Opleiding Informatica

Automating Dynamic Analysis of Python
Proof of Concept Exploits

Daniel Kuiper

Supervisors:
Soufian El Yadmani & Dr. Olga Gadyatskaya

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 27/08/2024

Abstract

As reliance on complex digital system grows, ensuring their security has become increasingly crucial. Proof-of-concept (PoC) exploits are important for understanding and demonstrating software vulnerabilities within these systems. PoCs provide concrete examples that illustrate how vulnerabilities can be exploited, thereby facilitating better analysis and remediation. This study focuses on automating the dynamic analysis of Python PoC exploits. While PoCs are crucial for showing vulnerabilities, they pose risks if sourced from untrustworthy repositories. This research aims to improve the efficiency of analyzing these exploits through automation.

The methodology involves processing a dataset of 8,302 Python repositories from GitHub, using static and dynamic analysis techniques to investigate PoCs. The automated system resolves dependencies and generates arguments for dynamic analysis through fuzzing techniques.

For this study we developed Codarg, an automated system designed to generate argument vectors through static code analysis, with the aim to enhance effectiveness of dynamic analysis by maximizing code coverage. Our findings show that a good combination of static and dynamic analysis techniques can increase code coverage considerably. For example, when using exit status-driven arguments, the average coverage increased from 34.07% to 38.42%, and coverage-driven arguments further raised it to 40.23%. In scenarios where Codarg-generated arguments were used, the average code coverage improved from 22.63% to 30.75%.

This research demonstrates the potential for automated dynamic analysis to improve the security evaluation of PoCs. The results suggest that automated systems can play a valuable role in enhancing cybersecurity practices by providing efficient and reliable analysis of exploit code.

Contents

1 Introduction

Our society increasingly relies on complex cyber-physical and digital systems that are integral to critical infrastructure, including finance, healthcare, and transportation. However, these systems are prone to flaws that can be exploited by attackers. These flaws, known as vulnerabilities, are cataloged by the Common Vulnerabilities and Exposures (CVE)[1] system, managed by MITRE[2], which assigns unique identifiers (CVE ID) to them.

Proof-of-Concept (PoC) exploits are programs designed to demonstrate the potential of exploiting these vulnerabilities. Traditionally, PoCs have been used with a benign intent to better understand systems and improve security through penetration testing. Recently, however, there has been a concerning increase in malicious PoCs distributed to public platforms like GitHub[3] [YTG23]. This platform, unlike established exploit platforms like Exploit-DB[4] and the Metasploit Framework[5], does not have a submission process in place to validate the legitimacy of the PoC.

As the number of malicious PoCs grows, so does the challenge of identifying these threats. Dynamic analysis is a method for examining PoCs by running them and observing their behavior. This approach is particularly valuable as it provides insights into how PoCs behave under real-world conditions. Although techniques such as fuzzing and sandboxing are well-documented, their application to Python PoCs has so far been limited. Most studies focused on static analysis or dynamic analysis on binary executables, leaving a gap in the study of Python PoCs.

The aim of this research is to address this gap by automating the dynamic analysis of Python-based PoC exploits. Manual analysis of PoCs is time-consuming and prone to errors, making automation a crucial advancement. By automating the process, our research seeks to improve efficiency and reliability.

Dynamic analysis is most valuable when the PoC is executed as if it were a in real-world situation. This requires real-world input, often in the form of arguments. These inputs are crucial for triggering the intended execution paths and gathering meaningful data about the PoC's behavior. One of the key challenges in analyzing Python code is determining what arguments a program accepts, as the expected inputs can vary depending on the (type of) program. Programs typically accept a mix of positional and optional arguments, which complicates the analysis.

In this work, we present the design and implementation of Codarg, an automated system capable of generating argument vectors through static code analysis. Codarg is able to generate argument vectors including optional arguments. This system has been validated experimentally on a representative sample of 369 Python repositories, demonstrating its effectiveness in improving the informativeness of dynamic analysis for Python-based PoCs.

[1] https://cve.org
[2] https://mitre.org
[3] https://github.com
[4] https://exploit-db.com
[5] https://metasploit.com

Additionally, techniques like feedback-driven fuzzing can complement Codarg by finding valid positional arguments. This is achieved through the generation of random inputs and observing how the program reacts, and refining these inputs to maximize code coverage.

Our research methodology involves setting up a virtualized environment to safely execute Python PoCs. We automate the process of generating input arguments, running the PoCs, and collecting behavioral data. Automating dynamic analysis reduces the workload on security research and increases the speed at which new PoCs can be analyzed. This study highlights the importance of automating dynamic analysis for Python PoCs and provides a framework for further research.

2 Background

This section explores the methods and tools for analyzing Python Proof-of-Concepts exploits. It addresses the Python language and the unique challenges it poses, such as managing dependencies. The discussion also covers both static and dynamic code analysis, highlighting how each approach identifies potential issues in code. Additionally, the chapter introduces fuzzing as a technique for discovering argument sets that maximize code coverage.

2.1 Python

Python is a versatile and widely used, interpreted language known for its simplicity and readability. Python's extensive standard library and the availability of numerous third-party packages make it a popular choice for a wide range of applications. However, analyzing Python programs presents unique challenges, not only due to its dynamic nature [HH09], but also due to differences between Python versions. Dynamic analyses in particular can be further complicated by the need to accommodate dependency management.

2.1.1 Differences Between Python 2 and 3

Python 2 and Python 3 incompatibilities are well-documented and have been a common challenge in the Python community since Python 3 was introduced in 2008 [vR24]. Apart from the transition from the `print` statement to the `print()` function, Python 3 introduces changes in string handling, integer division, and module names that are not backward compatible with Python 2. This also means that tools designed for one version might struggle to interpret code written for another version. This is especially true for programs that utilize Python's built-in `ast`[6] module to parse an abstract syntax tree (AST). While Python 2 has been officially deprecated in 2020, many legacy projects still use it.

2.1.2 Dependencies

Managing dependencies is an important aspect of Python project development, often handled using `Pip`[7], the package installer included with Python. `Pip` installs dependencies in the environment it is called from, whether global or virtual. Tools like `Virtualenv`, `Pipenv`, and `Poetry` manage dependencies by keeping them separate from the global Python installation, which prevents version conflicts [May19]. `Conda` offers its own tools for managing environments and dependencies, supporting multiple versions of Python on the same system.

2.2 Static Analysis

Static analysis refers to the examination of code without executing it [GS15]. This type of analysis is often used to find potential vulnerabilities, coding errors, and adherence to coding standards, but also for detecting malware. Static analysis tools parse the code and create an abstract representation, which is then used to detect patterns that may indicate issues.

[6] https://docs.python.org/3/library/ast.html
[7] https://pip.pypa.io

2.2.1 Heuristic Scan

Heuristic scanning in static analysis involves using rule-based approaches to detect suspicious patterns [BMFT15]. These rules are often derived from known vulnerabilities and coding errors. Heuristic scans can identify potential zero-day vulnerabilities by looking for code patterns that are indicative of malicious behavior or poor coding practices. However, heuristic scans can sometimes produce false positives due to their reliance on pre-defined rules.

2.2.2 Semantic Scan

Semantic scanning looks at the meaning and flow of the code, not just the syntax [Hot05]. This analysis can detect issues that simple pattern matching might miss. It examines how different parts of the code relate to each other, like variable scope and function calls, to understand the program's logic. This helps find deeper problems like logic errors and security vulnerabilities that depend on context.

While static analysis can be powerful, it has limitations in identifying runtime behaviors that only manifest during executions. Techniques such as encryption, binary packing, and obfuscation are often used to evade detection during static malware analysis [YLAI17], making dynamic analysis essential to identify these issues.

2.3 Dynamic Analysis

Dynamic analysis involves examining the behavior of a program during its execution [Bal99]. This type of analysis is important for understanding how a program operates in real-world scenarios, as it can uncover issues that static analysis might miss. By running the program in a controlled environment, dynamic analysis tools can observe how the program interacts with system resources, handles inputs, and responds to different conditions. The program's behavior can be observed using two primary methods.

2.3.1 Debugger

A debugger is a dynamic analysis tool that enables automatic runtime observation of a program's behavior. Attaching a debugger to a program allows for step-by-step code execution, which enables the automatic inspection of variables, memory states, and control flow during runtime. This detailed examination helps identify and understand runtime behaviors and helps pinpoint the exact circumstances under which vulnerabilities are triggered. However, there are several disadvantages associated with using a debugger for automatic runtime analysis, including a critical one: some malware or malicious code can detect the presence of a debugger, after which it might alter its behavior or disable certain functionalities to evade detection.

2.3.2 Sandboxing

Sandboxing provides a controlled execution environment that is isolated from the rest of the system. This isolation ensures that any potentially harmful actions performed by the program do not affect the host system [RSRS24]. Sandboxing can be achieved by using operating system features such as chroot jails or containers, or by using virtualization software like VirtualBox[8] and QEMU[9], and is particularly useful for analyzing untrusted code, as it allows the program to run freely in a safe environment where its behavior can be monitored without risk. In the context of this thesis, automated sandboxing using virtualization enables the safe execution of Python Proof-of-Concept exploits while capturing detailed behavior data.

2.4 Fuzzing

Fuzzing is a technique mostly used to test the robustness of programs by providing randomly generated inputs [SGA07]. In the context of this research, fuzzing is used to discover valid argument sets that maximize code coverage. Feedback-driven fuzzing involves using execution time, exit codes, and coverage metrics to guide the input generation process [Zha22]. This approach attempts to select the most effective inputs, and thereby enhances dynamic analysis by covering more code paths and potentially exposing threats or suspicious behavior in the software.

2.4.1 Time-Based Feedback Loop

In feedback-driven fuzzing, execution time is used as a metric to prioritize inputs. Longer execution times often indicate more complex code paths being executed, which can lead to higher coverage and the discovery of more issues. By focusing on inputs that result in longer executions, the fuzzing process can more effectively explore the program's behavior.

2.4.2 Exit Code and Other Output

Exit codes and other outputs provide good feedback for fuzzing. A successful execution typically returns a status code of 0, while non-zero exit codes indicate errors or exceptional conditions. Analyzing these outputs helps identify which inputs lead to successful executions and which will trigger errors. This will help guide the fuzzing process to refine the inputs for better coverage.

