# Designing Trainee Performance Assessment System for Hands-on Exercises

***Austėja Bauraitė***
*Institute of Computer Science, Vilnius University*
*Vilnius, Lithuania* *austeja.bauraite@mif.vu.lt*

***Agnė Brilingaitė***
*Institute of Computer Science, Vilnius University*
*Vilnius, Lithuania* *agne.brilingaite@mif.vu.lt*

***Linas Bukauskas***
*Institute of Computer Science, Vilnius University*
*Vilnius, Lithuania* *linas.bukauskas@mif.vu.lt*

**Abstract**

Practical hands-on exercises for trainees in information technologies and information systems provide tools needed to develop, operate, and test cloud-based infrastructures. Exercises, frequently carried out in simulated settings, offer a practical approach to highlight the significance of skills related to structured decision-making and detailed configuration in the command line. Therefore, the paper proposes a unique data analysis solution that encompasses safe sandboxing and examines user-provided command-line data throughout the exercises. The development of the solution focuses on command-line input parsing, tokenization, and structured analysis, providing a viewpoint on the knowledge level in simulated scenarios. The work provides insight into structured command-line data analysis and the complexities of command execution. The prototype is based on modular *Bourne-Again Shell* and *Python* modules and asynchronous data collection. The paper contributes to the educational processes by improving training performance assessment techniques and offering insights into the exemplified field of penetration testing of information systems.

**Keywords:** educational system design, hands-on exercises, performance evaluation, data-driven assessment, command-line

## 1. Introduction

A rapidly evolving cybersecurity domain impacts a vast sector of information technologies (IT) and information systems (IS). IT specialists take measures to keep systems resilient against cyber incidents and various threats. IS specialists require skills to design, develop, and support systems and processes that involve the cybersecurity dimension. Thus, comprehensive training of IT and IS specialists is crucial. As cyber threats become more sophisticated, robust training, which encompasses self-learning, instructing others, and continuous skill development, becomes vital. Hands-on exercises bridge the gap between theoretical concepts and their application in real-world scenarios, offering a dynamic and interactive method of education.

Training activities range from academic education to training programs to enhance cybersecurity awareness, influence user behaviour, and develop a culture of cybersecurity [2]. Efficient training must be relevant, current, and tailored to the needs of participants, creating simulations for these situations to provide an environment where they can improve their skills. This learning is indispensable for grasping cybersecurity's intricate and changing nature, putting theoretical knowledge to the test in diverse circumstances.

Training exercises focusing on command-line proficiency are crucial in developing a more

advanced skill set [13]. They allow learners to apply academic knowledge in various simulated scenarios that closely resemble actual cyber-threat landscapes [19]. Such hands-on experiences are invaluable in comprehending the complexities of cybersecurity, where learners are challenged to adapt and apply their expertise. IT and IS specialists must employ command-line skills to perform system penetration testing, mitigate vulnerabilities, or tune-up system configuration. The role of assessment modules in training and educational programmes is critical [4]. Additionally, employing IT and AI-ready tools provides a structured approach to understanding learning outcomes, identifying areas for improvement, and tailoring future training to tackle existing and evolving challenges [8]. The data from cybersecurity exercises offers rich insights for learners and educators, enabling feedback that enhances the learning experience [9].

The contribution of this paper is to establish novel methods to analyse command-line-based user activity to identify user experience levels and assess competences during training without prior knowledge and pre-testing. Specifically, the novelty is focused on formalisation, evaluation criteria, and metrics to derive trainee expertise. This work outlines the method from understanding the command line structure to developing the solution and its operational methods, aiming to create a prototype for analysis and transform data into insights. The paper discusses hands-on cybersecurity exercises, command-line interface complexities, the design of the data analysis tool, its technical implementation, and the results gathered from testing with the openly available data set.

The rest of the paper is structured as follows. Section 2 sets the research background, and Section 3 provides problem setting. The design of the proposed solution is covered in Section 4. Section 5 presents the experimental setup of the proof of concept implementation. Section 6 provides experimentation results and discusses assessment viewpoints. Section 7 concludes the paper, outlines limitations and reviews the future work.

## 2. Background

Penetration testing [17], as a part of the cybersecurity area, assists organisations in security compliance and includes finding vulnerabilities and their mitigation. Typically, professionals advance their skills to tackle cyber threats, mirroring real-world challenges, during intensive hands-on training sessions called cyber defence exercises (CDX). CDXs offer technical, hands-on experience in active defence against simulated incidents, testing participants' technical expertise and response agility [10]. Participants engage in CDX roles across different teams in a cyber range environment, simulating real-world IT infrastructures to prepare for a spectrum of cybersecurity challenges, such as crisis situations during cyber incident handling [12].

The command-line interface (CLI) plays a pivotal role [11] in exercises, demanding specific command inputs and a deep understanding of systems for tasks like network configuration and penetration testing. CLI's efficiency, especially for experienced users, is explored by Voronkov, Martucci, and Lindskog [15], highlighting its preference among system administrators for certain tasks over graphical user interfaces (GUIs). CLI allows for scripting and command piping, enhancing task performance efficiency and requiring fewer system resources, making it ideal for high-performance computing environments. Moreover, CLI-based training in cybersecurity is instrumental in improving problem-solving skills and adaptability. CLI command learning can also be specialised in the use of one or more commands, e.g. for learning the git tool or networking scenarios [3]. Švábenský et al. [20] demonstrate how hands-on exercises that require quick and innovative thinking enhance professionals' analytical capabilities. Although the CLI may present a steep learning curve for beginners, its precision, flexibility, and efficiency are unmatched for IT professionals, developers, and system administrators, making it an essential tool for complex tasks or automating repetitive ones in computing environments. This hands-on experience with the CLI significantly enhances the learning process, preparing students for future developments in this technical area and allowing educators to adjust their teaching strategies for

improved training effectiveness [7, 19].

