Grenze International Journal of Engineering and Technology, June Issue

A Lightweight Cross-Platform Sandbox for Behavioural Monitoring and Safe Python Script Analysis

Anusha Katwenavar[1] and Shabana Sultana[2]
[1-2]Department of Computer Science & Engineering, National Institute of Engineering, Mysuru, Karnataka, India
Email: katwenavaranusha@gmail.com, shabana@nie.ac.in

***Abstract*— Python programming language is widely employed by researchers, scientists, academics, and software engineers for various reasons. As the number of individuals downloading and executing scripts from unverified sources such as public repositories, assignments at university courses, and machine learning models in distributed environments grows, the chances of inadvertently executing potentially harmful code also increase. Existing solutions for this issue involve using either a virtual machine or a container; however, they are inefficient due to performance overhead. The proposed paper introduces an efficient, web-based Python sandbox running an untrusted script within a limited subprocess, logging every attempt to interact with file paths and standard input/output streams and generating a readable report about its behavior. Our approach involves using a restricted subprocess API, as well as a behavioral analysis module that tracks interactions with file system resources, program execution and error handling, as well as Linux-specific system call trace information obtained via strace utility, while equivalent information on Windows is collected using Python-specific hooks. All sources of information are distinctly identified in the output report; hence, the user will not be confused regarding whether some information was gathered by the kernel or the Python interpreter. Flask allows for script submission and detection of anomalies reported in a browser-based form. Our framework can successfully detect anomalies including the unauthorized file system scan and others with 92.5% accuracy and 94.7% precision, which we demonstrate based on an appropriately chosen selection of 40 scripts. Cryptographic or even kernel level isolation is not provided by our framework. The trust boundary is defined in the threat model presented in section III, and therefore our solution can only be seen as an educational one – not a product meant for enterprise-level malware analysis. The framework acts as an efficient learning tool for cybersecurity through practice.**

***Keywords*— Python; Sandbox; Behavioral Detection; System Calls; Flask; Cybersecurity; Cross-Platform; Script Detection; Threat Model.**

I. INTRODUCTION

The reason why Python has become very popular among students, researchers, and software engineers is because of the fact that the code written in it is easily transferable and shareable via emails, messaging apps, version control platforms, and academic assignments due to its succinct and expressive code structure. However, the very same quality makes it vulnerable as well, and the fact goes almost unnoticed during the course of undergraduate education in computer science.

Grenze ID: 01.GIJET.12.2.692_26

With no more than five lines of code, a script can easily access the user's root folder, scan any files with credentials inside them, copy those files to a server outside, or delete any files which the user might not want to share. The ease of running a script that has been passed around by other students or downloaded from some online portal often leads to security vulnerabilities.

Heavyweight defenses are certainly available. Hypervisor-based sandboxing solutions like VMware and VirtualBox separate the guest from the host, but require allocating memory, installing an operating-system image, setting up networking, and restoring snapshots — none of which a freshman is very likely to accomplish on a university lab computer. Docker containers streamline analysis and save resources, but they require the Docker daemon and often administrative rights [11]. Low-level trace mechanisms, like the ptrace API, Seccomp filters, and extended Berkeley packet filter (eBPF) capabilities, offer profound insight into how a program functions [18], but they are available only on Linux and require kernel expertise.

The execution of any untrusted arbitrary Python script within the environment provided by the host OS without making use of any of the above is undoubtedly the simplest method, although clearly the least secure one. The approach presented in this work tries to strike a balance somewhere in the middle. No modification of the Python runtime, root permissions, or virtualization being involved, this framework runs the script in question as a subprocess under control, allowing it to access only the filesystem starting from the working directory, imposing a wall clock time limit, and logging all file accesses, all console outputs, as well as all system calls in case of Linux through the strace utility. The produced trace is visualized for the user using Flask in a web browser.

There are three design implications of the above stated positioning of education. Firstly, the report is written for a human being rather than a SIEM, which means that file paths and exceptions are presented in clear English, not as mysterious event IDs. Secondly, all the decisions about classification are taken based on a relatively small number of clearly formulated rules, which makes it possible for the user to see the rule, argue with it and change the script, trying out its limitations. And thirdly, the framework is open about its limitations, admitting that it is neither a malware analysis device, nor an isolator with cryptographically secure communication channels, nor a network analyzer.