2.4.3 Coverage Feedback Loop

Coverage-guided fuzzing uses coverage metrics to select inputs that execute previously unexplored code paths [NH19]. By focusing on inputs that maximize code coverage, this approach helps the analysis explore as much of the codebase as possible. This method is particularly effective for finding parts of the code that may be prone to errors or vulnerabilities.

[8] https://virtualbox.org

[9] QEMU (Quick-Emulator) is an open-source machine emulator and virtualizer. https://www.qemu.org

3 Related work

In this section, we look at past research that relates to our goal of automating the dynamic analysis process for Python Proof-of-Concepts. Our approach has been influenced by some of these works, and we will examine their contributions. We will focus on studies about malicious PoC analysis and automated input argument generation.

Our research builds on the findings of the "Investigating Malicious CVE Proof of Concept Exploits on GitHub" study by El Yadmani, The, and Gadyatskaya [YTG23]. Their study investigates GitHub-hosted PoC exploits for known vulnerabilities from 2017 to 2021. These Pocs were statically analyzed using a set of heuristics to identify malicious PoCs. Out of 47,285 repositories analyzed, 899 (1.9%) contain malicious code aimed at exfiltrating data or installing malware. Techniques such as base64 encoding and hexadecimal obfuscation were commonly used to conceal these payloads. These findings highlight a significant risk for the security community using GitHub-sourced PoCs and underscore the need for better detection methods and caution. Our research extends this work by focusing on the automation of dynamic analysis for Python-based PoCs. While their research primarily involved static analysis to understand the behavior and impact of the exploits, our approach seeks to streamline this process through automated dynamic analysis. This automation involves the use of code analysis to generate input argument vectors.

The automatic generation of arguments was influenced by the research on "Compositional Dynamic Test Generation" (SMART and DART methodologies) by Godefroid et al. [God07]. The principles of systematic input generation and dynamic testing described in this earlier research influenced our approach to generating and refining argument vectors. They focused on using dynamic test generation to explore program paths and use feedback to improve inputs. This is partly mirrored in our use of fuzzing and feedback loops to maximize code coverage. The idea of compositional testing in SMART, which summarizes function behavior for higher-level testing, influences our method of identifying entry points and generating arguments. This automated, iterative process reflects the dynamic test generation techniques from SMART and DART research.

With studies like one by Holkner and Harland [HH09] highlighting Python programs' frequent use of dynamic features, the need for effective dynamic analysis systems is underscored. Since, research has introduced various frameworks and tools designed to address the unique challenges posed by Python's dynamic nature. Examples include a benchmarking tool that tests dynamic linking and loading capabilities of Python programs [LAdS$^+$07] and a dynamic analysis framework that offers runtime analysis using event hooks [EP22]. Although these contributions provide a solid foundation for understanding Python's performance characteristics, they do not focus on dynamic analysis in the context of malware assessment.

Dynamic malware analysis, however, has been broadly studied. Recent advancements include a model that uses transformer architectures to enhances malware detection and classification [TDBR24], and a hybrid approach that combines static and dynamic analysis to improve ransomware detection [ASFG24]. Despite these advancements, a notable gap remains in the dynamic malware analysis of PoC exploits.

To our knowledge, Oei's research [Oei23] is the only study that addresses dynamic analysis in the context of PoC exploits, specifically focusing on analysis for Java-based PoCs. While Oei's approach shares the same overall goal as ours, there are significant differences. The most important being the type of PoC, which in their study required compilation before being executed. They also chose for a design that approached the PoCs as a black-box, leaving the inner workings unknown. While following a similar approach could be considered, Chaudhry et al. [CGP24] have demonstrated the effectiveness of utilizing Python source code analysis tools for automatic project installation. However, they do not extend to finding intended execution paths, which is an aspect our research seeks to advance.

4 Data analysis

4.1 Data set

The dataset we use in our research originates from the study by El Yadmani et al. [YTG23]. This extensive dataset includes a subset of 8,305 distinct Python repositories, with 8,302 currently accessible. The repositories are clustered by year from 2017 to 2021. These clusters overlap because they are based on the Common Vulnerabilities and Exposures (CVE) that the PoC exploits target. Some repositories contain multiple PoCs for CVEs from different years.

El Yadmani et al. focused on GitHub-hosted PoC exploits for known vulnerabilities, which includes interesting findings related to Python repositories. They found that Python has become the dominant programming language among hackers and exploit developers, primarily due to its extensive range of libraries that support both fundamental programming and hacking tasks. One notable example involved CVE-2019-0708[10] (BlueKeep), where a base64-encoded line in a Python script decoded to execute another script, leading to a malicious VBScript on Pastebin, which contained the Houdini worm malware.

4.2 Preliminary Data Analysis

As Python 2 and Python 3 share a lot of incompatibilities, we have analyzed the used Python versions per repository in order to identify the considerations for designing the automation. Our analysis of Python versions over the years shows a clear shift towards Python 3. In 2017, Python 2 was more common than Python 3, reflecting the longstanding use of Python 2 in many projects. By 2018, the number of Python 3 entries increased, showing a move towards the newer version. In 2019, Python 3 entries surpassed Python 2 for the first time, highlighting the community's effort to transition. This shift became more evident in 2020 and 2021, with Python 3 entries consistently outnumbering Python 2. This trend shows the migration from Python 2 to Python 3, likely driven by Python 2's end-of-life and the advantages of Python 3. We can conclude from this data that despite Python 3 has become the more dominant Python version, Python 2 still has a significant share in the PoCs, meaning our automation should work with both PoCs using Python 2 and Python 3.

Table 1: Repositories using Python 2 and Python 3

Year	**Python 2**	**Python 3**
2017	2326	833
2018	2119	1357
2019	1762	1585
2020	292	507
2021	249	715

In our analysis of Python environment managers used in repositories (Tables 2 and 3), we found

[10] https://www.cve.org/CVERecord?id=CVE-2019-0708

that the majority do not contain any known environment-specific files, which may indicate a reliance on system-wide packages or a lack of environment management. The most common manager is `venv`, a lightweight environment manager included with Python, reflecting its convenience and ease of use. `Pipenv` environments are used by a few repositories, which highlights its adoption for managing dependencies and environments together, though it is not widely popular. Similarly, `Poetry` environments are used by a small number of repositories, indicating some preference for its streamlined package management and environment setup.

Table 2: Python dependency manager usage

Environment manager	**Count**
`venv`	1632
`Poetry`	12
`Pipenv`	11
None / unknown	6896

Table 3: Requirement-specification availability

Requirement-file	**Count**	**Percentage**
`requirements.txt`	1632	19.08%
`Pipfile`	5	0.05%
`pyproject.toml`	4	0.04%
`requirements.txt` and `Pipfile`	6	0.07%
`requirements.txt` and `pyproject.toml`	8	0.09%
None / unknown	6896	80.64%

We found that `pip`'s `requirements.txt` is the most commonly used method for specifying dependencies in repositories, likely because `pip` is the default package installer included with Python. Other methods observed include `Pipfile`, used by `Pipenv`, which aims to improve dependency management but has not gained widespread popularity. `Pyproject.toml`, used by `Poetry`, simplifies package management and publishing but is also less commonly adopted. A few repositories use both `Pipfile` and `requirements.txt` or `pyproject.toml` and `requirements.txt`, which suggests a transition phase or a need for compatibility with multiple tools. For many repositories we were not able to detect the used requirements specification, which may indicate simple projects with no dependencies, or alternative dependency management practices.

In this preliminary data analysis, we also focused on argument parsing methods due to their important role in the context of PoC exploits. PoCs are designed to interact with specific systems or applications, often requiring inputs such as the location of the target, commands to execute, or necessary credentials. These inputs can be passed through command line arguments. By understanding what argument parsing methods are used, we can design a system that can efficiently generate the appropriate arguments for successful PoC executions. Parsers found in our dataset include `optparse`, `argparse`, `getopt`, `click`, `pwnargs` and `docopt`.

Table 4: Parsing methods per entry point

Method	**2017**	**2018**	**2019**	**2020**	**2021**	**Total**	**%**
`sys.argv`	880	1227	1941	1205	236	5489	36.75%
`argparse`	1530	1546	1220	467	432	5195	34.78%
`getopt`	386	16	14	20	10	446	2.99%
`optargs`	188	107	112	9	18	434	2.90%
`pwnargs`	45	0	0	0	0	45	0.30%
`click`	7	1	2	10	9	29	0.19%
`docopt`	3	5	2	0	0	10	0.07%
None / unknown	567	1133	876	377	329	3282	21.98%
Error	4	0	1	0	0	5	0.01%

`Optparse` is the older one, included in Python 2.3, and deprecated in Python 3.2. `Argparse`, introduced in Python 2.7 and 3.2, is now the standard. `Getopt`, available since early versions of Python, is deemed less user-friendly. `Click` and `docopt` are third-party libraries with powerful features but are less commonly used.

From Table 4, we see that `argparse` is the most popular parser over the years, likely because it is part of the standard Python library. The high count of entries categorized as ‘Unknown’ or ‘None’ suggests many entry points use custom or no parsing methods. Although `sys.argv` is not considered an argument parser, we still included it in our table because many scripts handle command-line arguments directly using `sys.argv` without any additional parsing library. Overall, the trend shows a strong preference for `argparse`. The errors included in the table are caused by Python files that could not be opened or read.

5 Infrastructure

5.1 Sandboxing

Running untrusted PoC exploits can compromise system integrity. To handle these risks and ensure a secure environment, we use QEMU along with the KVM[11] accelerator for guest virtualization. This combination allows QEMU to act as a Type 1 hypervisor, which provides near-native performance and improved security by running directly on the host machine's hardware. This setup isolates all PoC executions on a dedicated sandbox machine, which effectively quarantines any potential threats.

Additionally, this configuration enhances our automation capabilities through the QEMU Machine Protocol (QMP)[12]. With QMP, we can execute tasks such as controlling the power state, attaching or ejecting devices, and handling snapshots for both active and offline virtual machines. These tasks are essential for maintaining control over the virtualized environment and have it operate smoothly.

We establish communication with the guest machine using SSH[13] over a specified port. This secure channel allows for file transfers and remote command execution, which we need for deploying and executing PoC exploits in a controlled manner. For file transfers, we use `rsync`[14] over SSH. The PoC repositories include Git metadata (`.git` directories) that are irrelevant to our use case. These files can slow down transfers due to the overhead of transmitting many small files. Using `rsync` allows us to exclude these files, streamlining the process.