Maennel [9] covers the application of learning analytics in cybersecurity training, focusing on cybersecurity exercises to enhance learning effectiveness. Learning analytics and learning metrics provide evidence of the learning process and learning improvements. Online cybersecurity exercises are a powerful tool to teach cybersecurity content, but it has a challenge related to the individual assessment [6].

Analysis of CLI artefacts includes string tokenization and context analyses. Trizna [14] introduces the Shell Language Preprocessing (SLP) library, which focuses on the tokenization and encoding of Unix and Linux Shell commands. It outlines the limitations of traditional NLP methods in this context and proposes alternative encoding strategies such as label, one-hot, and TF-IDF encodings, implemented through bashlex and additional syntactic logic for complex command syntax handling. Analysis of command similarities [5] enables pattern recognition and its application to attribute actions to the adversary. Structured features of the Shell command can be used to identify malicious commands [1].

Quantitative and qualitative data analyses of CLI can identify the efficacy of command-line tool usage in cybersecurity training simulations. Švábenský et al. [19] focused on the frequency and types of commands employed by trainees, with a particular emphasis on tools like *nmap*, *ls*, and *cd*. The study further explored terminal command usage, identifying common errors and misconceptions among trainees. However, they did not analyse the variability of commands in terms of parameter or option usage.

As users may track their progress over time, see trends in their errors, and modify their learning tactics accordingly, error analysis of this kind encourages self-improvement and independent learning in a self-learning environment. The relationship between user error and cognition is examined by Wu et al. [16]. In order to create more user-friendly user interfaces and instructional materials, it is imperative that user errors must be understood from a cognitive point of view, as the study highlights. It also highlights that distinct user groups, such as novice, ordinary, and expert, show differing error rates.

This paper presents a solution for analysing user activities at the command structure level in the CLI environment to ensure individual assessment in online training. Analysis of correct and erroneous commands and result aggregation in the group context enables evaluation of the individual performance for the defined tasks.

## 3. Problem Setting

The most common interface for security developers and operators to work with is the feature-rich CLI. Therefore, a systematic approach to freely executed sequences of CLI commands and the ability to analyse the richness and correctness of the command is essential. The assessment of trainee actions in a CLI-based environment requires specialised tools to be involved.

The CLI operates through a sequence of crafted steps. It begins with tokenization, where the input command is dissected into smaller tokens. Parsing is the next stage, where these tokens are checked against the CLI's grammar rules. In Shell, data can be read from the file, string, or user's terminal. After the CLI breaks down the command into smaller parts (tokenization) and checks them (parsing), it runs the command based on this perception. The CLI systematically handles inputs, particularly those commands that are complex, involving several steps or require certain conditions to be met before they can run. Shell processes can be redirected to streams. After execution, Shell processes the exit status of the command, which determines whether the command was successful or if an error occurred. By breaking down commands to their fundamental components and interpreting their syntax, the CLI gains an ability to analyse command structures and usage patterns. This depth of analysis is valuable for both educators and practitioners, offering deeper insights into the nuances of command-line interactions.

Lexical analysis breaks down an input into words and phrases, known as tokens. Each token

carries a specific meaning. Technically, tokens surround everything from the commands themselves to various options and arguments. The role of the tokenizer is to categorise each segment of the command accordingly. Tokenization can be represented using the formal constructs.

Each command is a string $s \in S$ composed of a sequence of characters $s = \langle c_1 c_2 c_3 \dots c_n \rangle$, where each $c_i \in CH$ represents a character in the command, and set $S$ denotes a set of strings. The tokenization function $tf$ maps the string $s \in S$ to a sequence of tokens $T = \langle t | \forall t \in S \wedge \exists s' = \sqcup t : s' = s \rangle \subset S$. A token set is a subset of a string set, and concatenation of tokens can make the original string $s$.

Meta-characters play a crucial role in the tokenization and influence command parsing process. A set of meta-characters, $M \subset CH$, includes delimiters such as a pipe, ampersand, semicolon, relational operators, parentheses, spaces, tabs, and newlines. The presence of the meta-characters in the command string $s$ determines the boundaries of each token. Each token $t \in T$ is classified based on its composition. A *word* is a token without any characters from $M$ and not enclosed in quotes. An *operator* contains at least one character from $M$ and is unquoted. Operators are categorised into control and redirection types, denoted as $C$ and $R$, respectively. Set $C$ includes control operators such as a newline, a vertical pipe, and parentheses. Set $R$ includes redirection operators such as '<<'.

Let us consider a complex *Nmap* command represented as a string $s$ = "*nmap -p 80,443 --open -sV example.com*". The tokenizer identifies *nmap* as the foundational command token. Then, it separates the options *-p*, *--open*, and *-sV*, alongside with *80,443* and *example.com* as arguments. The tokenization function $tf$ applied to the command string $s$ returns a token sequence, $T$=⟨"nmap", "-p", "80,443", "--open", "-sV", "domain-example.com"⟩. Fig. 1 illustrates the tokenization result and context information. As illustrated in this example, the tokenization

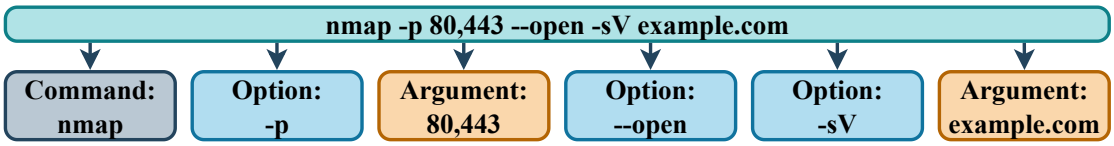



**Fig. 1.** Tokenization of the Nmap command

successfully breaks down the command into individual components. However, these components, or tokens, lack meaning when isolated. Only within the correct context that the meaning of each token can become clear. For instance, the token *-p* in isolation is ambiguous, because its purpose is only clarified when considered in conjunction with the base *nmap* command, where it specifies the ports to scan. Similarly, the token $80, 443$ is a basic string of characters until it is understood as the argument for the *-p* option, specifying the port range for the *nmap* command. The tokenization phase, while essential for parsing the command, does not bring semantic comprehension. This is derived from the following stages, where the syntactic structure is interpreted, and the command is executed.