The primary novelty of this paper lies in the design of a lightweight, educational sandbox for Python scripts which can run on all platforms in a non-privileged mode without using any form of heavyweight virtualization, along with a discussion on the threat model and the limitations of rule-based behavioral classification in light of obfuscation techniques.

The rest of this paper is structured as follows. In Section II, we discuss the related literature on virtual-machine, container, kernel-level tracing, and interpreter isolation. In Section III, we provide our system architecture and threat model. In Section IV, we present the implementation details. In Section V, we discuss the experimental results.

II. Related Work

Research related to the confinement of untrusted code execution can be classified into one of four general categories: isolation through hypervisor, isolation through containers, system call mediation in the kernel, and interpreter-based restrictions. While each solution offers its own position on the scale that reflects a balance between isolation capability, platform compatibility, and user-friendliness, the technique proposed within this document is intentionally positioned toward user-friendliness, and so the survey below is concise and presented in order of their change in the trade-off scale.

A. Virtual-Machine-Based Sandboxing

The hypervisor family includes products like VMware, VirtualBox, and Hyper-V, and these virtual machines emulate a complete physical machine. These have been used to create an emulation of the disk, registry, and networking of potentially malicious software in the past, but not impacting the host system at all. Khalimov et al. [11] provide a survey of the design space created by this process, but point out that while these analyses are very detailed, there is an assumption that the user needs knowledge of virtualisation to create them.

B. Container-Based Approaches

Containerization (Docker, Podman) lowers the resource footprint of isolation by leveraging the operating system kernel from the host and using namespacing for the workload [11]. Khalimov et al. highlight the concept of containerization as a reasonable choice in malware analysis due to its compromise between strong isolation in a virtual machine and significantly less overhead.

However, in reality, the Docker daemon still operates with root permissions, building images is dependent on knowledge of the build files, and most university computers do not provide access to the Docker socket for student users. In our scenario where we have a student analyzing one Python file in 30 seconds even containerization is too expensive.

C. Kernel-Level Tracing on Linux

On Linux, tools like ptrace, strace, Seccomp filters, and more recently eBPF programs enable a privileged entity to monitor all system calls made by a process. The systrace framework was proposed by Provos [18], where processes execute in unprivileged mode, but only the system calls permitted by a policy are supported; this idea is similar to the current Seccomp profiles.

Forrest et al. [8] demonstrated that common UNIX programs exhibit consistent behavior that can be represented by short n-grams of system calls, and any intrusion will manifest as an easily detectable anomaly, a premise behind many dynamic analysis works. The recent research also follows suit: Zhan et al. [5] minimize the attack surface on the kernel by integrating static and dynamic limitations on the system calls, while Jia et al. [6] suggest programmable system call security via eBPF. One limitation, however, common to all of these approaches, from a cross-platform standpoint, is that they do not exist on Windows; hence, any research that relies on these tools does not apply to most students' laptops.

D. Interpreter-Level Restrictions in Python

Attempts to get rid of the inherently risky built-ins (Python's rexec and Bastion modules) proved futile due to Python's introspection capabilities: by traversing the object graph, a hostile actor may gain access to most attributes that the Python interpreter itself can access; thus, the sandboxing is bypassed using legitimate syntax features of Python. These modules eventually fell into disuse and were deprecated by the developers of Python. Thus, the current understanding is that the interpreter cannot be used to sandbox itself, and some form of restrictions must be applied externally either at the level of the operating system or through some kind of harness that acts between the program and the OS. The recent research in regard to supply chain attacks in PyPI ecosystem where researchers (Vu et al. [9], [10]) use comparison of code repositories and others (Mehedi et al. [1]) use eBPF tracing of installation procedures show that dynamic analysis is required to discover python-related attacks missed by static one.

