logs system events. SysMon exports a file called "syslog" which can be viewed with the `SysmonLogView`[78] program that SysMon provides. An example of the logs from SysMon can be seen in figure 14. The example shows the logs for the execution of the commands runner program. Right after this we see that the commands runner program runs its first command, which is `''bash /home/sandbox/b-abderrahmane_CVE-2021-44228-playground/exploits.sh''`, judging from the parent PID of this event. Every PoC analysis from the dynamic analysis process will have the commands runner program run the commands found by the ABE, so this provides an easy way to find the PID of these commands and analyze them in SysMon and ProcMon.
For example, we see that the PID for this Bash command is 1618. Right after the event creating this process we can see that it does a `curl`[79] command to "http://host.docker.internal:8080/", see figure 15. This should get caught by Fakenet-NG, so we can look at the PCAP and search for it. As can be seen in figure 16 where we viewed the PCAP file with Wireshark[80], there are DNS[81] requests to "host.docker.internal", and the port is unreachable. This is probably because we do not have Docker and the sequence of execution is off, since this script should likely be run after setting up the Docker. To get more information on this script we go to ProcMon and find the `syscalls` this PID is doing. We found that before executing `curl` it actually does "env" without any arguments, we think that this is likely debug code as it just prints the environment variables in `stdout`. We then see it performs the `curl` command and after a while it times out. From this we can derive that this script is trying to see if "http://host.docker.internal:8080/" is reachable to do something with

[77]PID (Process ID) is a unique number that identifies a process.
[78]https://github.com/Sysinternals/SysmonForLinux
[79]curl is a command line tool and library for transferring data with URLs. https://curl.se/
[80]Wireshark is a network protocol analyzer. https://www.wireshark.org/
[81]DNS (Domain Name System) translates domain names into IP addresses.