Parsing, in the context of command analysis, refers to the process of analysing a sequence of tokens to discern their syntactical and hierarchical relationships within the command structure. It is about making sense of how different parts of a command fit together to form a coherent instruction, which when executed produces wanted results. The primary goal of parsing would be to derive the syntactic structure of the command. This includes identifying the command name, its options, arguments, and how they are logically connected.

When analysing the command line, familiarising with different delimiters used in CLI are important, the role of flags as boolean options to indicate true or false states, and the significance of the *--help* flag in providing detailed command information. Additionally, the CLI conventions to represent required and optional parameters, handle arguments that accept multiple values, and the syntax for mutually exclusive arguments are vital for comprehensive syntax analysis. In the

context of Shell command line processing, the parsing function, denoted as $P$, transforms a sequence of tokens into a structured representation of the command.

Here, each token $t_i \in T$ represents a distinct element of the original command. The function $P$ meticulously maps these tokens to their corresponding roles in the command structure. The CommandName is the first token $t_1$, while OptionsList and ArgumentsList are formulated from subsequent tokens based on their syntactical and contextual properties. Additionally, special tokens indicating pipes and redirections are categorized into RedirectionOperator and ControlOperator, respectively. Pipes and Redirections, integral to command execution flow, are identified by specific Meta-characters. These elements, besides the first element, can have variable order and presence in the command structure, reflecting the flexible nature of Shell command syntax. This mapping ensures that the sequence and hierarchical relationship of the tokens are preserved, leading to an accurate interpretation of the command as a whole unit. Options in a Shell command, signified by '-' or '--', are often paired with their subsequent Arguments. Arguments are tokens that do not qualify as Options or Meta-characters. The mapping function $F_{\text{oa}}$ is defined to identify these pairs (see Eq. 1.) Applying $F_{\text{oa}}$ to the command tokens, option-argument pairs are constructed, Pairs $= \{(t_i, F_{\text{oa}}(t_i)) \mid t_i \in \text{Options}\}$.

$$F_{\text{oa}}(t_i) = \begin{cases} t_{i+1} & \text{if } t_{i+1} \notin \text{Options} \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

In the case of specific commands like *nmap*, where the last token often represents the target of the scan, a modified option-argument mapping function $F_{\text{oa}}^{nmap}$ is defined as presented in Eq. 2. Applying $F_{\text{oa}}^{nmap}$ to *nmap* command tokens, it identifies both option-argument pairs and standalone arguments: Pairs$^{nmap} = \{(t_i, F_{\text{oa}}^{nmap}(t_i)) \mid t_i \in \text{Options}\}$ and StandaloneArgument $= t_m$ if $t_m \notin$ Pairs$^{nmap}$.

$$F_{\text{oa}}^{nmap}(t_i) = \begin{cases} t_{i+1} & \text{if } t_{i+1} \notin \text{Options} \wedge i+1 < m \\ \emptyset & \text{if } i+1 = m \vee t_{i+1} \in \text{Options} \end{cases} \tag{2}$$

Let us consider the previous nmap command to illustrate this parsing method. Applying the parsing function $P$, the following structure can be obtained: CommandName is *nmap*, Options $=$ {"-p", "–open", "-sV"}, Pairs$^{nmap}$ $=$ {("-p", "80,443"), ("--open", $\emptyset$), ("-sV", $\emptyset$)} *example.com* is a StandaloneArgument, and Pipes and Redirections are empty sets.

Statistical methods can be used to analyse the frequency and patterns of command usage (Command Structure Analysis; CSA.) For instance, using a counter to count the occurrences, can provide specific patterns like *Command-Option* or *Command-Argument-Option*. A high frequency of *Command-Argument-Option* might indicate a preference for commands with specific configurations, suggesting a user's skill level or the complexity of tasks they are performing. A CSA function $sf$ maps each sequence of tokens $T$ to a command structure $cs \in CS$, where $CS$ is the set of all the possible command structures. A command structure can be defined as a sequence of token types representing the syntactic structure of a command. Each command structure $cs$ is a sequence of elements $cs = \langle t_1, t_2, t_3, \ldots, t_m \rangle$ where each $t_i$ corresponds to a type of token (such as Command, Option, Argument, etc.). The frequency of each command structure $cs$ in the dataset is calculated using the function $F(cs)$, defined as the sum of occurrences of $cs$ for each command string $s \in S$: $F(cs) = \sum_{s \in S}[sf(tf(s)) = cs]$. Here, $[sf(tf(s)) = cs]$ this function checks whether the structure identified by $sf(tf(s))$ is equal to a particular command structure $cs$ and if they are the same, the function returns 1, otherwise, it returns a 0. The formula counts how many times each unique command structure appears in the dataset. The unique command structure function $usf$ maps each tokenized command to its unique structural pattern. The frequency function $F_{ucs}$ counts the occurrences of each unique command structure within the dataset. This analysis provides insights into the complexity and variety of command usage, highlighting common patterns or unusual combinations in

command structures for individual commands. By identifying the most frequently used options and arguments, it can be defined, which features of a command are most valued or required by users in specific exercises. The analysis of metacharacters usage suggests the users' proficiency and sophistication level in command-line operations. Frequent use of complex metacharacters could indicate a high level of user expertise. Additionally, this analysis could aid in uncovering patterns or anomalies in command usage. Unusually high or low usage of certain options or arguments might suggest either a lack of awareness about these features or their irrelevance to the users' needs. In security-sensitive environments, these anomalous patterns could also be an indication of misuse, potentially flagging security concerns.