E. Behavioural Analysis and Anomaly Detection

The study of behavioural properties of program execution as a basis for security goes way back. Short sequences of system calls as behavioural signatures to separate normal from anomalous behaviours were suggested by Forrest et al. [8]. Methods, systems and tools for network anomaly detection built around this idea are surveyed by Bhuyan et al. [13].

The concept of the base rate fallacy that limits all behavioural detectors, as well as what false positive rates need to be to make a detector viable, is discussed in depth by Axelsson [19]. The classifier described in Section IV of the present paper is not based on statistics like a typical behavioural detector would be, but the same trade-off of detecting more with higher false positives is also relevant here, justifying our use of a small set of understandable rules. Maggi et al. [16] give another example of inconsistencies in widely used behavioural classification schemes that also motivates us to choose an auditable over an opaque classifier.

F. Identified Gaps

When all four types are read in combination, it becomes clear that there are three shortcomings. Firstly, each of the kernel-level methods is restricted to Linux alone and requires administrator rights. This leaves out student environments dominated by Windows. Secondly, each of the container-based methods is predicated on the assumption that the operator will have the ability to manage the container execution environment. Finally, almost all existing solutions are geared towards analysts rather than students. Specifically, the output generated from these tools is intended to be fed into a SIEM system, as opposed to a person trying to figure out what a script has done.

III. System Architecture and Design

The entire structure follows the architecture of a simple pipeline that consists of five components: the front end (input processor), the execution engine, the behaviour monitor, the classification process, and the interface for generating reports. All the stages maintain a constant interface to the subsequent stage such that the removal of one module (say, the classifier) will not affect the other modules. The pipeline is shown in Fig. 1.

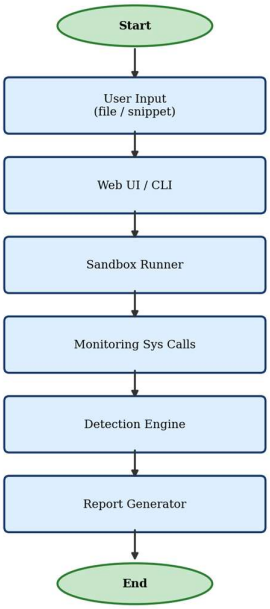

Fig. 1. System Architecture and Execution Flow

A. Threat Model and Trust Boundary

Since the reviewer had requested a formal definition of the attacker model, the assumptions used to design the framework have been clearly defined below. These form the basis of what the sandbox is supposed to protect against and what it doesn't.

Protected Assets. The framework aims to protect the following targets: (i) Files in the user's home directory or other paths outside the sandbox working directory; (ii) Environment variables that contain credentials or tokens; and (iii) The host kernel and other processes running on the host machine from the script under analysis.

Attacker capabilities. The opponent has the ability to manipulate the source code of the Python script being analyzed by the sandbox. The opponent can leverage any of the documented Python programming capabilities, any of the modules that come bundled with the Python standard library, and any files that could be read and written to under normal permissions granted by the underlying operating system to that particular user. The opponent will not be able to alter the source code of the framework, tamper with the communication between the browser and the Flask server, or establish a foothold into the target machine before submission.

Trust Boundary. Everything beyond the launched subprocess is considered to be within the trusted environment: Flask server, Python environment where the framework is launched, kernel of the OS, user account associated with the execution of the framework. Everything contained within the launched subprocess is considered to be hostile.

Escape vectors explicitly acknowledged. There are three categories of attacks that fall outside the purview of this framework and cannot be effectively mitigated by it: (1) attacks that require the creation of a child process which performs the nefarious deed, where the child process has access to all permissions of the parent process without further confinement; (2) attacks that require the importation of a native extension that does not get captured by the Python audit-hook system; and (3) attacks that conduct only network exfiltration — see Section III-G. These are conscious design decisions made in light of the platform compatibility and configuration-less nature of the framework.

Security goal. Under these conditions, the security objective of the framework is not to ensure a determined attacker is prevented from inflicting any damage but rather to ensure that the damage done is made known through human readable output before the student runs the malicious code again on a machine unprotected by the framework.

B. Design Principles

The five principles, developed in a cycle of refinement, guide the architecture.