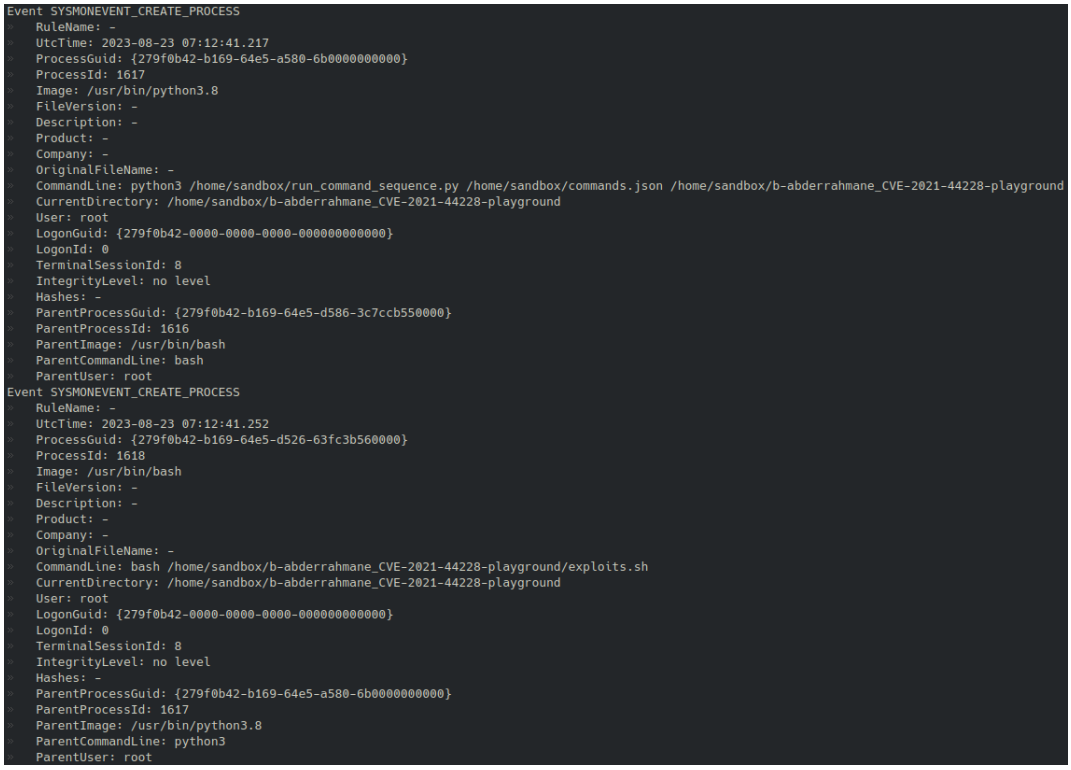

Figure 14: Example of the log file exported by SysMon.

it afterwards, such as, perhaps, sending the exploit to a Docker container, given the name of this script *"exploit.sh"*. This PoC repository contained four other commands found by the ABE that were analyzed. Two of those were Python files that immediately terminated after executing it. They were likely Python files included and used by other Python files given their names: *"constants.py"* and *"colored_formatted.py"*. An other Python file was executed with the argument "–help", and it likely just printed helper information and then returned with a `returncode` of 0. The last file was more interesting as it tried to run a Docker container and set up environment variables. However, we do not have Docker installed on the sandbox so this creation likely failed.

From the tested PoCs we have seen the execution of a build scripts that performed build commands and creations of JAR files in two PoC repositories. The commands were successful, but no JAR file execution commands were found by the ABE. In total the ABE did not find commands with a valid execution to analyze for four of the ten PoC repositories. There was one PoC repository that had some interesting activity in the monitoring files. The command that was executed for this PoC is `''java -jar /home/sandbox/Y4er_CVE-2020-2551/build/jar/weblogic_CVE_2020_2551.jar''`. `CVE-2020-2551`[82] is a vulnerability in the Oracle WebLogic Server. In SysMon, see figure 17 we found that right after running this PoC *"/usr/lib/policykit-1/polkitd"*[83] and *"/usr/lib/accountsservice/accounts-daemon"* are terminated, which are daemons[84] that handle user authorization and account management. Terminating these can lead to a loss of authorization control. In ProcMon we see that the PoC

[82] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2551

[83] https://linux.die.net/man/8/polkit

[84] A daemon is a program that runs in the background and typically play a role in managing the system.

```
Event SYSMONEVENT_CREATE_PROCESS
    RuleName: -
    UtcTime: 2023-08-23 07:12:41.274
    ProcessGuid: {279f0b42-b169-64e5-051a-090e6d550000}
    ProcessId: 1620
    Image: /usr/bin/curl
    FileVersion: -
    Description: -
    Product: -
    Company: -
    OriginalFileName: -
    CommandLine: curl --output /dev/null --silent --head --fail http://host.docker.internal:8080/
    CurrentDirectory: /home/sandbox/b-abderrahmane_CVE-2021-44228-playground
    User: root
    LogonGuid: {279f0b42-0000-0000-0000-000000000000}
    LogonId: 0
    TerminalSessionId: 8
    IntegrityLevel: no level
    Hashes: -
    ParentProcessGuid: {279f0b42-b169-64e5-d526-63fc3b560000}
    ParentProcessId: 1618
    ParentImage: /usr/bin/bash
    ParentCommandLine: bash
    ParentUser: root
```

Figure 15: Process PID 1618 performs a `curl` command.