For any given command $s$, the function $cef(tf(s))$ provides the frequency distribution of elements in $s$, therefore, linking a tokenized command to its element frequencies. To be able to extend this analysis across all commands in the dataset, a frequency function $F_{all}$ is defined: $F_{all}(e) = \sum_{s \in S} n$ where $(e, n) \in cef(tf(s))$. This general function aggregates the frequencies of each element $e$ across the entire dataset, which then provides a comprehensive insights into user interactions with command-line elements. Such an analysis is valuable for understanding user behaviour patterns, preferences in particular command usage. The average complexity $K$ for a set of commands by the user can be calculated as follows: $\bar{K} = \frac{1}{M} \sum_{i=1}^{n} m_i K(c_i)$, where $c_i$ is a command from the set of all certain command structures, $K(c_i)$ is a command complexity (based on the number of elements in a command), $m_i$ is a count of $c_i$ occurrences, and $M$ is a total number of command usages. The complexity variation $V$, which measures the spread of these complexities, is the standard deviation of complexity: $V = \sqrt{\frac{1}{M} \sum_{i=1}^{n} m_i (K(c_i) - \bar{K})^2}$.

In a command line, error analysis is essential for identifying common mistakes users make. In order to provide insights into user behaviour, this analysis can focus on classifying and evaluating the success or failure of instructions that are executed in the command-line environment. Examination of unsuccessful commands, stemming from syntax errors, improper command usage, or a lack of understanding of the purpose of command functions, might reveal error patterns. This data is essential for creating more focused teaching materials that could address prevalent areas of misunderstanding.

## 4. Solution Design and Architecture

We propose the information system design to support cybersecurity-related education and data-driven assessment of the trainee, either an IT or IS specialist. The proposed solution is a heterogeneous environment (see Fig. 2). As an actor, a *Course instructor*, seen on the far right, designs

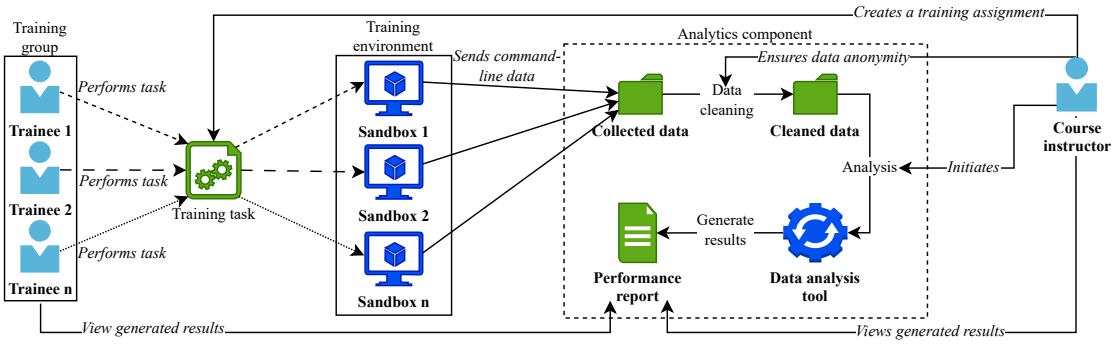



**Fig. 2.** Information system solution design

a *Training task* that requires the execution of CLI commands. The *Trainee*, seen on the left side of the diagram within the *Training group*, performs the task in an individual training *Sandbox* that is a part of the *Training environment*. There, the data is collected and sent to the *Analytics*

*component*. Once CLI-based data is collected, the instructor proceeds to the next steps. Within the *Analytics component* section, the *Course instructor* begins with anonymisation to ensure the private data is not disclosed, for example, private keys and possible plain-text passwords, i.e., performs *Data cleaning*. Then, the instructor launches the *Analysis* process. The final processes of the data analysis tool apply the predefined metrics on the collected and cleaned CLI data and generate trainee performance results, seen in a figure as a *Performance report*, in a structured format that can be used to examine results within the group context. The trainee and instructor can view the generated results.

The execution architecture of the solution (see Fig. 3) uses dual virtual machine configuration to separate and manage two fundamental processes, CLI usage pattern extraction and Shell command execution, in the *Data analysis tool* of the *Analytics component*. *The Main Virtual Machine*, seen on the left side of the figure, running as a Linux host, is responsible for data handling tasks, i.e., accepting and organising collected data. The main scripts of the component *Command-line Data Tool* orchestrate the entire process. They control the intake and parsing of data (see *Command-line Data*) and manage workflows to save and retrieve patterns (see database *Commands*), check command correctness, execute analytical tasks, and save the analysis results (see *Generated Results*). The *Sandbox Virtual Machine*, seen on the right side of the figure, is employed for error checking in the usage pattern extraction process. It enables the execution of commands in an isolated environment and returns the execution output to the main machine. Ideally, the sandbox is Kali or Parrot Linux OS distribution equipped with core utils and penetration and security testing tools.