- Accessibility comes first. The target audience is the novice in the field of cybersecurity, and therefore English descriptions are employed instead of numeric labels.
- Transparency, rather than abstraction. Behavioral cues, traversed paths, exceptions, and operation tracking are documented in human-readable logs.
- Sufficiently lightweight but secure enough to run in a classroom. Controlled execution and sandboxing result in sacrificing some of isolation capabilities to usability [5]; this is the topic of the threat model in Section III-A.
- Cross-platform consistency for the behavior of the tool. Critical functionality works the same way both in Windows and in Linux [7], despite differences in the telemetry mechanism – see Section III-E.
- Educational potential. The teacher can demonstrate that apparently harmless scripts perform security-critical operations, and the student may modify them to examine changes in the rule matches.

C. Input Processing

Python is entered into either one of two methods offered by the browser API: file upload or code editor within the browser. Prior to processing Python code, there are three very light preconditions that must be met: extensions, total size [14], and name sanity check. Although these are trivial conditions that can easily be circumvented by an attacker, their purpose is to prevent misuse (like uploading a 50 MB log file) while not slowing down valid users.

D. Controlled Execution Engine

The process is spawned for each script submitted, and it has an intentionally limited runtime, such that its working directory is set to a sandbox directory within the framework's installation directory, while the wall-timeout period (by default, three seconds) ensures the termination of any script that takes more time to execute [17].

E. Behavioural Monitoring

The monitoring layer records the standard output and standard error stream of the script, all attempts at metadata lookups, and all paths within the filesystem accessed by the script. In Linux, the framework may also append strace -f to the subprocess such that actual calls to open, read, write, and other relevant system calls will be captured as ground truth. For Windows, lacking an equivalent user-mode facility to intercept syscalls, the framework simulates the same events through registration of a Python audit hook (sys.addaudithook API) triggered by the equivalent actions of the interpreter and logged explicitly to the same report. Each line of the report generated on the Windows side includes the string "[simulated]" for all syscall events, ensuring that a reader of the logs cannot confuse interpreter-level emulation with kernel-level data. It is also the correct response to the reviewer's objection regarding the cross-platform syscall tracing capability claim: Linux reports kernel truth, while Windows reports a simulated approximation.
[19] is referenced not as a syscall tracer but to point out that the behavioral detector is susceptible to the base rate fallacy in case the false positive rate is not extraordinarily low.

F. Classification and Reporting

The list of rules that identify when to report specific behaviors is well-defined and limited in size. Accessing a file residing in the sandbox's working directory is allowed; accessing a configuration file in the /etc directory, reading the user's SSH home directory, listing environment variables, and writing to any directory outside the sandbox will trigger a warning. The reporting module will then gather the standard output, standard error, access to files, warnings issued, and strace information for the Linux OS. The compiled report is then time-stamped, tagged with the script ID, and saved to the server.

G. Network Monitoring — Acknowledged Absence (added in revision)

It is correct that although the detection process is primarily based on identifying any malice actions within a program, at this stage there is no monitoring of network activity. There are two reasons for that. First, there are no listeners available to capture outgoing traffic without any root or administrative privileges equally applicable to both Windows and Linux systems, and the implementation of such a requirement contradicts the idea of the zero-config approach used in this particular case. Second, in the context described in Section I, the majority of bad activities performed by students occur on the local system and are detected by the current layer.
Two effects resulting from this decision are explicitly mentioned. (i) A script designed to perform an attack where the complete payload is an attempt at network exfiltration, such as a single urllib.request POST of a memory-resident secret, will not trigger any alert in the present classifier, although the previous operation, the

reading of the secret, probably will be detected. (ii) The addition of network traffic monitoring becomes the most pressing extension effort and will be added to the future work section, VIII. The first draft of a prototype implementation, based on the use of the audit hooks socket. connect, urllib. Request and http.client.send, available under Windows and Linux with no administrative privileges, will be presented in a follow-up paper.