Snapshots are important for maintaining system integrity. A snapshot captures the exact state of the virtual machine at a specific point in time. This includes the contents of its memory and its disk- and device state. By creating snapshots, we can revert the virtual machine to a previous state if it becomes unstable or compromised. This way each PoC exploit can be analyzed in a clean environment, free from the influence of previous executions. We use automated scripts to streamline the creation and restoration of these snapshots. This automation ensures consistency and efficiency in operating the sandboxed environment.

These processes, as illustrated by figure 1, are integrated into a comprehensive shell script. The script orchestrates the execution of PoC exploits within our sandboxed environment. It systematically prepares the environment by setting up the necessary configurations, executes the PoC exploits, and logs the results. This methodical approach makes sure that each step is performed correctly and that we achieve a high level of control throughout the process. This detailed and structured procedure enables us to safely analyze potentially harmful exploits while we protect the integrity of our systems.

[11]KVM (Kernel-based Virtual Machine) is a virtualization module in the Linux kernel that allows the kernel to function as a hypervisor. https://linux-kvm.org

[12]QMP is a JSON-based protocol that enables machine-level communication with QEMU instances. https://wiki.qemu.org/Documentation/QMP

[13]SSH (Secure Shell) is a network protocol that provides secure, encrypted communication between a client and a server. https://openssh.com

[14]`rsync` is a utility for transferring and synchronizing files across systems, which can operate over SSH. https://rsync.samba.org

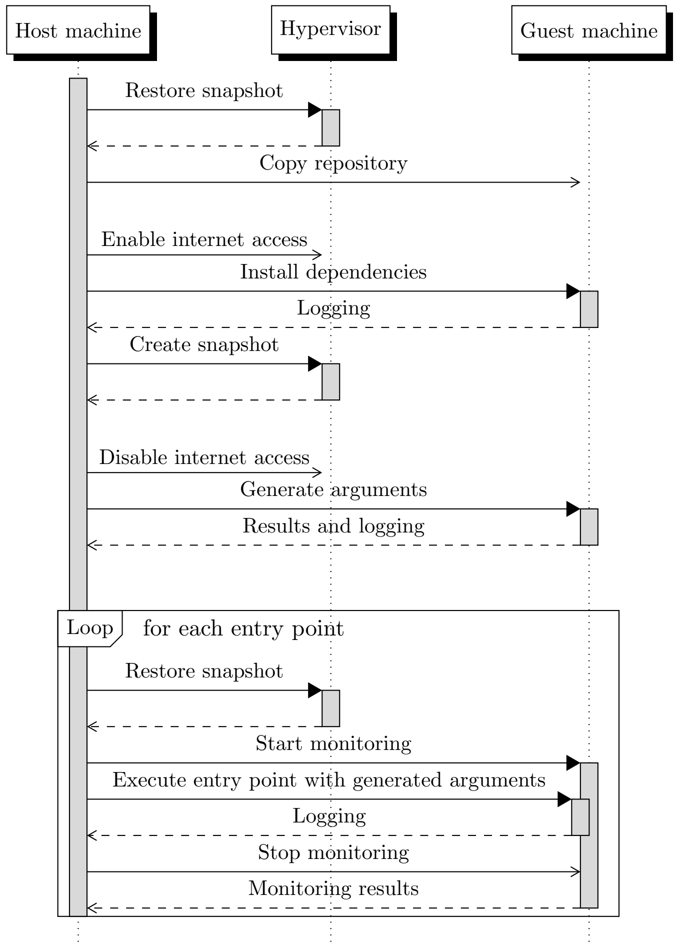


Figure 1: Automation sequence

5.2 Guest OS

In this research, we focus on automating a Linux guest machine. We selected Ubuntu 22.04.4 LTS server as the primary guest operating system due to its stability and compatibility with the automation tools required. The initialization process for the Linux guest machine begins with the manual installation of the operating system. We use VNC[15] for display access, which allows us to interact with the guest machine's graphical interface remotely. During this phase, we set up the filesystem and create user accounts. Next, we install and enable the OpenSSH[13] server to facilitate communication from the host machine to the guest machine. We configure OpenSSH to permit root logins and make sure that the SSH daemon starts automatically on boot. With SSH set up, the remaining steps can be completed using an automated script.

These steps involve updating and upgrading the system to ensure it is current and installing and configuring all necessary tools. Among these tools are FakeNet-NG[16], Procmon[17], and Sys-

[15]VNC (Virtual Network Computing) is a remote desktop protocol that provides a way to remotely control a computer's graphical desktop over a network.

[16]FakeNet-NG is a network simulation tool that mimics network services and records network activity.

[17]https://github.com/Sysinternals/ProcMon-for-Linux

mon[18], which will be detailed in section 6.4. We also install Miniconda, a minimal installer for `conda`[19]. Unlike Python-specific environment managers like `virtualenv` or `venv`, `conda` manages environments at the system level. This allows us to use different Python versions side-by-side, which is important because the PoCs in our dataset require different Python versions. We install both Python 2.7, the final major release of Python 2, and Python 3.11, released in 2022. This dual installation gives us compatibility with PoCs that span from 2017 to 2021.

After completing the setup and configuration of the guest machine, we create an initial snapshot to capture the machine's clean state. This snapshot acts as a baseline, which allows us to revert to a known, stable state before analyzing each PoC exploit. By reverting to this snapshot before each analysis, we make sure that every exploit is executed in a uniform and predictable environment, free from any prior modifications or contamination.

Although this research focuses on a Linux guest machine, the approach can also work with a Windows guest. This would involve installing Windows, setting up remote access, and configuring the monitoring tools[20]. While not within the scope of this research, this demonstrates the method's flexibility across different operating systems.

5.3 Ethical considerations

Our research involves running potentially harmful Proof-of-Concept (PoC) exploits. To handle these safely and ethically, we have adopted several practices to keep the experimental environment secure and well-managed.

We create isolated environments using QEMU with the KVM accelerator, which acts as a Type 1 hypervisor and operates directly on the host machine's hardware. This containment method helps keep any threats arising from our experiments confined to the virtual environment. We maintain security by regularly updating the host machine. This includes keeping both QEMU and KVM up to date with the latest security patches. While the sandbox operates on a fixed version, the containment measures prevent potential impacts from affecting the overall system.

Only authorized researchers can connect to the experimental environment via SSH. Access is restricted to the internal network or through a jump server, which reduces the risk of unauthorized use or data breaches. Our data does not include any confidential data. The data required for our experiments, such as the PoC repositories, is protected using the same access controls as the rest of the system. Since confidential data is not involved, additional data sanitization steps are not necessary. We handle PoC exploits ethically by using them solely for research purposes. Exploits remain on the experiment host machine and are only transmitted to the on-device sandbox environment for execution. This approach guarantees that exploits are not shared or used maliciously.

By following these practices, we make sure our research is both safe and ethical. These measures protect the integrity of our systems and ensure that our work adheres to high ethical standard.

[18] https://github.com/Sysinternals/SysmonForLinux

[19] `conda` is a cross-platform package and environment system that handles multiple languages. https://conda.io

[20] The selected tools have support for both Linux and Windows.

6 Automation

6.1 Requirements resolution

When performing dynamic analyses on potentially malicious PoCs, it is important that as much of the code is executed. By maximizing code coverage, we can explore as much detail as possible about how the PoC behaves, potentially uncovering hidden behaviors. This also means that it is important for the program not to stop prematurely due to missing dependencies. Typically, these dependencies are specified in files associated with a package manager.

In our Python repository dataset, the most prevalent dependency file is `requirements.txt`, which is commonly used with the `pip` package installer (see Table 3). Nonetheless, the same data shows that most repositories either lack a dependency specification or use a format that was not captured by our analysis. In this case we employ a methodology similar to that used by Chaudhry et al. [CGP24], which involves generating a `reqs.txt` file using the `Pipreqs`[21] module. `Pipreqs` is a popular Python package that automatically generates a `pip`-requirements file by analyzing the imports in the Python code. It is designed to work with both Python 2 and Python 3. However, issues can arise due to significant differences between the two Python versions.

If a Python 3 `Pipreqs` implementation tries to analyze a codebase written in Python 2, it might encounter syntax it doesn't understand. This can cause the analysis to fail because in this case `Pipreqs` expects Python 3 syntax. As a result, `Pipreqs` may produce errors or an incomplete requirements file, or no requirements file at all, because it can't correctly parse the Python 2 code.

To address this issue, we first determine the likely Python version used in the codebase. For this, we use a package called `Vermin`[22]. This package analyzes the source code for every Python file in a directory to determine the minimum Python version required by the codebase. It does this by parsing an abstract syntax tree (AST) and matching it to rules that correspond to specific Python versions. One limitation with this approach is that `Vermin` relies on Python's built-in syntax parser. This means that syntactical incompatibilities can cause `Vermin` to become unreliable. For example, using Python 2's print statement causes an invalid Python 3 AST (section 2.1.1). To minimize these issues, a pessimistic strategy is used in case of syntax errors: when `Vermin` encounters an invalid AST, it automatically assumes Python 2 is used. Another limitation is that some repositories use mixed Python versions, which, since we cannot detect project boundaries, causes Vermin to assume a version that fits neither.

Once a minimum Python version is determined, correct Python interpreters can be used to generate (if necessary) and install the `pip` requirements. As discussed in section 5, `conda` is used to configure environments for both Python 2 and Python 3. The binaries from these environments are in the following path: `path_to_miniconda/envs/<environment_name>/bin/python`. These can then be used to run the `Pipreqs` module (using `<python_path> -m pipreqs <path_to_repository>`) and `pip` module (using `<python_path> -m pip install`). Naturally, the same Python binary is used in later stages of the automation.

[21] https://github.com/bndr/pipreqs
[22] https://github.com/netromdk/vermin

Now that every repository includes a requirements specification, `pip` can be used to install them. When `pip`-install receives multiple package specifications, whether through command line arguments or via a requirements file, it aborts the entire installation if a single package fails to install. To circumvent this behavior, every package is installed in a separate `pip` install run. This ensures the most available dependencies, reducing the likelihood of a failing execution due to missing dependencies.

6.2 Entry point identification

Entry points represent the starting points for program execution. In Python projects, entry points are usually `.py` files intended to be run as scripts rather than imported as modules. Identifying these entry points is useful for several reasons.