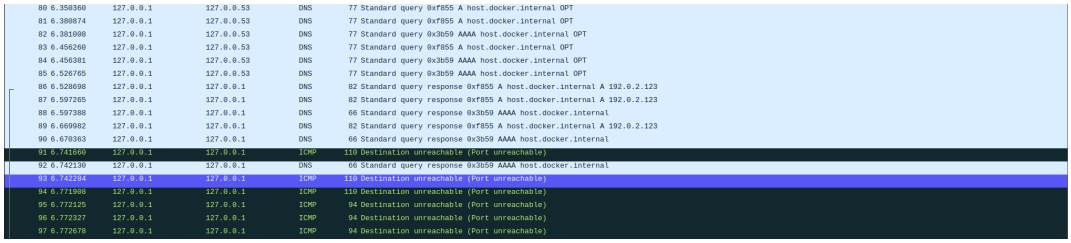

Figure 16: DNS requests to “host.docker.internal” in the PCAP file.

does a lot of `clone`[85] `syscalls` and does a core dump. In the logs of Fakenet-NG we did not see any network traffic from the PoC.

[85] https://linux.die.net/man/2/clone

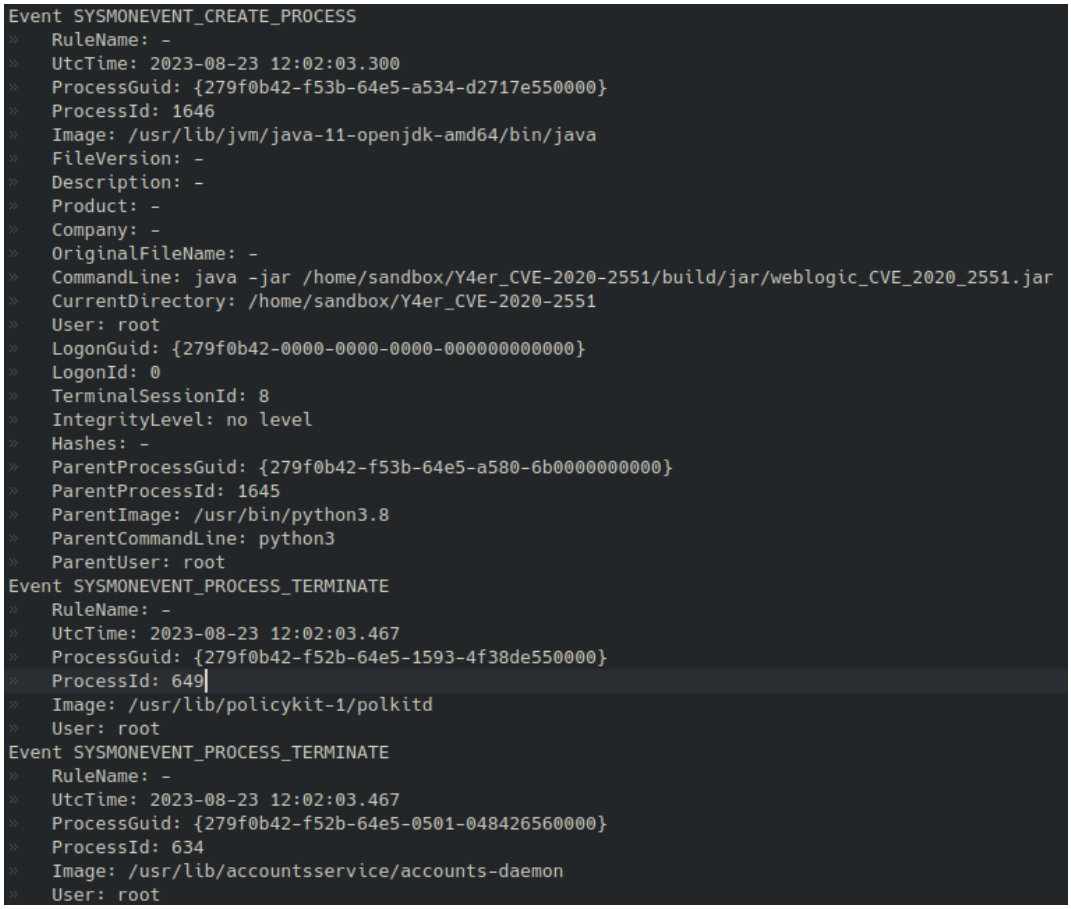

Figure 17: Executing the PoC and termination events of *"/usr/lib/policykit-1/polkitd"* and *"/usr/lib/accountsservice/accounts-daemon"* right after.