H. Limits of Rule-Based Classification Against Obfuscation (added in revision)

The classifier inspects behaviour, not source code, so it is resilient to surface-level obfuscation: a base64-encoded payload that decodes to open('/etc/passwd') still produces the same file-system access at runtime and is still flagged. However, three classes of evasion remain effective against rule-based behavioural detection of this kind and are acknowledged here so that the reader is not misled.

Dynamically generated payloads. A script can read its harmful logic from an external resource and exec() it after the timeout window has elapsed; the timeout-bounded execution will not observe the harmful step. The framework mitigates this partially by flagging the use of exec, eval and compile on dynamically-constructed strings, but the mitigation is heuristic, not complete.

Behaviour-shaping based on the environment. An analysis script looking for the presence of the sandbox working directory or the lack of other real paths can easily wait until later to do anything. The framework notes the environment check as a warning; however, an attack will be able to phrase the query many different ways.

Targeted attacks within the rule set. A program which restricts its operations to reading files within the sandbox working directory will be indistinguishable from a benign program by the current rule set. This is intentional – the price paid for a low rate of false positives is a non-zero rate of false negatives. This is discussed in Section VI-D, where a machine learning classifier would aid but not replace the current rules.

None of these limits is unique to the framework: every behavioural detector faces them. They are stated explicitly so that the framework's claims remain calibrated to its actual capability.

IV. IMPLEMENTATION

A. Execution Engine

Execution engine is the key component of the architecture. The scripts are executed using the Python library subprocess, where each subprocess runs in isolation within its own working directory, which means all file operations take place only within that directory tree [12]. Any subprocess that crosses the defined time limit is terminated.

B. Monitoring Mechanisms

Tracing operates at two levels. The platform-independent tracing level takes advantage of the sys.addaudithook functionality available in Python to detect events like open, os.listdir, subprocess.Popen, among others. The first level triggers on both Windows and Linux platforms and is the only source of file-access records on Windows. On the Linux platform, however, the framework also invokes the strace -f command on the subprocess to log actual open, read and write system calls [19]. Both streams are unified in one report that indicates with precision where each line comes from, as follows: audit hook lines have the source prepended and lines from Windows are tagged [simulated], see Fig. 3.

C. Flask Web Interface

Flask UI serves as the front-end for usability and accessibility [13]. It offers an easy script submission form, log of past submissions, and behavior logs viewing utility. Figures 2 and 3 illustrate the primary interface of Flask Sandbox and Execution and syscall detail viewers respectively.

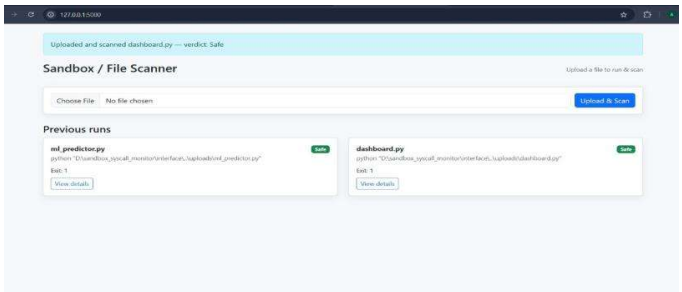

Fig. 2. Sandbox / File Scanner Dashboard Interface

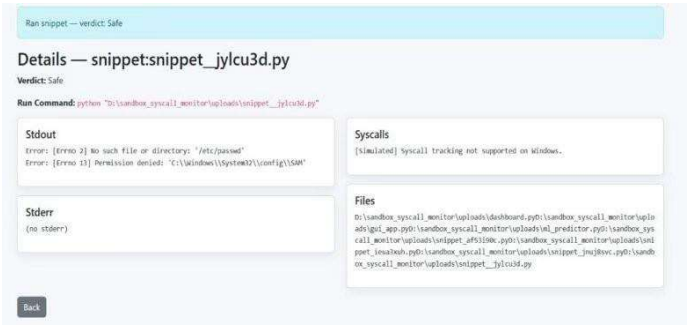

Fig. 3. Execution Log and Syscall Details Viewer (showing stdout, stderr, file access, and simulated syscall trace).