To begin with, identifying the correct entry points ensures targeted execution. To analyze the behavior of a Python PoC, we need to execute the script as it would be in real-world usage. This makes sure that our analysis covers the actual functionality that users would execute. Additionally, entry points typically include code that processes command-line arguments. Identifying these scripts allows us to generate and provide correct arguments, which is the basis for thorough dynamic analysis as intended by this research. Furthermore, running the correct entry points prevents unnecessary execution of modules that are not meant to be executed directly. This saves a lot of time during the analysis process.

To identify entry points, we consider a `.py` file to be an entry point if it contains top-level code. In Python, "'top-level code' is the first user-specified Python module that starts running. It is 'top-level' because it imports all other modules that the program needs" [Pyt23] (see Listing 1).

The identification process begins with module import analysis. We first scan all Python files in the repository to identify import statements. By comparing each file's relative path with these import statements, we can determine which files are not imported by others and mark them as potential entry points. Next, we perform a 'main check' for each Python file, which means searching for the `__name__ == "__main__"` condition. Files containing this check are also marked as entry points, as this is a common Python indicator for top-level code. Finally, we combine the results from the module import analysis and the main check to create a comprehensive list of entry points. This combined approach makes sure that we accurately identify all relevant scripts for dynamic analysis.

6.3 Argument generation

When we run command-line programs, we deal with two types of arguments: positional and optional. Positional arguments depend on their place in the argument vector (argv). They are straightforward because we know which argument goes to which parameter based on their position. Optional arguments are different. They use a name preceded by one or two prefix-characters, like `--help` or `/HELP`, to indicate that the argument is optional.

Listing 1: Top-level program

```python
# module_example.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

```python
# main.py
import module_example

def main():
    print("This is the main program.")
    result = module_example.add(5, 3)
    print(f"The result of addition is: {result}")