7 Discussion

This study has set the groundwork for automated dynamic analysis of PoCs. By using known techniques from automated black-box execution we have created a tool that performs automated black-box execution of Java PoCs. The findings of the experiments show that this tool can be used to automatically find the execution commands to automate the dynamic analysis process. In this section we will evaluate the dynamic analysis system we have created and the results from the experiment. We will show limitations in our design that can be improved for future work on automating the dynamic analysis process.

7.1 Evaluation

The automated dynamic analysis system consists of four components, where we leveraged the concepts of fuzzing and readme mining. We have seen that the Readme Miner covered 9.15% of PoC repositories of the tested dataset, which is much less than the 57% of found build commands in 1,500 Java projects from GitHub by Hassan and Wang[HW17] when automating Java project building using readme files. One explanation for this is that we restricted the internet access when running the Readme Miner, which probably reduces the amount of build commands found in the readme files that will work. In the data analysis we found that the dataset contained 195 repositories with a Markdown file containing the keywords for a build command in a code block, so this also suggests that the presence of build commands in the Markdown files in the dataset is low. We did, however, find that the other commands, such as Java and Python, are fairly present in the Markdown files of the dataset. This would indicate a high coverage for the Readme Miner, however the 9.15% contradicts this. It could be possible that these also require internet, although this would then also imply that the commands find by the Fuzzer and Script Runner should be low, but they are significantly higher with 466 and 455 commands found respectively. It is therefore very interesting that commands specified by the developers themselves in code blocks in Markdown files give less coverage than simply running the script without commands with the Script Runner and trying out random inputs with the Fuzzer. An explanation for this could be that the commands found in the Markdown files within a depth of 1 from the root directory is not covering enough of the Markdown files that contain useful commands.

In the coverage experiment, the Builder component of the ABE found 515 commands, which covered 249 (23.25%) PoC repositories of the dataset. In our data analysis we found that the build file for Maven was present in 525 (49.02%) of the dataset. The 515 build commands include the building of Java files with the Java compiler, therefore the number of build files from build automation tools covered is relatively low. This is in line with the findings of the studies, done by Sulír, et al. [SP16][SBM+20] where they tested the respective default build commands for Java projects. In the studies conducted in 2020 and 2016 they found that 59% and 38% of the analyzed dataset respectively failed their default build command. This shows that automatically building Java projects using the default build commands remains difficult.

In total the ABE has found commands for 614 PoC repositories in the dataset, which shows that it is able to find a command for 57.33% of the dataset. It could be that a repository only has a builder command, which is not so useful for the dynamic analysis system. By looking at the Java

commands found we can judge how well the ABE was able to find commands that execute the PoC. There were 225 PoC repositories with a Java command found, which means the ABE was able to find a command with a valid execution for 21.01% of the dataset. These are promising results that show the ABE can be useful for automating the dynamic analysis process. Furthermore, we have found that the Fuzzer is also a promising technique for performing automated black-box execution, as it found 466 commands and was responsible for 29.22% of all the commands found by the ABE. If we exclude build, the Fuzzer is responsible for finding 44.89% of the 1038 execution commands. This proves promising, especially given that our fuzzing technique is still fairly simplistic. If you look at the modern fuzzing techniques used in automated black-box testing[MPZ15], it is evident that this is only groundwork.

We saw that the experiment of the coverage of the dataset took in total 85 hours and 46 minutes, which is 4.57 minutes per PoC repository on average. This is a decent amount of time for the fact that the process consists of copying the PoC to the sandbox and logs from the sandbox, and starting up the sandbox two times. Note that we only ran the ABE with 30 cycles, so the time period could increase a lot when using hundreds or perhaps thousands of cycles as is done with Fuzzing in automated black-box testing. It also depends highly on the size of the PoC, as we saw that the nine PoCs with a size greater than 1GB ran for 17 hours and 46 minutes, which is 1.97 hours per PoC.