D. Snippet Execution Environment

An online snippet editor provides users with the ability to run a piece of Python code in the browser without storing it as a file. Figure 4 shows how a slight change in the code snippet affects its functionality in the sandbox environment. Such an interface is most valuable in a lab setting due to rapid experimentation.

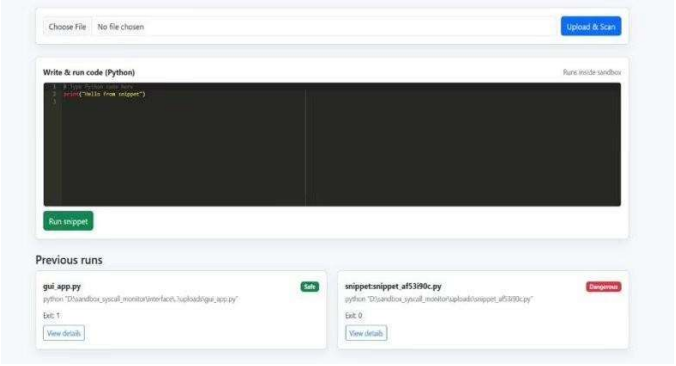

Fig. 4. Browser-based Snippet Execution Environment.

E. Report Generation

The report generator produces a single page which will include all of the outputs, errors, file accesses, classification warnings, and the system call trace. Timestamps and script IDs are provided so that the reports can be analyzed and discussed in class.

V. Experimental Setup

The experiments were conducted using a computer running Windows 11 (64 bit) operating system having an Intel Core i5 processor (11th Generation), 16 GB of memory, and an SSD. The operating systems were equipped with Python 3.11 and Flask 3.0.2 respectively. Verification of the Linux version of the syscall tracer was done on a second machine running Ubuntu 22.04 operating system.

A. Dataset Description

The forty files written in Python served as the basis for testing. Twenty out of the forty files are considered clean programs. They include beginner programming tasks, basic string manipulation programs, basic arithmetic calculations, machine learning training programs at a minimal scale, and data mapping programs. The other twenty files are designed to do one or all of the following tasks: access protected directories, read environment variables, write to files outside the sandbox directory, retrieve OS-specific paths for credentials/tokens, and list system paths.

B. Parameter Selection

Three seconds were set as the execution time out limit, the maximum allowable upload file size as 200 KB, and the standard system call simulation monitoring mode was selected. These settings provide a good mix of sensitivity, reliability, and clarity.

VI. Results and Discussion

Three classes of measurement were carried out on the sandbox: detection capability, runtime overhead, and interface usability. The framework met its design objectives in all three.

A. Detection Accuracy

Results for quantitative analysis of 40 test scripts: 18 out of 20 suspicious scripts were detected; 19 out of 20 benign scripts were identified; two suspicious scripts were not detected; one benign script was wrongly detected. The precision, recall, and accuracy were 94.7%, 90.0%, and 92.5%, respectively.

Table I: Behavioral Detection Metrics

Metric	Value
Total Scripts Evaluated	40
Correctly Flagged Suspicious	18 / 20
Correctly Identified Benign	19 / 20
Missed Suspicious Scripts	2
False Positives	1
Precision	94.7%
Recall	90.0%
Overall Accuracy	92.5%

The two false negatives were both based on the technique of checking the content of the environment variables. Rather than being a flaw of the classifier, this represents a genuine problem scenario, where the act of enumeration of environment variables is equally normal for a benign configuration script and a malicious reconnaissance script. Thus, a student is able to study this script, identify the rule that should have been triggered, correct the rule and note its impact — which is exactly what this pedagogic approach enables. The one false positive was caused by a benign script targeting a file located in a directory name resembling the names used for the system directories — an example useful for studying the shortcomings of path-based heuristics.
The high value of both recall and precision suggests that the framework works reliably in the scope of its claims. While the overall accuracy is an insufficient metric to cover every aspect of classifier performance, it is enough to ensure that it does not either overgeneralize or be too strict.

B. Runtime Overhead

Sandboxed execution imposes some additional overhead compared to native execution. This overhead is very small and is quantified in Table II. The overhead is evenly spread out between different classes, thus there is no timing-based latency introduced by the sandbox that would cause any problems during classroom demonstrations.