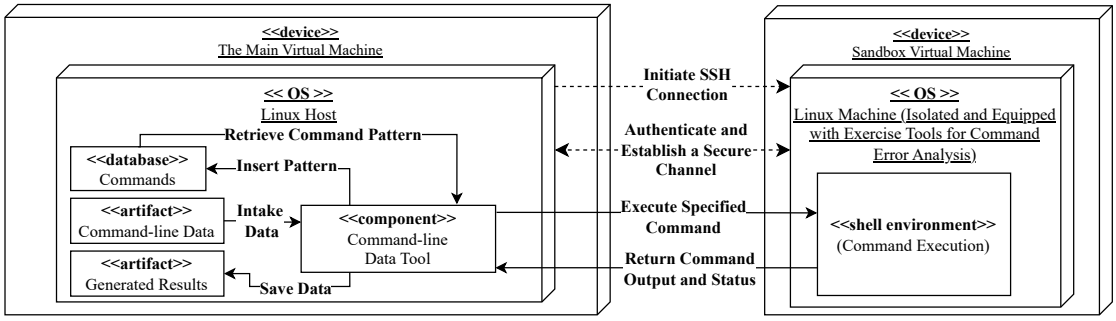



**Fig. 3.** Execution architecture of the solution's analytics component

Fig. 4 presents the relational model of the database component and illustrates the tokenized CLI command data insertion. Table CommandSequences stores an original CLI command. The output of its execution in the sandbox is recorded in CommandExecutionLog. Table CommandElements records the order of options and arguments. Each command and its options and arguments are stored in dedicated tables (see tables at the right). Therefore, before inserting new arguments and options, the system checks already saved elements. This design enables analysis of the options and arguments among all commands within the trainee group.

Fig. 5 presents the flow of the error-checking logic and command pattern extraction for an individual trainee. Firstly, the command is checked for the base exception. Configuration files define command types to skip, e.g., Python script execution. If such a command is detected, the system raises a base exception. Then, the tool sanitises the commands from the outbound file or network operations, such as switching IP addresses to localhost for scanning tools, and checks if it is already in the database (found in the logs of other trainees or previously detected in the log of this trainee). If the command is not in the database, the error-checking module sends it to the sandbox for execution. The performance report counts the trainee's errors explicitly.

The main virtual machine's isolation from the execution environment mitigates risks of system compromise, ensuring the integrity and security of the central processing machine. The

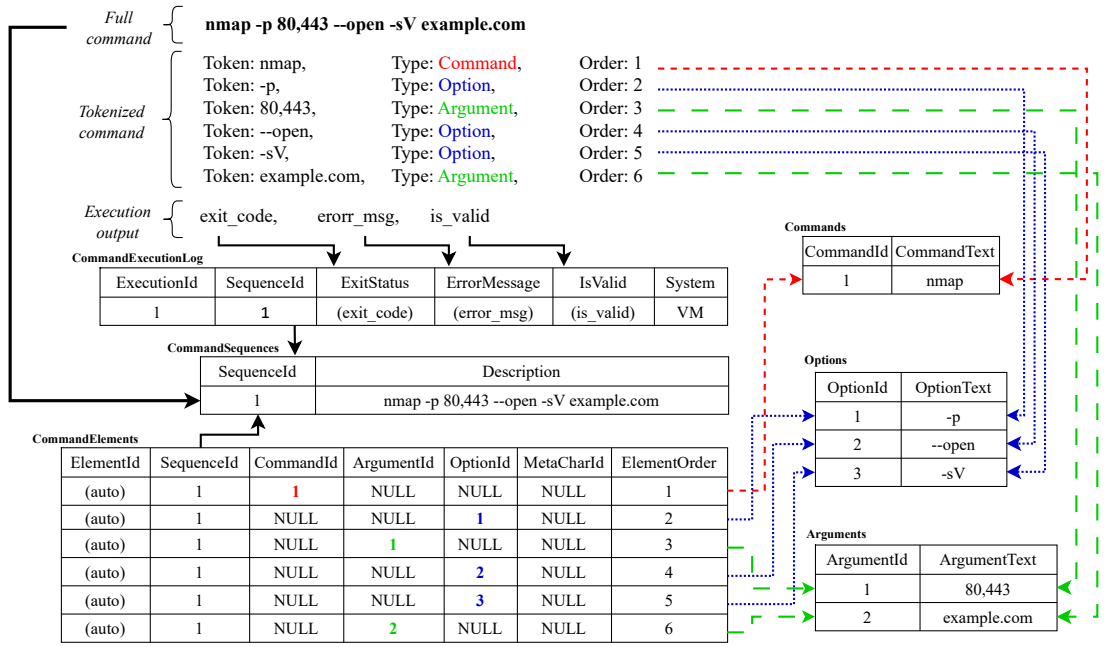



**Fig. 4.** Relational model and data insertion example

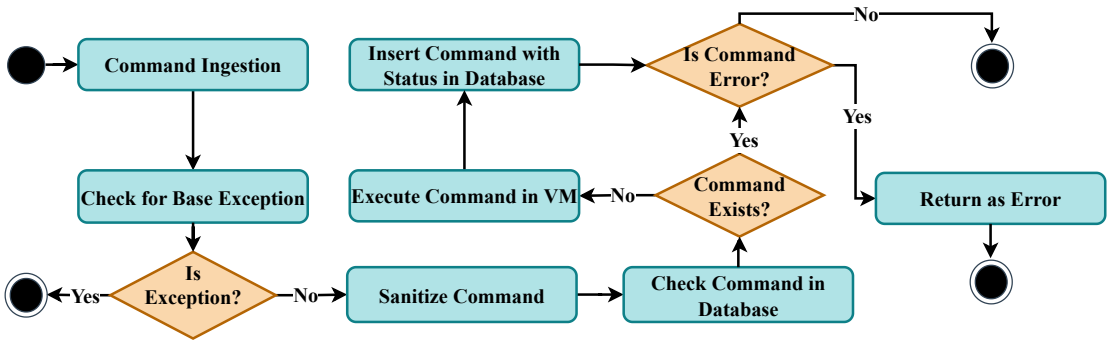



**Fig. 5.** The flowchart of the error-checking logic

sandbox virtual machine serves as a controlled testing ground. It replicates or closely resembles the trainee's operating environment, ensuring that Shell commands are executed in conditions equivalent to their original setting.