7.2 Limitations

Given that we are the firsts to automatically analyze Java PoCs and to create an automated black-box execution tool, limitations are of great importance to facilitate future work in this field. Although we have seen that Fuzzing is a promising technique for automatically finding commands and that automated black-box execution can be leveraged to perform automated dynamic analysis, there is room for a lot of improvement. We have seen that dynamic analysis is possible when we tested ten PoC repositories from the dataset, but we also saw that many of the commands that were executed were not useful or required a specific sequence to make it work, such as first starting up a server and doing something with it. Furthermore, our research is limited by not exploring PoCs that require deployment services, such as Docker. We have seen that a Dockerfile was present in 21.20% of the dataset, thus fairly present in the PoCs and definitely explorable in future work. An other type of PoC that we are not exploring with the dynamic analysis system are PoCs that require real-time inputs during execution via `stdin`. This would be an other step, as you have to both find inputs to execute the PoC and find inputs during the execution. Although the Fuzzer is promising, it is limited to the arguments specified in its arguments list. We used arguments that are often used in PoCs, but this still limits the input exploration. It is therefore recommended to explore other types of input generation, such as perhaps using Machine Learning, as is regularly done in fuzz testing.

An other limitation of our dynamic analysis process is that we have to use three sandbox runs per analysis, while you manually usually only do one. We did this to give the sandbox internet access for the Builder component of the ABE, and restrict the internet for the other runs. We think it is not possible to reduce the sandbox runs to a single run, given that you want to separate performing dynamic analysis and performing the ABE. If you would perform dynamic analysis along with the

ABE in a single sandbox run, the system would be infected before running the monitoring tools, which could affect the monitoring. And you do not want to start monitoring before the ABE, since the monitoring reports would than be very large and cluttered with executions that the ABE is trying. However, two sandbox runs seems the minimum. If one could find out how to run the ABE in one sandbox run securely it would reduce the number of sandbox runs to two and the amount of time an individual analysis takes. Along with this limitation of the sandbox runs, the internet connection in the first sandbox run also is a limitation, as it remains a risk to connect anything related to potential malware to the network. We only ran default build commands in this sandbox run, but it is still a risk. Although, this is also a risk when doing manual malware analysis, one has to be extra careful when doing it automatically as you have less control and oversight.

8 Conclusion

We have created an automated dynamic analysis system by creating a new tool that performs automated black-box execution through a form of fuzzing and readme mining. This tool, which we have named Automated Black-box Executer, automatically finds commands with valid executions for building and running Java PoCs. We leverage this to automate the dynamic analysis process, such that manual effort is reduced to only evaluating the monitoring reports generated by the automated dynamic analysis system. The approach of the Automated Black-box Executer is derived from analyzing a dataset of 1071 PoC repositories from GitHub and adapting techniques used in automated black-box testing. In the data analysis we were able to get a clear shape of the data in PoC repositories. We found a Markdown file in 90.48% of the dataset, which we analyzed further by analyzing the presence of command keywords in and outside code blocks. In these findings we found that the keyword "java" was present in 50.05% PoC repositories with a Markdown file. Further analysis showed that 79.64% of the examined keywords were inside a code block, which indicates that these are command instructions. We leveraged this by mining the Markdown files for potential commands given by the developer(s) of the PoC. Additionally, we used Fuzzing to try random inputs to found executables to find input combinations with valid executions. This technique is derived from techniques for automated black-box testing, where we randomly generate input combinations using feedback from previous runs. We tested the coverage of the Automated Black-box Executer by running it on the same dataset used for data analysis. It was able to find a command in 57.33% of the 1071 PoC repositories, with in total 1595 commands. Out of these, 549 were Java commands, which covered 21.01% of the dataset. The Fuzzer component was responsible for finding 44.89% of the execution commands. These are fairly good results given that the dynamic analysis system is limited by not implementing functionality for PoCs that require deployment, such as PoCs that use Docker which was found to be present in 21.20% of the dataset. We also tested the automated dynamic analysis system on ten random PoCs from the same dataset. We were able to get useful monitoring reports for command executions for six out of the ten repositories and analyze them. These findings show that the ABE is able to automate the dynamic analysis process and that Fuzzing is a promising technique in facilitating this. Future work should consider exploring other techniques for automated black-box execution, such as using Machine Learning for generating input combinations for the Fuzzer. With this research we lay the groundwork for automated dynamic analysis and automated black-box execution.