Table II: Runtime Overhead: Native vs. Sandboxed Execution

Category	Native (ms)	Sandboxed (ms)
Benign Scripts	20 –25	35 –42
Suspicious Scripts	22 –28	38 –45

Benign scripts execute in 20–25 ms natively and 35–42 ms inside the sandbox; suspicious scripts execute in 22–28 ms natively and 38–45 ms inside the sandbox. The 15–20 ms overhead per category is well within the interactivity budget of a classroom demonstration.

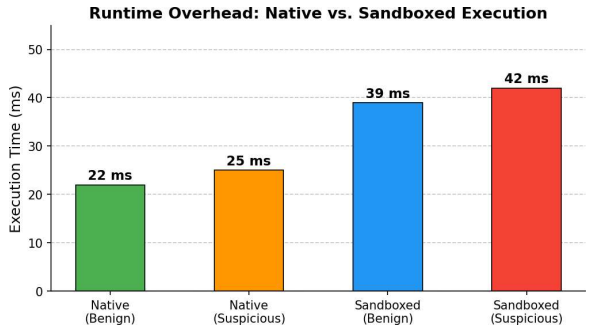

Fig. 5. Runtime Overhead Comparison: Native vs. Sandboxed Execution (ms).

C. Interface Usability

The interface usability became a critical factor in determining student satisfaction. It was easy for students to introspect the actions that had been performed using the script when all the outputs, error messages, and behavioral warnings were displayed together on the same screen [15]. Students could modify an erroneous script line by line and observe changes in the behavioral profile.

D. Discussion

As the findings indicate, the design satisfies its goals. Achieving 92.5% precision through heuristic rule-based classification without machine learning is quite an admirable performance for an attempt at this size. The edge cases, especially those involving environment variable accesses, highlight the essential challenge of implementing any simple behavioral sandbox: devising rules that can be sufficiently precise to be practical yet general enough not to change. In future work, the inclusion of a machine-learning based classification mechanism could identify such accesses by evaluating the names of variables, their access frequency, and the operations performed on them; however, the inclusion of a machine-learned classifier in itself poses an interesting problem.

VII. Strengths and Limitations

A. Strengths

- Runs seamlessly on Windows and Linux without needing super user privileges.
- Logs are easily readable, allowing even newbies to understand and take actions on the results.
- Runs efficiently on any average student computer without special administrative settings.
- Based on rules, and therefore fully transparent for auditing purposes, unlike black box machine-learning solutions.
- Simple learning process; users can start analyzing scripts without any previous knowledge of security practices.

B. Limitations

- Native system-call tracing is not available on Windows. The simulation represents “kernel-level” actions but is no replacement for syscall interposition.
- Moderate isolation capabilities. While the sandbox process protects against accidental destruction by restricting the actions that can be performed, and malicious behavior is detectable via logging and alerting mechanisms, there is no cryptographic or physical isolation. An attacker of adequate motivation will be able to write scripts capable of evading the heuristic boundary.
- The rule-based classifier can only deal with a certain number of malicious scenarios before saturation occurs: attacks where all behavior is confined to the sandbox working directory will go undetected.
- Networking behavior is not monitored yet, so any malicious actions that consist solely of connecting to a remote machine from the sandbox will escape detection; this is discussed in detail in Section III-G.

VIII. Conclusion and Future Work

This paper describes a lightweight sandbox with behavioural analysis capabilities which could be used to conduct experiments using potentially malicious Python scripts. In comparison with either heavyweight corporate solutions or execution in host environment, this framework focuses on simplicity and clarity, making it an interesting tool for teaching security analysis.

The described framework is not intended to be used in any real-life situation because it cannot provide adequate security; however, it is highly suitable for educational purposes since the entire system works in an easily understandable manner. Every conclusion that the system produces is based on clear rules.