""" the 'main check' """
if __name__ == "__main__":
    main()
```

In Python, we can access positional arguments directly using the `sys.argv` list. For optional arguments, we often use a dedicated parsing module. Table 4 shows the various argument processing methods used in Python repositories, detected through basic import string matching for each entry point. The most common methods are `argparse`, `sys.argv` and unknown/none. Since `argparse` and `sys.argv` are best suited for generating arguments, we will focus on these two.

Understanding the need for argument generation involves looking at how it enhances the code coverage in dynamic analysis. By inspecting the code to determine which positional and optional arguments a program expects, we can generate arguments that explore more parts of the code. This leads to better code coverage and a more informative execution. By including the process of identifying expected arguments as a step in the automation, we increase the likelihood of uncovering hidden threats and suspicious behavior.

6.3.1 Argument identification

When aiming for maximum code coverage, it is important to understand what argument sets would be considered valid. This is where Code Analysis Driven Argument Generation (Codarg) comes in. This method uses the parser's specification to look for common patterns and matches them to one of the argument categories as specified in Table 5. These categories and patterns were determined through manual inspection of the PoCs from the dataset. We rely on the AST to extract these patterns, focusing on the argument name, its destination, metavar, help string, and expected type as provided in `argparse`'s `add_argument`[23] method call. The argument name is the most important identifier, as it is the method's only required parameter. Destination and metavar are optional but can provide additional information about how the argument is used. The help string describes the argument in a human-readable format, which can be used to identify the argument's purpose. The expected type is the type that the argument should be converted to. This can be used to determine

[23] https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.add_argument

the argument's category, as different types often require different types of arguments. All these attributes are matched against the patterns in Table 5.

Table 5: Argument categories and patterns

Category	**Value**	**Patterns**			
		Names	**Substrings**	**Chars**	**Types**
port	[0,65535]	port		p	int, ord, float
IP address	127.0.0.1	hostname, ip	host	i	
url	127.0.0.1:[0,65535]	target, url		t, u	
credential	root	username, password	user, usr, pass, pwd		str, ascii
command	uname -a	command	cmd	c	
file path	file_[1,1000]	file, directory, input, output, path, payload, exploit, dump, exe, binary, cert, list	bin, dir	d, e, f, l, o	file, path, open

During the generation phase, argument vectors are built from the identified arguments in the entry point. This involves creating suitable values for each identified argument without executing the script. This phase follows a systematic approach to make sure the arguments are accurately represented and ready for dynamic analysis.

For categorized arguments, the same specifications used to identify them can be used to generate corresponding values. This process is done statically, meaning the script is not executed at this stage. Instead, argument values are constructed based on predefined rules.

Default values are used directly if provided in the entry point. This makes sure the argument behaves as expected in its default state. When the argument specification includes a set of choices, the first element from that set is selected. This guarantees the argument always has a valid value.

For arguments in a specific category without default values or choices, the category's generate function is used. This function produces suitable values that match the expected data type and usage context of the argument. For example, a category for port numbers generates a number within the range of valid port numbers.

If an argument does not match any predefined category, a generic placeholder value like `"value"` is used. This means no argument is left out, even if it falls outside the expected categories. Flag-type arguments, such as help (`--help`) and version (`--version`), are intentionally ignored during the generation phase. These flags usually stop the execution if the script after showing some information.

Arguments that are required or positional (i.e., those without prefix characters) are always included in the argument vector. This mimics real-world usage where these arguments are mandatory. When an argument takes multiple values (as indicated by the `nargs` parameter), the generation phase produces the required number of values. This might involve generating a list or sequence of values that match the expected format.

Since Codarg relies on parsing the AST, it is only compatible with entry points targeting Python 3. If the entry point uses an argument parser other than `argparse`, or if it targets a Python 2 version, a dynamic approach in the form of fuzzing is used instead.

6.3.2 Fuzzing

Feedback-driven fuzzing is used to dynamically construct an argument vector by iteratively generating arguments, as illustrated by Figure 2. In this method, per position in the vector, a value for each category (see Table 5) will be generated. These values are passed sequentially to the entry point. During this process, after every value has been evaluated, the feedback processor determines the most suitable argument for the current position. This decision-making process is guided by the feedback received from the entry point's execution. Based on this feedback, the fuzzer's control flow can be influenced in one of two ways:

- **Exit:** The fuzzer stops if the execution is successful or if no further improvements are possible (i.e. the argument set is deemed optimal).
- **Continue:** The fuzzer proceeds to the next value for evaluation if the current flow should be continued.

Once the optimal value is selected, the fuzzer iterates to the next position in the argument vector, continuing this systematic process. The simplified fuzzer used in this research aims for maximum code coverage, which deviates from the common objective of finding vulnerabilities or code flaws.

A crucial aspect of this approach is the execution of every entry point through the feedback processor's run command. This ensures that all information the feedback processor requires in its evaluation and decision-making is available for every run. We have implemented two feedback processors, each with its own run command and decision-making process. These will be detailed in the following sections.

Coverage feedback processor

The coverage feedback processor plays an important role in optimizing the dynamic analyses' effectiveness by gathering and enhancing runtime metrics to find the most suitable arguments for the argument vector. This process, known as coverage-guided feedback, relies on collecting detailed execution data to evaluate and refine the argument sets.

To ensure coverage metrics are collected reliably, all program executions are routed through

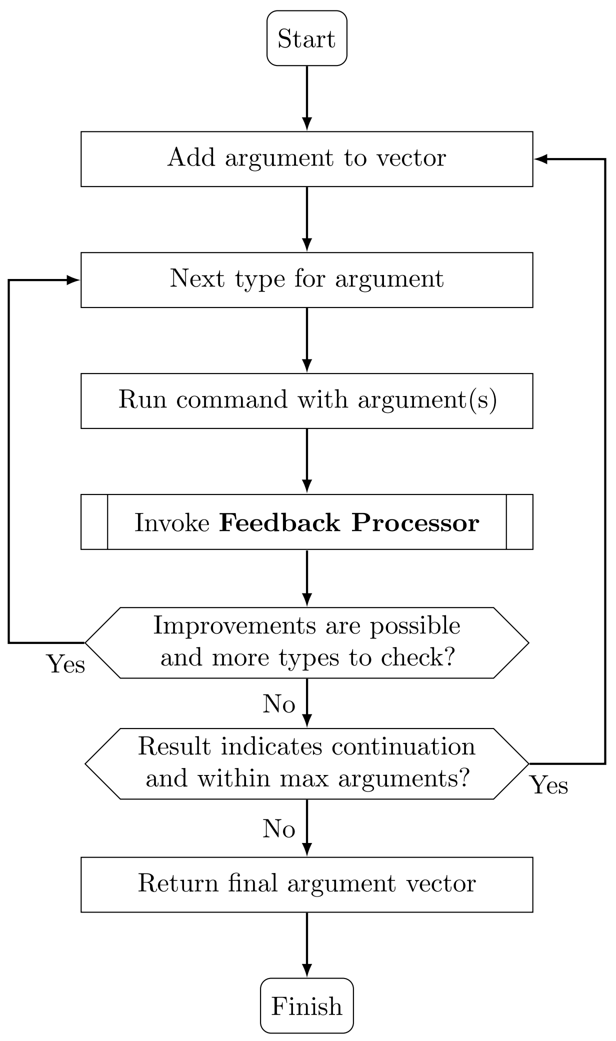

Start
Add argument to vector
Next type for argument
Run command with argument(s)
Invoke **Feedback Processor**
Improvements are possible and more types to check?
Yes
No
Result indicates continuation and within max arguments?
Yes
No
Return final argument vector
Finish

Figure 2: The fuzzer main loop

the `coverage.py`[24] module using the command `python -m coverage run [argv...]`. This approach guarantees that data on executed lines of code during runtime is accurately gathered. Specifically, the module tracks arrays of executable line numbers and executed line numbers for each module run. However, these raw metrics require further processing to be useful.

The key metric derived from the raw `coverage.py` data is code coverage, calculated as the ratio of executed lines to executable lines:

$$\text{Coverage} = \frac{\text{count}\,(\text{executed lines})}{\text{count}\,(\text{executable lines})}$$

This metric provides a straightforward measure of how much of the codebase is exercised by the current argument set.

Additionally, there was a consideration of a metric that evaluates the extent of code exploration by comparing the highest executed line number with the last executable line number in a file. However,

[24]https://github.com/nedbat/coveragepy

this metric was excluded since it mainly reflects the structural layout of the code rather than the depth of the execution. E.g. Listing 1 shows a Python program where the last line of code will always be executed regardless of the input arguments.

The coverage metric is averaged over the executed modules to obtain a general view of the program's execution under the current argument set. The feedback processor uses these averaged metrics to compare the current argument set with previous ones, determining suitability based on higher coverage.

This comparative analysis influences the fuzzer's control flow:

- **Exit:** If the predefined coverage threshold is reached, the fuzzer stops, indicating the current argument set is adequate for dynamic analysis.
- **Exit:** If adding an additional argument decreases coverage, the fuzzer should stop, because further arguments are most likely counterproductive.
- **Continue:** In all other cases, the fuzzer continues generating and testing new arguments to improve coverage.

By focusing on these metrics, the coverage feedback processor optimizes the argument sets used for dynamic analysis for maximum code coverage and deeper code paths, which helps uncovering potential threats and behavior anomalies more effectively.

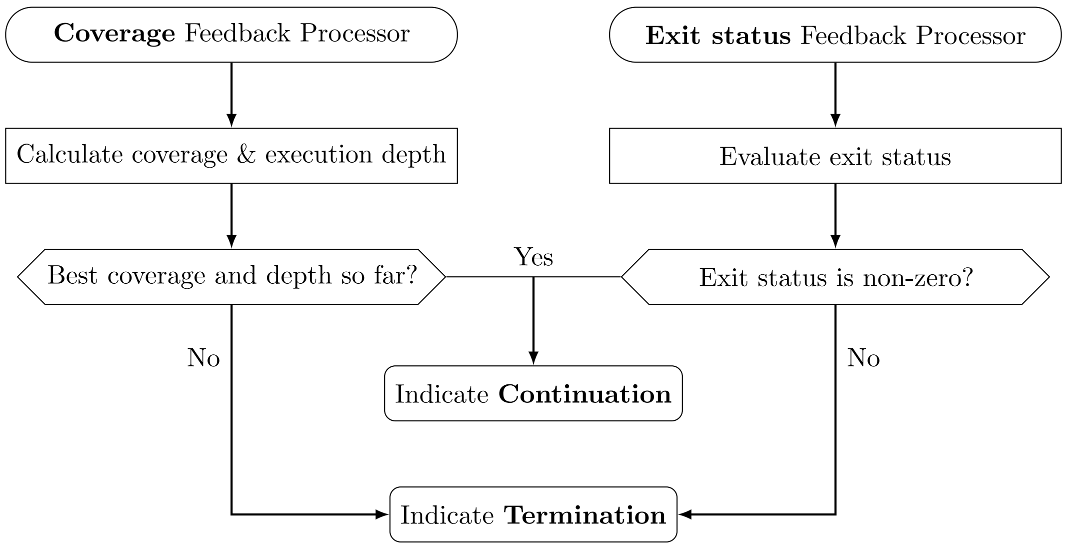

Coverage Feedback Processor
Exit status Feedback Processor
Calculate coverage & execution depth
Evaluate exit status
Best coverage and depth so far?
Yes
Exit status is non-zero?
No
No
Indicate **Continuation**
Indicate **Termination**

Figure 3: The feedback procedures

Exit status feedback processor

The exit status feedback processor utilizes another execution metric: the exit status of the program. This metric provides a direct indication of whether the program terminated successfully or encountered runtime failures.

Historically, the concept of exit status codes can be traced back to early UNIX systems [Ope14]. According to common conventions, particularly those outlined by the C standard library, a zero exit status indicates successful execution, while non-zero exit statuses signify runtime errors or failures. This simple interpretation makes exit status a reliable measure of execution effectiveness.

The exit status can be directly obtained from the subprocess result, so there's no need for specialized commands to retrieve it. This simplicity makes the exit status a fast and efficient feedback mechanism.

The control flow for the exit status feedback processor mirrors that of the coverage feedback processor:

- **Exit:** The fuzzer stops upon achieving a successful execution (zero exit status), thereby marking the current argument set as optimal since it cannot be improved further.
- **Continue:** The fuzzer continues generating and testing new arguments if the exit status is non-zero, which indicates runtime failures and the need for further optimization.

It is important to note that since there is no universal standard for exit codes, the results might vary and could potentially lead to inaccurate interpretations of the program's behavior. However, this approach does provide a pragmatic method for leveraging exit statuses to refine the argument sets and improve the effectiveness of dynamic analysis.

6.4 Execution

The execution phase is arguably the most important part of the dynamic analysis process. Here, the results from the argument generation phase are put to the test. For each entry point in the PoC repository the following steps and tools are used to execute Python exploit PoCs within the sandbox and capture the behavioral data for further analysis:

1. **Restore sandbox state**
   Before executing any entry point, we first restore the sandbox to a clean state. This involves reverting to the previously created snapshot where the repository and all necessary dependencies are already installed. This way each execution starts from a consistent, known state, which allows for reproducible results.
2. **Start Monitoring Tools**
   We use several monitoring tools to observe the behavior of the entry points during execution:
   (a) FakeNet-NG intercepts and redirects all or specific network traffic while simulating legitimate network services. It allows us to see how the entry points interact with network resources, which is important for understanding potential exploit paths.
   (b) Procmon traces system call activity, which provides insights into what the entry points are doing at a low level. For example, Procmon can show file access patterns, and other system interactions.

(c) Sysmon logs detailed information about process creation, network connections, file creation, and more. Sysmon's logs help us understand the lifecycle of processes spawned by the entry points and their interactions with the system.

These tools are started beforehand to make sure they capture all relevant data from the moment the PoC begins execution.

3. **Execute the PoC**
   We then execute the entry point of the PoC with the optimal argument set as determined during the argument generation phase. Each execution is subject to a timeout mechanism that will terminate the PoC execution to prevent infinite loops or excessively long runs. A beneficial side effect of this timeout is our granular control over the execution phase so we can make sure that each PoC is analyzed within a reasonable timeframe.

4. **Stop Monitoring Tools**
   After the PoC execution completes or the timeout is reached, we stop the monitoring tools. This step makes sure that all the data collected during the execution is finalized and saved. By capturing the initial and final states of the PoC run with each tool, we gather comprehensive logs and traces for optimal analysis later on.

5. **Data Collection and Transfer**
   The final step is transferring the collected data back to the host machine for later inspection. This transfer ensures that all execution data is securely stored and available for detailed analysis. The actual analysis of this data is beyond the scope of this research, but the collected information serves as a foundation for identifying and understanding the exploit's behavior.

7 Experimental Results

In this section, we outline the results of our experiments. We start by discussing how we chose and prepared our samples, including the methods we used to ensure they were representative. We then move on to describe the data collection process and the analysis techniques applied. We explain the sample size calculations and how we distributed samples across different groups. Additionally, we detail the setup and execution of our experiments, highlighting the tools and procedures used to gather data. Lastly, we present our findings, showing the capabilities of our automated dynamic analysis system through an example.

7.1 Sample selection and preparation

As mentioned in section 4, the repositories were preclustered and contained some duplicates. These duplicates were excluded from the experiment through deduplication. After this correction, the population size N was determined by subtracting the duplicates from the original dataset.

The sample size was determined using Cochran's formula: $n = \frac{Z^2 \cdot p(1-p)}{E^2}$ adjusted for finite population correction, which accounts for the size of the population using $n_{corr} = \frac{n}{n - \frac{n-1}{N}}$. We used $Z = 1,96$ (corresponding to a 95% confidence level), $p = 0.5$, and $E = 0.05$, where Z is the z-score, p is the estimated proportion of the population, E is the margin of error, and N is the population size (number of repositories after deduplication).

This results in a representative sample size of 369 repositories. These will be proportionally distributed over the clustered data. The sample size n_i for each cluster i was determined using the following formula: $n_i = \lceil \frac{S_i}{S_{total}} \times n_{corr} \rceil$, where S_i represents the size of cluster i, and S_{total} is the total size of all clusters (which is the same as N).

Table 6: Summary of Data Size and Deduplication Statistics by Year

Year	Initial Size	Duplicates	Size after Deduplication	Sample Size
2017	2446	183	2263	103
2018	2586	162	2424	110
2019	2224	84	2140	98
2020	563	35	528	25
2021	734	6	728	33
Total	8553	470	8083	369

7.2 Data Collection and Analysis

The experiment was conducted using QEMU with KVM enabled. The virtual machine was configured with 4GiB of memory and equipped with 4 CPU cores, each operating with a single thread. For execution parameters, each execution, including those in the fuzzing process, was limited to 4 seconds. The maximum number of arguments generated per entry point was set to 8. These settings minimize the experiment duration while preserving efficacy.

As part of the dynamic analysis process, the runtime execution was thoroughly logged using the tools listed in section 6.4. FakeNet-NG captured network traffic in PCAP[25], Procmon logged a system call trace, and Sysmon produced system event logs. We will explore these collected log files for one repository from the sample in section 7.3.

In addition, for each entry point per repository, a variety of data was systematically collected to evaluate code behavior and performance. This includes runtime code metrics both with and without dependencies installed, and detailed code coverage information from different argument generation approaches. These include coverage-driven fuzzing, exit status-driven fuzzing, Codarg, and the base case of no provided arguments. Exit-status results, the argument parser used, and a record of any exceptions encountered during execution were also documented.

The experiment took approximately 25 hours to complete, averaging about 4 minutes per repository. This duration includes the additional time it took to collect runtime code metrics, which was about 11 hours, or roughly 2 minutes per repository, meaning a typical run would take slightly more than 2 minutes per repository on average.

Out of the 369 repositories in our sample, spanning 541 entry points, 26 entry points exited before any code coverage could be measured. This is likely due to invalid syntax, which might have been caused by incorrect version determination.

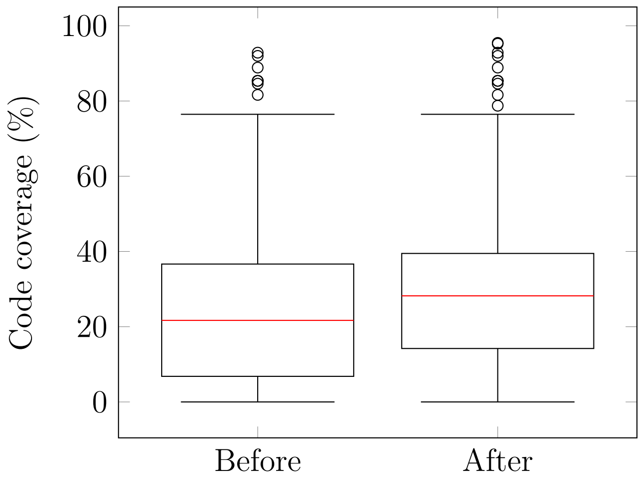


Figure 4: Code coverage before and after requirements resolution (with empty argument vector)

When looking at the code coverage gain from just installing dependencies, we found that the average coverage without dependencies, with no arguments is 24.76%. When dependencies are installed, the average coverage with no arguments increases to 29.32%. This results in an average coverage gain for dependency resolution of 4.56 percentage points (pp).

[25]PCAP (Packet Capture) is a file format for capturing network traffic data. https://tcpdump.org

The distribution of code coverage varies based on the generation method used and whether Codarg can be utilized. The sample will be split into two distinct groups that span the entire dataset. For the 423 entry points that are not supported by Codarg, the average code coverage without any arguments is 34.07%. When exit status-driven arguments are passed, the average increases to 38.42%. Using coverage-driven arguments, the average code coverage rises to 40.23%. There is no available data for Codarg arguments in this category.

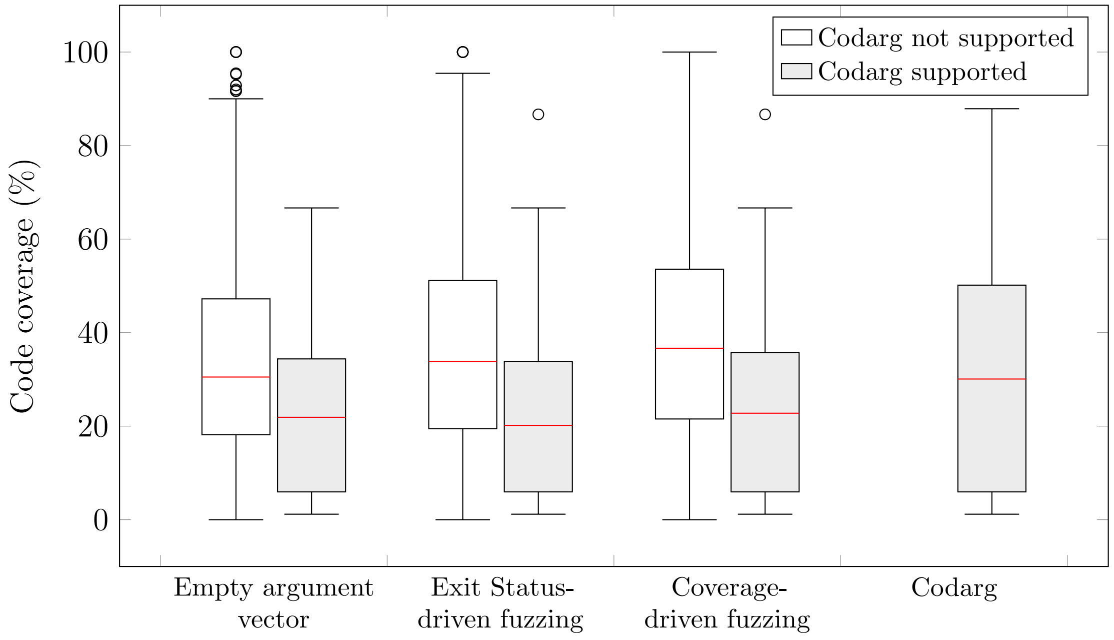


Figure 5: Coverage percentages by generation method per entry point

For the other 118 entry points that are supported by Codarg, the average code coverage without arguments is 22.63%. When using exit status-driven arguments, the code coverage rises to 22.93%. Coverage-driven arguments result in an average code coverage of 24.84%. With Codarg generated arguments, the average code coverage of 30.75% can be observed.

7.3 Running the automated dynamic analysis system

To provide a sense of the resulting data, we select one repository from the sample to illustrate how a single cycle from the automation might appear in a typical run. The PoC in this section targets CVE-2019-15107[26] which uses a command injection vulnerability for Webmin[27], a web-based admin panel.

After the repository is copied to the sandbox, requirement resolution is started. Logging (Listing 2) shows us that this repository did not specify its required dependencies, so a requirements file had to be generated. The resulting requirements from this `Pipreqs` invocation are installed next one-by-one to ensure maximum success.

[26] https://www.cve.org/CVERecord?id=CVE-2019-15107
[27] https://webmin.com

Listing 2: Requirements resolution logging