References

[AIB12] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[Bal99] Thoms Ball. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, page 216–234, Berlin, Heidelberg, 1999. Springer-Verlag.

[Bal05] Thomas Ball. A theory of predicate-complete test coverage and generation. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 1–22, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. 51(3), may 2018.

[BGZ17] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017*, pages 447—-450, United States, 2017. IEEE. MSR 2017 : 14th International Conference on Mining Software Repositories, MSR ; Conference date: 20-05-2017 Through 21-05-2017.

[BKM14] Clark Barrett, Daniel Kroening, and Thomas Melham. *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, June 2014.

[BPSZ08] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 143–157, 2008.

[BRN+09] Dominic Balasuriya, Nicky Ringland, Joel Nothman, Tara Murphy, and James R. Curran. Named entity recognition in Wikipedia. In *Proceedings of the 2009 Workshop on The People's Web Meets NLP: Collaboratively Constructed Semantic Resources (People's Web)*, pages 10–18, Suntec, Singapore, August 2009. Association for Computational Linguistics.

[CSA14] Nuno Ramos Carvalho, Alberto Simões, and José João Almeida. Dmoss: Open source software documentation assessment. *Comput. Sci. Inf. Syst.*, 11:1197–1207, 2014.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[GF16] Gillian J. Greene and Bernd Fischer. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 804–809, New York, NY, USA, 2016. Association for Computing Machinery.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[HMLW17] Foyzul Hassan, Shaikh Mostafa, Edmund S.L. Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47, 2017.

[HW17] Foyzul Hassan and Xiaoyin Wang. Mining readme files to support automatic building of java projects in software repositories. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 277–279, 2017.

[JNR+18] Sainadh Jamalpur, Yamini Sai Navya, Perla Raja, Gampala Tagore, and G. Rama Koteswara Rao. Dynamic malware analysis using cuckoo sandbox. In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 1056–1060, 2018.

[KLP17] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2511–2513, New York, NY, USA, 2017. Association for Computing Machinery.

[MPZ15] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. Chapter four - recent advances in automatic black-box testing. In Atif Memon, editor, *Advances in Computers*, volume 99 of *Advances in Computers*, pages 157–193. Elsevier, 2015.

[PLS19] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[PTT+18] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github README files. *CoRR*, abs/1802.06997, 2018.

[SBM+20] Matúš Sulír, Michaela Bačíková, Matej Madeja, Sergej Chodarev, and Ján Juhár. Large-scale dataset of local java software build results. *Data*, 5(3), 2020.

[SP16] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, nov 2016.

[VGT14] Mihai Vasilescu, Laura Gheorghe, and Nicolae Tapus. Practical malware analysis based on sandboxing. In *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, pages 1–6, 2014.

[YTG23] Soufian El Yadmani, Robin The, and Olga Gadyatskaya. Beyond the surface: Investigating malicious cve proof of concept exploits on github, 2023.

[YZC+17] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2139–2154, New York, NY, USA, 2017. Association for Computing Machinery.

[Zal16] Michał Zalewski. American fuzzy lop - whitepaper. 2016.

[ZWCX22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), sep 2022.