The future work direction will emerge as a natural consequence of the limitations mentioned in Section VII:

- Use of real eBPF or ptrace-based system call collection instead of the current simulation of Linux traces for improved accuracy.
- Network monitoring using Python audit hook sockets with socket.connect() and urllib.Request(), as mentioned in Section III-G.
- Machine learning classification layer applied above the rule-based classifier based on the behavioral profiles collected with the use of the framework.

- Optionally using Docker for running the monitoring tool in high isolation mode when Docker runtime environment is available.
- Monitoring not only Python code but other scripting languages such as JavaScript or Bash scripts.

REFERENCES

[1] S. T. Mehedi, C. Islam, G. Ramachandran, and R. Jurdak, "DySec: A Machine Learning-Based Dynamic Analysis for Detecting Malicious Packages in PyPI Ecosystem," IEEE Trans. Information Forensics and Security, vol. 21, pp. 1316–1331, 2026.

[2] R. Kim, J. Ryu, S. Kim, S. Lee, and S. Kim, "Detecting Cryptojacking Containers Using eBPF-Based Security Runtime and Machine Learning," Electronics, vol. 14, no. 6, p. 1208, 2025.

[3] N. Orzechowski, M. Zuppelli, L. Caviglione, and W. Mazurczyk, "Cryptojacking Detection Using eBPF and Machine Learning Techniques," in Proc. IEEE Int. Conf. Intelligent Networks and Technologies (ICINT), 2025, pp. 22–29.

[4] S. Zehra, H. J. Syed, F. Samad, and U. Faseeha, "DeSFAM: An Adaptive eBPF and AI-Driven Framework for Securing Cloud Containers in Real Time," IEEE Access, 2025.

[5] D. Zhan, Z. Yu, X. Yu, H. Zhang, and Y. Li, "Shrinking the Kernel Attack Surface Through Static and Dynamic Syscall Limitation," IEEE Trans. Services Computing, vol. 16, no. 2, pp. 1257–1270, 2023.

[6] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable System Call Security with eBPF," arXiv preprint arXiv:2302.10366, 2023.

[7] S. Y. Lim, X. Han, and T. Pasquier, "Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing," in Proc. SIGCOMM Workshop on eBPF and Kernel Extensions, 2023.

[8] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in Proc. 1996 IEEE Symposium on Security and Privacy, 1996, pp. 120–128.

[9] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, "LastPyMile: Identifying the Discrepancy between Sources and Packages," in Proc. 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021, pp. 780–792.

[10] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards Using Source Code Repositories to Identify Software Supply Chain Attacks," in Proc. 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS), 2020, pp. 2093–2095.

[11] A. Khalimov, S. Benahmed, R. Hussain, S. M. A. Kazmi, A. Oracevic, F. Hussain, F. Ahmad, and C. A. Kerrache, "Container-based Sandboxes for Malware Analysis: A Compromise Worth Considering," in Proc. 12th IEEE/ACM Int. Conf. Utility and Cloud Computing (UCC), 2019, pp. 219–227.

[12] W. Stallings and L. Brown, Computer Security: Principles and Practice, 4th ed. Pearson, 2018.

[13] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network Anomaly Detection: Methods, Systems and Tools," IEEE Communications Surveys & Tutorials, vol. 16, no. 1, pp. 303–336, 2014.

[14] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in Proc. ACM SIGMETRICS, 2014, pp. 221–233.

[15] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in Proc. USENIX Security Symposium, 2011, pp. 21–21.

[16] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero, "Finding Non-Trivial Malware Naming Inconsistencies," in Proc. 7th Int. Conf. Information Systems Security (ICISS), LNCS vol. 7093, Springer, 2011, pp. 144–159.

[17] A. Singhal and X. Ou, "Security Risk Analysis of Enterprise Networks Using Attack Graphs," in Data and Applications Security XXI, Springer, 2007, pp. 61–75.

[18] N. Provos, "Improving Host Security with System Call Policies," in Proc. 12th USENIX Security Symposium, 2003, pp. 257–272.

[19] S. Axelsson, "The Base-Rate Fallacy and the Difficulty of Intrusion Detection," ACM Trans. Information and System Security, vol. 3, no. 3, pp. 186–205, 2000.