```
Creating reqs.txt file for poc
creating subprocess: ['/home/pocker/miniconda3/envs/py311/bin/pipreqs', 'poc']

[ truncated ]

subprocess: INFO: Successfully saved requirements file in poc/reqs.txt
creating subprocess: ['/home/pocker/miniconda3/envs/py311/bin/python', '-m', 'pip', 'install',
↪  'beautifulsoup4==4.12.3']

[ truncated ]

subprocess: Installing collected packages: soupsieve, beautifulsoup4
subprocess: Successfully installed beautifulsoup4-4.12.3 soupsieve-2.5
creating subprocess: ['/home/pocker/miniconda3/envs/py311/bin/python', '-m', 'pip', 'install', 'Requests==2.32.3']
subprocess: Requirement already satisfied: Requests==2.32.3 in ./miniconda3/envs/py311/lib/python3.11/site-packages
↪  (2.32.3)
```

Once the required packages are installed and a snapshot is created, it is time to generate an argument vector to use in the actual execution step. We can see in the logging for this step (Listing 3) that the entry point uses `argparse` for argument parsing, allowing the automation to use Codarg for argument generation. During the argument identification a match is found for `-host` and the arguments with default values are omitted.

Listing 3: Argument generation logging

```
name_or_flags: ['-host'], kwargs: {'help': 'Host to attack', 'required': True, 'metavar': 'IP'}
name_or_flags: ['-port'], kwargs: {'help': 'Port of the host ~ 10000 is Default', 'metavar': 'Port', 'default':
↪  '10000'}
name_or_flags: ['-cmd'], kwargs: {'help': 'Command to execute ~ id is Default', 'metavar': 'Command', 'default':
↪  'id'}
found keyword ip in IP
ignoring ['-port'] because it has a default value
ignoring ['-cmd'] because it has a default value
```

The generated argument vector will be used in the execution phase of the cycle. Listing 4 presents us with the process creation, along with all the output that the entry point prints to the screen. This shows the full argument vector and a truncated exception trace reporting that the attempted connection is refused.

This behavior can be observed in the data produced by both Sysmon (Listing 5), and FakeNet-NG. We use Wireshark[28] to examine the captured packets and filter out the noise provided by the SSH connection. The TCP[29] trace (Figure 6) reflects the request made by the entry point to localhost (127.0.0.1) on port 10000, showing how the attempt to initiate a secure connection (SYN packets) is refused (RST and SYN responses).

[28]Wireshark is a tool for capturing and analyzing network traffic. https://wireshark.org
[29]TCP (Transmission Control Protocol) ensures reliable and ordered data delivery over a network.

Listing 4: Execution logging

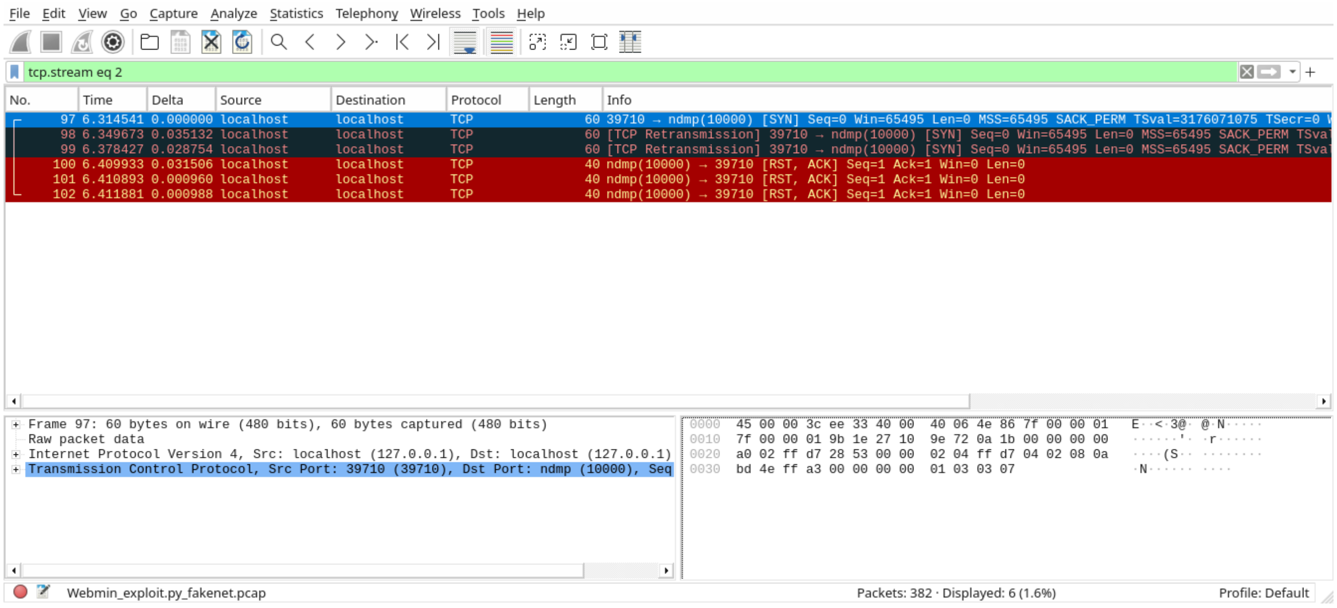
```
creating subprocess: ['/home/pocker/miniconda3/envs/py311/bin/python', 'poc/Webmin_exploit.py', '-host',
↪  '127.0.0.1']
subprocess:
subprocess:
subprocess:
subprocess:                  1.890 expired Remote Root
subprocess:
subprocess:                  By: n0obit4
subprocess:                  Github: https://github.com/n0obit4
subprocess: ----------------------------------------
subprocess: Traceback (most recent call last):

[ truncated ]

subprocess: requests.exceptions.ConnectionError: HTTPSConnectionPool(host='127.0.0.1', port=10000): Max retries
↪  exceeded with url: /password_change.cgi (Caused by NewConnectionError('<urllib3.connection.HTTPSConnection
↪  object at 0x7fd9a1e495d0>: Failed to establish a new connection: [Errno 111] Connection refused'))
```

Listing 5: Sysmon event logging (excerpt)