## 5. Experimental Setup

The sandbox prototype was implemented as a proof of concept for the proposed solution. The prototype reflects the execution architecture (see Fig. 3) of the solution for the experimental setup. OpenNebula IaaS with KVM hypervisor was used to create two virtual machines. The Main Virtual Machine, built using a Debian OS template, had the developed tool installed with the corresponding SQLite3 database and the downloaded dataset. The tool comprised Bash and Python scripts. Bash scripts orchestrated the flow of the tool. They performed remote Bash execution, while Python scripts performed data loading, tokenization, parsing, and the analysis part. The Sandbox Virtual Machine was used as a separate environment to execute sanitised CLI commands over SSH connection for error analysis. Therefore, the machine was equipped with main exercise tools to test these commands and return their output to the central machine analysis tool.

We used the open dataset [18] to execute the simulated experiments to test our analytical components. The dataset consists of 13,446 Shell commands from 175 individuals participating in cybersecurity training. The dataset contains several subsets associated with different tasks. We selected the "Junior Hacker Adaptive" subset for experimentation because it contained the most significant number of participants and commands. It included 58 participants who executed 12–148 commands to familiarise themselves with the environment, scan the network, connect to the server, find and retrieve the required file, and crack its password.

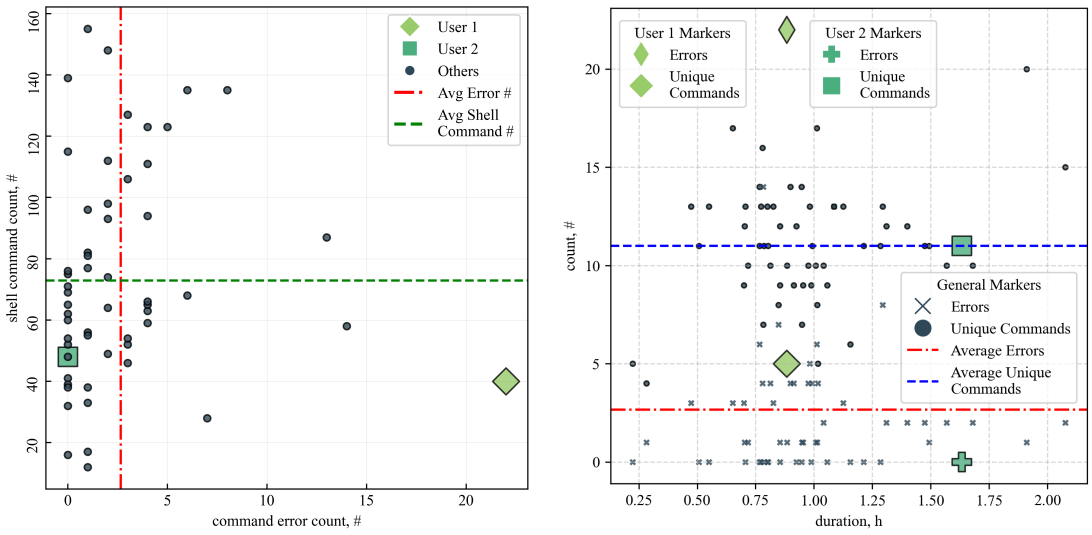
The dataset offers realistic content of user interactions in a cybersecurity learning context. The data was gathered during actual training activities using a transparent, open-source logging system, accurately showing command-line operations typical in cybersecurity, such as penetration testing. The dataset includes a variety of commands used in Bash, ZSH, and Metasploit shells, all stored in a JSON format, making it a valuable tool for diverse research activities. Each sandbox file is formatted using newline delimited JSON.

The provided JSON files represent recorded user actions in a cybersecurity training environment. Each entry in each dataset file includes attributes like `cmd_type`, indicating the nature of the command executed (e.g., Bash command), and `pool_id` and `sandbox_id`, which are identifiers for the specific training environment. The `timestamp_str` attribute timestamps each action, allowing for temporal analysis of user activities. The `hostname` and `username` fields provide context on the system and user account from which the trainee executed the commands, and the fields are vital to follow access levels and system interactions. The `cmd` attribute records the exact command executed, offering insights into the user's primary focus and command-line efficiency. Additionally, the `wd` (working directory) and `ip` (IP address) fields provide further context, revealing the user's navigation patterns and network interactions.

Some participants performed tasks with long breaks, sometimes lasting days. The long breaks, more extended than 1800 seconds, were subtracted from the total task duration time to get a more concentrated view of time. Thus, the duration of task solving included only shorter breaks as part of the training process. In the configuration file of the prototypes' error-checking module, the base exception list included the execution of specific console-based commands that open environments or require user input to operate or stop the processes, i.e. interactive commands such as terminal editors (vim), password changing (passwd) or device availability utilities (ping), and Bash or Python scripts, as such data is not logged and cannot be processed or analysed.

## 6. Results and Discussion

The experiments were executed on the open dataset to demonstrate the proof of concept that it can perform lexical and syntax analysis of the logged CLI calls, produce structured output, and generate visual reports as performance indicators of an individual (or several individuals) in the context of the group results. The compiled insightful diagrams revealed key patterns and improvement areas based on the commands the trainees executed. This interpretation of the results is based exclusively on the data from a single dataset, without prior knowledge of the trainees and their level of expertise. Consequently, trainee performance during this training session determines the evaluation. In each figure of this paper, visual markers are used to distinguish particular trainees to illustrate and track individual performance indicators: diamonds represent User 1, and squares represent User 2. These markers visually *represent how these participants interact with the task and tools compared to the average team performance*. The figures also show the distribution of individual results within a team. Fig. 6a plots the number of CLI commands against the number of errors, allowing the accuracy of command usage to be assessed within the average group performance. It highlights that higher command usage does not necessarily lead to more errors, *indicating the participants' learning curve*. For example, User 1 executed almost the same number of commands as User 2 but experienced much more errors. Both users executed fewer commands than the average of executions within a group.