```
<Event>
    <System>...</System>
    <EventData>
        <Data Name="RuleName">-</Data>
        <Data Name="UtcTime">2024-07-29 08:49:36.755</Data>
        <Data Name="ProcessGuid">{4fb92b80-57a0-66a7-126a-650000000000}</Data>
        <Data Name="ProcessId">5659</Data>
        <Data Name="Image">/home/pocker/miniconda3/envs/py311/bin/python3.11</Data>
        <Data Name="FileVersion">-</Data>
        <Data Name="Description">-</Data>
        <Data Name="Product">-</Data>
        <Data Name="Company">-</Data>
        <Data Name="OriginalFileName">-</Data>
        <Data Name="CommandLine">/home/pocker/miniconda3/envs/py311/bin/python poc/Webmin_exploit.py -host
        ↪  127.0.0.1</Data>
        <Data Name="CurrentDirectory">/home/pocker</Data>
        <Data Name="User">pocker</Data>
        <Data Name="LogonGuid">{4fb92b80-0000-0000-e803-000000000000}</Data>
        <Data Name="LogonId">1000</Data>
        <Data Name="TerminalSessionId">29</Data>
        <Data Name="IntegrityLevel">no level</Data>
        <Data Name="Hashes">SHA256=52cd9b3c0d00de5c101eae7abcb66a5b35572e3d7bebd08b3a2e657e1b0ffb8f</Data>
        <Data Name="ParentProcessGuid">{4fb92b80-579f-66a7-126a-650000000000}</Data>
        <Data Name="ParentProcessId">5658</Data>
        <Data Name="ParentImage">/home/pocker/miniconda3/envs/py311/bin/python3.11</Data>
        <Data Name="ParentCommandLine">miniconda3/envs/py311/bin/python</Data>
        <Data Name="ParentUser">pocker</Data>
    </EventData>
</Event>
<Event>
    <System>...</System>
    <EventData>
        <Data Name="RuleName">-</Data>
        <Data Name="UtcTime">2024-07-29 08:49:37.458</Data>
        <Data Name="ProcessGuid">{4fb92b80-57a0-66a7-126a-650000000000}</Data>
        <Data Name="ProcessId">5659</Data>
        <Data Name="Image">/home/pocker/miniconda3/envs/py311/bin/python3.11</Data>
        <Data Name="User">pocker</Data>
    </EventData>
</Event>
```


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
tcp.stream eq 2
```

```
No.   Time     Delta    Source     Destination  Protocol  Length  Info
   97 6.314541 0.000000 localhost  localhost    TCP           60 39710 → ndmp(10000) [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=3176071075 TSecr=0 W
   98 6.349673 0.035132 localhost  localhost    TCP           60 [TCP Retransmission] 39710 → ndmp(10000) [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSva
   99 6.378427 0.028754 localhost  localhost    TCP           60 [TCP Retransmission] 39710 → ndmp(10000) [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSva
  100 6.409933 0.031506 localhost  localhost    TCP           40 ndmp(10000) → 39710 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  101 6.410893 0.000960 localhost  localhost    TCP           40 ndmp(10000) → 39710 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
  102 6.411881 0.000988 localhost  localhost    TCP           40 ndmp(10000) → 39710 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

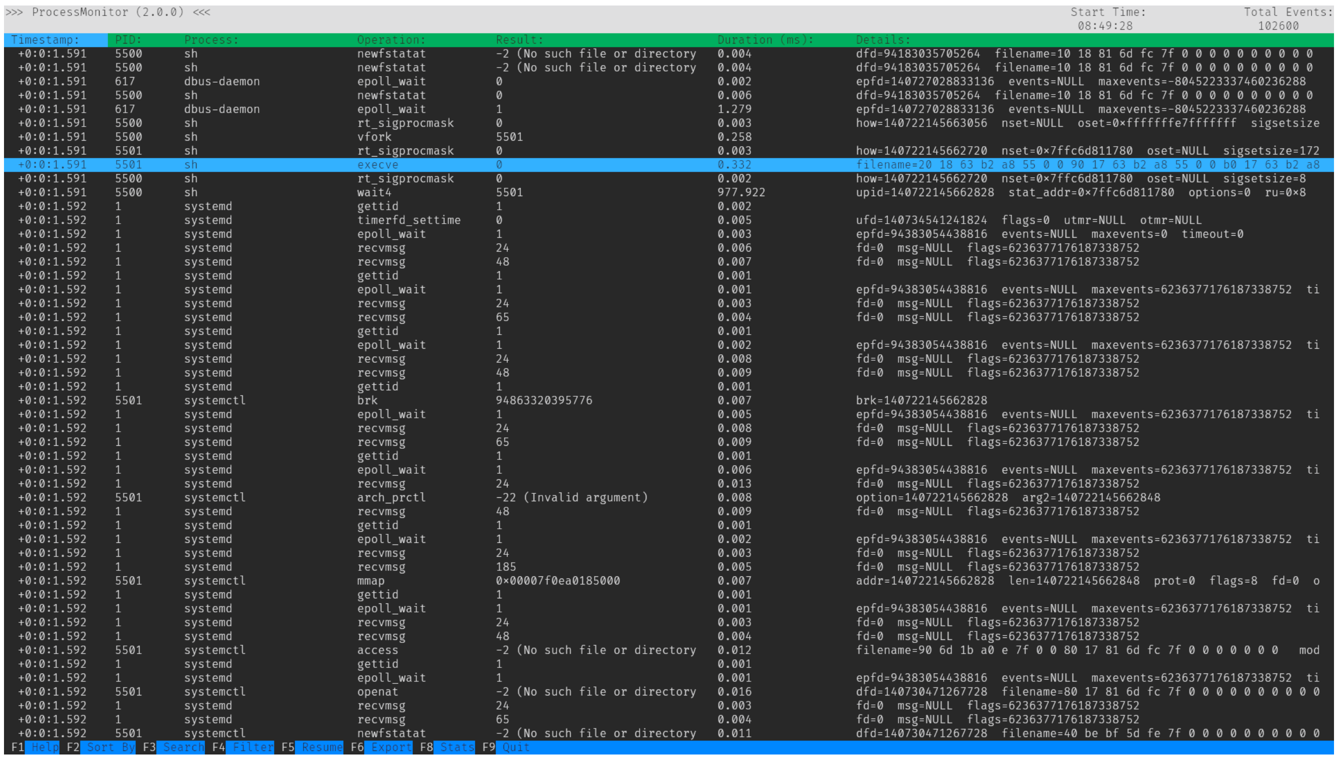
```
+ Frame 97: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
  Raw packet data
+ Internet Protocol Version 4, Src: localhost (127.0.0.1), Dst: localhost (127.0.0.1)
+ Transmission Control Protocol, Src Port: 39710 (39710), Dst Port: ndmp (10000), Seq
```

```
0000  45 00 00 3c ee 33 40 00  40 06 4e 86 7f 00 00 01   E··<·3@· @·N·····
0010  7f 00 00 01 9b 1e 27 10  9e 72 0a 1b 00 00 00 00   ······'· ·r······
0020  a0 02 ff d7 28 53 00 00  02 04 ff d7 04 02 08 0a   ····(S·· ········
0030  bd 4e ff a3 00 00 00 00  01 03 03 07               ·N······ ····
```

```
Webmin_exploit.py_fakenet.pcap          Packets: 382 · Displayed: 6 (1.6%)          Profile: Default
```

Figure 6: TCP trace viewed in Wireshark

The resulting event logs from Procmon contain only system calls, making it very difficult to analyze if you do not know what to look for. Additionally, this results in the log file growing very large, very quickly. These logs can be viewed using Procmon for further inspection. Figure 7 shows a fragment out of the 102,600 total events.



```
>>> ProcessMonitor (2.0.0) <<<                                                                     Start Time:          Total Events:
                                                                                                     08:49:28              102600
Timestamp:   PID:     Process:             Operation:         Result:                       Duration (ms):   Details:
 +0:0:1.591  5500     sh                   newfstatat         -2 (No such file or directory  0.004           dfd=94183035705264  filename=10 18 81 6d fc 7f 0 0 0 0 0 0 0 0 0 0
 +0:0:1.591  5500     sh                   newfstatat         -2 (No such file or directory  0.004           dfd=94183035705264  filename=10 18 81 6d fc 7f 0 0 0 0 0 0 0 0 0 0
 +0:0:1.591  617      dbus-daemon          epoll_wait         0                              0.002           epfd=140727028833136  events=NULL  maxevents=-8045223337460236288
 +0:0:1.591  5500     sh                   newfstatat         0                              0.006           dfd=94183035705264  filename=10 18 81 6d fc 7f 0 0 0 0 0 0 0 0 0 0
 +0:0:1.591  617      dbus-daemon          epoll_wait         1                              1.279           epfd=140727028833136  events=NULL  maxevents=-8045223337460236288
 +0:0:1.591  5500     sh                   rt_sigprocmask     0                              0.003           how=140722145663056  nset=NULL  oset=0×fffffffe7fffffff  sigsetsize
 +0:0:1.591  5500     sh                   vfork              5501                           0.258
 +0:0:1.591  5501     sh                   rt_sigprocmask     0                              0.003           how=140722145662720  nset=0×7ffc6d811780  oset=NULL  sigsetsize=172
 +0:0:1.591  5501     sh                   execve             0                              0.332           filename=20 18 63 b2 a8 55 0 0 90 17 63 b2 a8 55 0 0 b0 17 63 b2 a8
 +0:0:1.591  5500     sh                   rt_sigprocmask     0                              0.002           how=140722145662720  nset=0×7ffc6d811780  oset=NULL  sigsetsize=8
 +0:0:1.591  5500     sh                   wait4              5501                           977.922         upid=140722145662828  stat_addr=0×7ffc6d811780  options=0  ru=0×8
 +0:0:1.592  1        systemd              gettid             1                              0.002
 +0:0:1.592  1        systemd              timerfd_settime    0                              0.005           ufd=140734541241824  flags=0  utmr=NULL  otmr=NULL
 +0:0:1.592  1        systemd              epoll_wait         1                              0.003           epfd=94383054438816  events=NULL  maxevents=0  timeout=0
 +0:0:1.592  1        systemd              recvmsg            24                             0.006           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            48                             0.007           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  1        systemd              epoll_wait         1                              0.001           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  1        systemd              recvmsg            24                             0.003           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            65                             0.004           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  1        systemd              epoll_wait         1                              0.002           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  1        systemd              recvmsg            24                             0.008           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            48                             0.009           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  5501     systemctl            brk                94863320395776                 0.007           brk=140722145662828
 +0:0:1.592  1        systemd              epoll_wait         1                              0.005           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  1        systemd              recvmsg            24                             0.008           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            65                             0.009           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  1        systemd              epoll_wait         1                              0.006           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  1        systemd              recvmsg            24                             0.013           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  5501     systemctl            arch_prctl         -22 (Invalid argument)         0.008           option=140722145662828  arg2=140722145662848
 +0:0:1.592  1        systemd              recvmsg            48                             0.009           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  1        systemd              epoll_wait         1                              0.002           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  1        systemd              recvmsg            24                             0.003           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            185                            0.005           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  5501     systemctl            mmap               0×00007f0ea0185000             0.007           addr=140722145662828  len=140722145662848  prot=0  flags=8  fd=0  o
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  1        systemd              epoll_wait         1                              0.001           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  1        systemd              recvmsg            24                             0.003           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            48                             0.004           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  5501     systemctl            access             -2 (No such file or directory  0.012           filename=90 6d 1b a0 e 7f 0 0 80 17 81 6d fc 7f 0 0 0 0 0 0 0   mod
 +0:0:1.592  1        systemd              gettid             1                              0.001
 +0:0:1.592  1        systemd              epoll_wait         1                              0.001           epfd=94383054438816  events=NULL  maxevents=6236377176187338752  ti
 +0:0:1.592  5501     systemctl            openat             -2 (No such file or directory  0.016           dfd=140730471267728  filename=80 17 81 6d fc 7f 0 0 0 0 0 0 0 0 0 0
 +0:0:1.592  1        systemd              recvmsg            24                             0.003           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  1        systemd              recvmsg            65                             0.004           fd=0  msg=NULL  flags=6236377176187338752
 +0:0:1.592  5501     systemctl            newfstatat         -2 (No such file or directory  0.011           dfd=140730471267728  filename=40 be bf 5d fe 7f 0 0 0 0 0 0 0 0 0 0
F1 Help F2 Sort By F3 Search F4 Filter F5 Resume F6 Export F8 Stats F9 Quit
```

Figure 7: Procmon event trace

8 Discussion

8.1 Evaluation

Different methods of argument generation were examined for entry points both supported by Codarg and those that are not. For entry points not supported by Codarg, the system started with a decent level of code coverage without providing any arguments. This coverage increased when using more complex feedback processors. In contrast, entry points that do support Codarg began with a much lower initial code coverage. Although the use of fuzzing techniques led to some improvements, these gains were a lot more balanced compared to non-Codarg entry points. However, when Codarg generated arguments were used, there was a substantial increase in code coverage. This shows that argument generation using Codarg can be a powerful approach to boost code coverage.

The difference in coverage gains between Codarg-supported and non-Codarg-supported entry points can be attributed to the reliance on named arguments in Codarg-supported entry points. These arguments tend to be more complex and context-specific. This means that naive guessing is much less effective due to the number of possible combinations. By taking advantage of Python's interpretive nature, where the code can easily be read and analyzed, the system can effectively narrow down the combination space, leading to higher coverage.

A less groundbreaking finding is that installing software dependencies indeed does help run the program. This is an expected outcome but it still plays an important role in the rest of the automation. As seen in Figure 4, there is a noticeable coverage improvement when the necessary requirements are identified and installed. Where initially entry points often exited early due to unavailable modules, after the requirements are resolved, the program is able to proceed beyond these limitations. This lead to a more effective and thorough analysis.

8.2 Limitations

Our current system for detecting Python 2 projects has a significant limitation due to exclusively testing the Python version with a `Vermin` release that only supports Python 3 syntax. This means our automated argument generation and dynamic tests are not fully effective for Python 2 projects. Additionally, we are unable to identify project boundaries if there are multiple entry points within a single repository that target different Python versions. This makes our version detection unreliable under these circumstances, potentially resulting in undetected compatibility issues or incorrect test results.

The software has not been tested on Windows or macOS. If a repository uses operating system-specific APIs, this might cause errors when the software runs on Linux. This means that functions or features designed for Windows or macOS could behave unpredictably or fail when executed on Linux.

A coverage metric on fuzzing can be misleading because it does not always reflect how well a fuzzer tests a program. If our simplified fuzzer sends invalid or too few arguments, the program might enter different code paths, like those that show help messages for proper use. These alternate paths might increase the coverage metric but do not test the program in the same way valid arguments would. As a result, the fuzzer could mistakenly conclude that these paths are preferable, leading it to a local optimum and giving a false sense of thorough testing.

The current system only supports identifying arguments for the `argparse` module. Although this might seem like a limitation at first glance, it is not really a drawback because `argparse` is the most commonly used tool for parsing command-line arguments in our dataset and in general.

Input generation for Python programs that do not receive their inputs through command-line arguments is not supported. This limitation means we cannot generate inputs for entry points that require input from the standard input stream or a configuration file, or rely on hard-coded 'inputs'. Graphical applications are also not supported for input generation.

8.3 Future work

While the current system automates the execution and logging phases of dynamic analysis, it does not perform the actual analysis. It currently prepares logs for human review rather than interpreting them by itself. To create a truly automated system, it is necessary to integrate an interpreter that can process and understand these logs. This interpreter would need to evaluate whether the PoC is benign or malicious. Without this analysis capability the system is not fully automated, as it still depends on manual inspection.

In addition to achieving full automation, further enhancements could be made to increase the effectiveness of the dynamic analysis. For example, maximizing code coverage could be improved by implementing 'controlled inputs' for our argument generation and dynamic testing. Passing a specific IP address and port combination and making sure the test environment has a service listening on that port could increase code coverage. The same is true for passing a valid file path and name to test file operations, etc. By implementing these controlled inputs, out dynamic analyses would be able to simulate real-world conditions more accurately, which will likely lead to more informative executions.

Another interesting possibility for enhancing the argument generation is the usage of AI-driven code analysis. By feeding the source code to machine-learning models, these can understand the handling of input arguments and 'usage strings', which are usually used to output information on how to use the program when e.g. `--help` was entered as an argument. The models might even be able to understand the PoC to such a degree that it can reliably generate a few argument vectors that together produce a code coverage that approaches 100%, which might make fuzzing and our current argument generator (Codarg) obsolete.

9 Conclusions

This research aimed to automate the dynamic analysis of Python Proof of Concept (PoC) exploits. We achieved this by developing and testing an automated system. This system processed a representative sample of Python PoC repositories, where the entry points in each repository were dynamically analyzed in a sandboxed environment.

We started by creating a virtualized environment to safely execute Python PoCs. This environment used QEMU and KVM for virtualization, which made sure that the tests ran in isolation to protect the host system from any potential harm. We automated the creation and restoration of snapshots to streamline the process and allow for predictable and reproducible analyses.

Next, we automated the dependency resolution process. Using the `Pipreqs` module, we generated missing requirements specifications to make sure all necessary dependencies were identified and installed before executing the PoCs. This step was important because missing dependencies could halt the analysis prematurely.

We then moved on to the argument generation phase. Codarg was designed to generate argument vectors through static code analysis. Codarg was supplemented by fuzzing techniques to maximize code coverage. The generated arguments were used in the execution phase, and their effectiveness was measured. One limitation here was the restriction of Codarg to entry point that use `argparse` for argument handling.

The execution phase involved running the PoCs with the generated arguments while monitoring their behavior using FakeNet-NG, Procmon, and Sysmon. These tools provided detailed logs and traces, capturing the interactions of the PoCs with the system and network resources.

Our results showed that the automated system notably improved the code coverage of Python PoCs. For instance, using exit status-driven arguments, the average code coverage increased from 34.07% to 38.42%. Coverage-driven arguments raised it further to 40.23%. When using Codarg-generated arguments on supported PoCs, the average code coverage improved from 22.63% to 30.75%. These results indicate that Codarg can effectively boost code coverage, although Codarg supported roughly 20% of repositories.

In conclusion, automating the dynamic analysis of Python PoCs is not only feasible but also beneficial. While there are limitations, particularly concerning the Python version determination, the overall approach demonstrates significant potential as a foundation for further research.

References

[ASFG24] Md Ahsan Ayub, Ambareen Siraj, Bobby Filar, and Maanak Gupta. Rwarmor: a static-informed dynamic analysis approach for early detection of cryptographic windows ransomware. *International Journal of Information Security*, 23(1):533–556, 2024.

[Bal99] Thomas Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[BMFT15] Alexander Barabanov, Alexey Markov, Andrey Fadin, and Valentin Tsirlov. A production model system for detecting vulnerabilities in the software source code. In *Proceedings of the 8th International Conference on Security of Information and Networks*, pages 98–99, 2015.

[CGP24] Tanmay Chaudhry, Abhay Garg, and Yogesh Pathak. Envyr: Instant execution with smart inference. *Procedia Computer Science*, 238:1068–1073, 2024.

[EP22] Aryaz Eghbali and Michael Pradel. Dynapyt: a dynamic analysis framework for python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 760–771, 2022.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, 2007.

[GS15] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*, pages 581–591. Springer, 2015.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pages 19–28, 2009.

[Hot05] C Hote. Extension of static verification techniques by semantic analysis. In *24th Digital Avionics Systems Conference*, volume 2, pages 9–pp. IEEE, 2005.

[LAdS$^+$07] Gregory L Lee, Dong H Ahn, Bronis R de Supinski, John Gyllenhaal, and Patrick Miller. Pynamic: the python dynamic benchmark. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 101–106. IEEE, 2007.

[May19] Justin Mayer. Zen of python dependency management. EuroPython, 2019. https://doi.org/10.5446/44843.

[NH19] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[Oei23] Jimmy Oei. Automated dynamic analysis of java exploit proof-of-concepts. Bachelor's thesis, Leiden University, 2023.

[Ope14] OpenBSD. Exit manual page. https://man.openbsd.org/exit, 2014. Accessed: 2024-07-28.

[Pyt23] Python Software Foundation. Python 3 Documentation: __main__ - Top-level script environment, 2023. Last updated on Jul 27, 2024; accessed on Jul 28, 2024.

[RSRS24] Rahul Raj R, Naveen S, Subhikshan R, and Tarun S. Malware analysis using sandbox. *SSRN Electronic Journal*, 2024.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[TDBR24] Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. Nebula: Self-attention for dynamic malware analysis. *IEEE Transactions on Information Forensics and Security*, 2024.

[vR24] Guido van Rossum. What's new in python 3.0, 2024. Last updated on Jul 27, 2024; accessed on Jul 28, 2024.

[YLAI17] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):1–40, 2017.

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

[Zha22] Wenxi Zhang. Obtaining fuzzing results with different timeouts. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 236–239. IEEE, 2022.