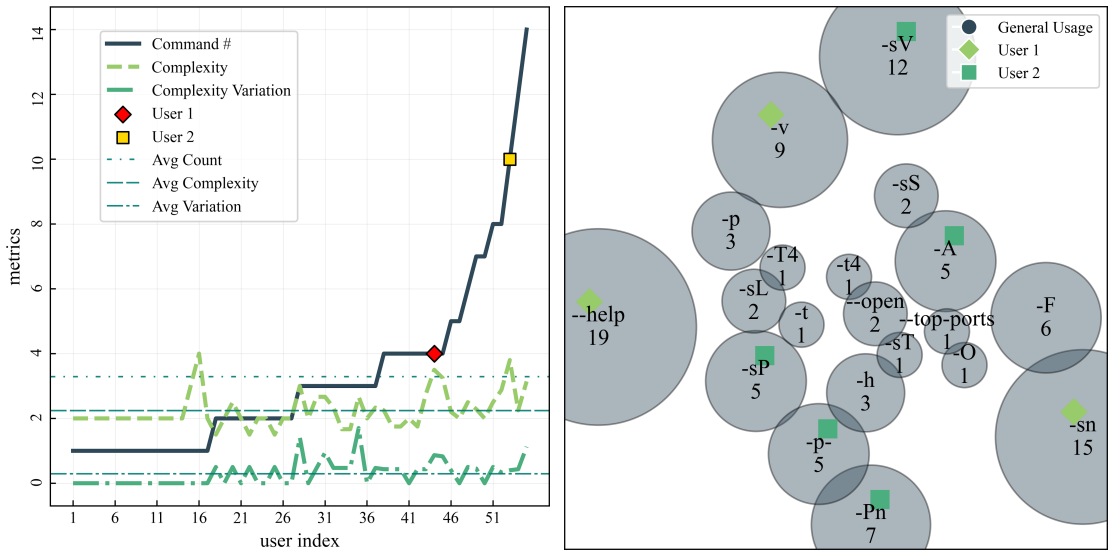

**(a)** CLI command count vs. command error count

**(b)** Task duration vs. error count vs. unique command count

**Fig. 6.** Error count alignment with execution duration and CLI command count

Fig. 6b aligns the task duration with error counts and unique command counts to *gain perspective on the performance of the unique tool usage based on task completion time and its correlation with command line inaccuracy*. It demonstrates that longer duration does not necessarily correlate with error frequency, as a bigger part of participants with longer task execution time managed to stay below average error count. For example, User 1 worked on the task longer than User 2, using fewer unique commands but experiencing more errors. Another important observation is the relationship between unique command usage and errors. Sessions with a higher variety of commands did not always lead to more errors. This suggests that *proficiency in command usage and familiarity with tools can significantly impact performance outcomes*.

Fig. 7a highlights the usage of the Nmap tool, charting the count, complexity, and variation of command structures. This view can help indicate the trainee's familiarity with the specific tool




**(a)** Nmap command usage analysis **(b)** Nmap option usage

**Fig. 7.** Analysis of Nmap command usage

and how well the trainee can use it with minimal effort to complete the average group indicators. The figure orders the users according to the command count. High command complexity among certain users suggests a more advanced level, and their moderate complexity variation means consistency in their command execution (see equations in Sec. 3.) *High command complexity variations might indicate experimental approaches.* Beginners with consistently low complexity scores and minimal variations indicate less experience; this could also reflect a more cautious approach to command usage, potentially avoiding errors. Fig. 7a indicates complexities and overall tool usage alongside the application of specific Nmap options presented in Fig. 7b. This view demonstrates the main trends of tool functionality usage. In this case, it shows the most used and underutilised Nmap features. For example, User 1 used parameters applied by many peers. In contrast, User 2 used some parameters that were not common within a group.

Fig. 6–7 work as a tool for the instructor to indicate the skill level of the particular trainee in the assessment process. Based on the figures, User 1 struggled with command execution, as seen by a high error rate in the CLI command analysis. The trainee executed 40 commands, but 22 errors indicate a substantial error rate of 55%. This result suggests a lack of familiarity with the commands and the system. The frequent usage of the --help option in Nmap and the most common option usage further supports the hypothesis that User 1 is in a learning phase, potentially lacking knowledge of some commands. In contrast, User 2 displayed superior command execution skills, executing 48 commands with no errors, indicating high proficiency and confidence in handling hands-on tasks. Using Nmap with complex options like -A for aggressive scanning and -sV for service version detection showcases a strategic and thorough approach. These options are critical for in-depth system analysis, showing that User 2 can handle these commands effectively without errors.

Overall, this detailed analysis not only highlights the varying skill levels and areas for improvement among the participants but also shows the importance of personalised and group training programs, which could tackle arising issues and enhance training efficiency. In addition, *certain users can be identified as potentially taking on more complex tasks to further engage in more challenging activities*, while others might need more basic training to get to a certain level.

## 7. Conclusions, Limitations, and Future Work

The professional development of IT and IS specialists includes training using hands-on exercises. However, the training process should be part of the educational system that employs tools for semi-automated and AI-ready assessment and evaluation of performance proficiency. The proof of concept, implemented according to the design of the analytics component of the proposed solution and executed with an open dataset, demonstrated individual performance assessment based on command execution patterns, frequency of errors, and task duration. The performance report offered insights into the trainee's confidence and familiarity with tasks. Diverse command structures, from basic Command-Argument to complex sequences, suggested varying technical skill levels among participants. The community of educators benefits from the designed solution for *the formative assessment of the trainee, getting the report of command usage patterns, and aligning individual performance with a group and command usage patterns detected within the group*. Therefore, it could work as a tool to *increase training efficiency, ensure transparency, and shape training plans*.

There are potential limitations of this study. Firstly, using one specific dataset to gain insights into student performance could cover only part of the diverse possibilities of command usage. Then, the interpretation of the results may yield different insights depending on the type of exercise conducted and the competence of the course instructor. Furthermore, the data did not include the background information, e.g., what engagement elements the exercise included or if the system provided hints. Technical limitations could also pose challenges; for example, different CLI environments may require different parsing of command elements to identify their exact meaning within the command structure correctly. Finally, the proposed method identifies syntax errors but cannot detect logical ones.

Future improvements could enhance the tool's resilience, safety, and flexibility. More accurate identification and sanitisation of complex data types, advancing error detection, and accommodating different command execution environments would be crucial for a broader application spectrum. Additionally, incorporating AI tools for refined data pattern recognition, anomaly detection, and user behaviour forecasting could significantly enrich the tool's analytical capabilities. These advancements elevate the tool's utility in more complex scenarios and provide deeper, AI-driven insights into cybersecurity training and education.

## References

1. Chen, S., Yang, R., Zhang, H., Wu, H., Zheng, Y., Fu, X., Liu, Q.: Sifast: An efficient unix shell embedding framework for malicious detection. In: Athanasopoulos, E., Mennink, B. (eds.) Information Security. pp. 59–78. Springer Nature Switzerland (2023)
2. Chowdhury, N., Katsikas, S., Gkioulos, V.: Modeling effective cybersecurity training frameworks: A delphi method-based study. Computers & Security 113, pp. 102551 (2022)
3. Conte de Leon, D., Goes, C.E., Haney, M.A., Krings, A.W.: Adles: Specifying, deploying, and sharing hands-on cyber-exercises. Computers & Security 74, pp. 12–40 (2018)
4. Foster, C., Francis, P.: A systematic review on the deployment and effectiveness of data analytics in higher education to improve student outcomes. Assessment & Evaluation in Higher Education 45(6), pp. 822–841 (2020)
5. Hobert, K., Lim, C., Budiarto, E.: Semantic similarities for honeypot collected linux shell commands. In: 2023 11th International Conference on Information and Communication Technology (ICoICT). pp. 172–177 (2023)
6. Karjalainen, M., Kokkonen, T., Taari, N.: Key elements of on-line cyber security exercise and survey of learning during the on-line cyber security exercise. In: Lehto, M.,

Neittaanmäki, P. (eds.) Cyber Security: Critical Infrastructure Protection, pp. 43–57. Springer International Publishing (2022)
7. Kendon, T., Stephenson, B.: Unix literacy for first-year computer science students. In: Proceedings of the 21st Western Canadian Conference on Computing Education. WCCCE '16, ACM (2016)
8. Ley, T., Tammets, K., Pishtari, G., Chejara, P., Kasepalu, R., Khalil, M., Saar, M., Tuvi, I., Väljataga, T., Wasson, B.: Towards a partnership of teachers and intelligent learning technology: A systematic literature review of model-based learning analytics. Journal of Computer Assisted Learning 39(5), pp. 1397–1417 (2023)
9. Maennel, K.: Learning analytics perspective: Evidencing learning from digital datasets in cybersecurity exercises. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 27–36 (2020)
10. Maennel, K., Brilingaitė, A., Bukauskas, L., Juozapavičius, A., Knox, B.J., Lugo, R.G., Maennel, O., Majore, G., Sütterlin, S.: A multidimensional cyber defense exercise: Emphasis on emotional, social, and cognitive aspects. SAGE Open 13(1), pp. 21582440231156367 (2023)
11. Mirkovic, J., Aggarwal, A., Weinman, D., Lepe, P., Mache, J., Weiss, R.: Using terminal histories to monitor student progress on hands-on exercises. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. p. 866–872. SIGCSE '20, ACM (2020)
12. Seker, E., Ozbenli, H.: The concept of cyber defence exercises (CDX): Planning, execution, evaluation. In: 2018 International Conference on Cyber Security and Protection of Digital Services. pp. 1–9. IEEE (2018)
13. Singh, T.D., Khilji, A.F.U.R., Divyansha, Singh, A.V., Thokchom, S., Bandyopadhyay, S.: Predictive approaches for the unix command line: curating and exploiting domain knowledge in semantics deficit data. Multimedia Tools and Applications 80(6), pp. 9209–9229 (2021)
14. Trizna, D.: Shell language processing: Unix command parsing for machine learning (2022), arXiv
15. Voronkov, A., Martucci, L.A., Lindskog, S.: System administrators prefer command line interfaces, don't they? an exploratory study of firewall interfaces. In: Lipford, H.R. (ed.) Fifteenth Symposium on Usable Privacy and Security, SOUPS. USENIX Association (2019)
16. Wu, X., Huang, X., Xu, R., Yang, Q.: An experimental method study of user error classification in human-computer interface. Journal of Software 8, pp. 2890–2898 (11 2013)
17. Zabicki, R., Ellis, S.R.: Chapter 75 - penetration testing. In: Vacca, J.R. (ed.) Computer and Information Security Handbook, pp. 1031–1038. Morgan Kaufmann, Boston, third edition edn. (2017)
18. Švábenský, V., Vykopal, J., Seda, P., Čeleda, P.: Dataset of shell commands used by participants of hands-on cybersecurity training. Data in Brief 38, pp. 107398 (2021)
19. Švábenský, V., Vykopal, J., Tovarňák, D., Čeleda, P.: Toolset for collecting shell commands and its application in hands-on cybersecurity training. In: 2021 IEEE Frontiers in Education Conference (FIE). pp. 1–9 (2021)
20. Švábenský, V., Vykopal, J., Čeleda, P., Tkáčik, K., Popovič, D.: Student assessment in cybersecurity training automated by pattern mining and clustering. Education and Information Technologies 27(7), pp. 9231–9262 (2022)