Future Generation Computer Systems 180 (2026) 108392

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

RoWD: Automated rogue workload detector for HPC security

Francesco Antici [a,b,*], Jens Domke [a], Andrea Bartolini [b], Zeynep Kiziltan [b], Satoshi Matsuoka [a]

[a] Riken Center For Computational Science, Kobe, Japan
[b] University of Bologna, Bologna, Italy

ARTICLE INFO

Keywords:
High-performance computing
Machine learning
Artificial intelligence
System security

ABSTRACT

The increasing reliance on High-Performance Computing (HPC) systems to execute complex scientific and industrial workloads raises significant security concerns related to the misuse of HPC resources for unauthorized or malicious activities. Rogue job executions can threaten the integrity, confidentiality, and availability of HPC infrastructures. Given the scale and heterogeneity of HPC job submissions, manual or ad hoc monitoring is inadequate to effectively detect such misuse. Therefore, automated solutions capable of systematically analyzing job submissions are essential to detect rogue workloads. To address this challenge, we present RoWD (Rogue Workload Detector), the first framework for automated and systematic security screening of the HPC job-submission pipeline. RoWD is composed of modular plug-ins that classify different types of workloads and enable the detection of rogue jobs through the analysis of job scripts and associated metadata. We deploy RoWD on the Supercomputer Fugaku to classify AI workloads and release SCRIPT-AI, the first dataset of annotated job scripts labeled with workload characteristics. We evaluate RoWD on approximately 50K previously unseen jobs executed on Fugaku between 2021 and 2025. Our results show that RoWD accurately classifies AI jobs (achieving an F1 score of 95%), is robust against adversarial behavior, and incurs low runtime overhead, making it suitable for strengthening the security of HPC environments and for real-time deployment in production systems.

1. Introduction

High-Performance Computing (HPC) systems are essential for executing large-scale, complex computational workloads. However, their increasing use raises concerns about the potential misuse of HPC resources for unauthorized or malicious activities, such as illicit training of Artificial Intelligence (AI) models, cybercrime, or cryptocurrency mining [1]. Such activities can threaten the security, confidentiality, and availability of HPC infrastructures, ultimately compromising system integrity [2,3]. Consequently, security screening of HPC job executions (computational tasks running on the system) is critical to ensure compliance with institutional and legal policies, protect intellectual property, and maintain user trust.

HPC systems operate as shared environments in which thousands of jobs are continuously executed by users from diverse application domains. In this context, detecting and preventing rogue job executions is a non-trivial task. The scale and heterogeneity of job submissions make manual inspection or ad hoc monitoring impractical [4]. Therefore, automated and systematic solutions are required to identify job operations and verify their compliance with system policies, ensuring a secure and reliable HPC environment. A promising approach is the use of Machine Learning (ML) techniques to analyze either job scripts (Bash scripts specifying hardware requirements, setup, and executed operations) or job execution performance data. These methods have proven effective in characterizing job behavior and inferring the nature of executed workloads [1,5,6]. However, to date, no solution provides automated and systematic security screening of the HPC job-submission pipeline. To address this challenge, we make the following contributions:

- The formalization of RoWD (Rogue Workload Detector),[1] the first security-screening framework for the HPC job-submission workflow. RoWD is composed of modular plug-ins that act as detectors for different job types (e.g., AI workloads, cybercrime, or cryptomining jobs) and automatically and systematically scan job submissions to identify rogue executions.
- The deployment of RoWD on the Fugaku Supercomputer[2] to classify AI jobs. To achieve this, we create and release SCRIPT-AI, the first publicly available dataset[3] comprising job scripts enriched with fea-

[1] https://github.com/francescoantici/RoWD/
[2] https://www.fujitsu.com/global/about/innovation/fugaku/
[3] https://github.com/francescoantici/SCRIPT-AI

* Corresponding author.
E-mail addresses: francesco.antici@riken.jp (F. Antici), jens.domke@riken.jp (J. Domke), a.bartolini@unibo.it (A. Bartolini), zeynep.kiziltan@unibo.it (Z. Kiziltan), matsu@acm.org (S. Matsuoka).

https://doi.org/10.1016/j.future.2026.108392
Received 16 October 2025; Received in revised form 7 January 2026; Accepted 16 January 2026
Available online 22 January 2026

tures describing the job workload (i.e., the *application* and its *domain*) and specifics (i.e., the *job scheduler software* and the *programming language*), and whether the job involves AI operations.
- Experimental evaluation of RoWD on approximately 50K previously unseen jobs executed on Fugaku between 2021 and 2025, showing accurate classification (achieving an F1 score of 95%), robustness against adversarial behavior, and low runtime overhead, making it suitable for strengthening the security of HPC environments and for real-time deployment in production systems.

For the Fugaku deployment, we focus on the classification of AI jobs, following a specific request from researchers at RIKEN, where Fugaku is hosted. However, RoWD can be readily configured to classify and detect other types of malicious or rogue activities, such as cybercrime or cryptomining workloads, as well as emerging threats. Due to the lack of labeled datasets for these activity types, we limit our evaluation to AI job classification.

In the rest of the paper, after we discuss the related work in Section 2, we present RoWD in Section 3 and detail its deployment on Fugaku in Section 4, where we also describe the SCRIPT-AI dataset. We experimentally evaluate RoWD on Fugaku in Section 5. Before we conclude in Section 7, we discuss the limitations of our approach in Section 6.

2. Related work

Several past works tackled the characterization of jobs' applications in HPC systems. In this context, two classes of solutions emerged. The first focuses on classification through the analysis of the time-series of the performance metrics of the job execution, as in [7,8], while the second using job code, as in [2,5,6]. The former class of techniques presents two main problems. First, they require a real-time monitoring software which collects the time-series of the performance counters with high resolution (e.g. every second or 100 milli-seconds), during all job executions. As explained in [9], such an infrastructure can incur a significant overhead on the system operations, and it is not a standard in production HPC systems. Indeed, a common practice is collecting aggregated (single values and not time-series) performance counters per job execution, as shown in [9], which are extracted at the time of job completion from node sensors (by knowing the start and end-time of the job and the allocated nodes). Second, such data can be collected only after the end of job execution [9], thus ruling out the possibility of preventing a possible rogue job execution. Due to these reasons, we consider this approach not suited for our purposes and limit the discussion of related work to the second class of techniques.

The approach of [2] relies on a Random Forest (RF) model trained on fuzzy hashes of the job binaries, a type of data which is not always obtainable (e.g. when the job uses an interpreted language) and which is bound to the system architecture, making the tool specific to a system. Instead, our approach is system agnostic, as it works with the job script. We show in Section 5 that a fuzzy hash approach on job scripts is not suited for our task, and it is outperformed by RoWD when classifying AI jobs. In [6], the authors compute job similarity based on an ML model which analyzes the CFG (Control-Flow Graph) of the job code and different graph similarity measures. As explained in [2], the extraction of the job CFG is a complicated reverse engineering effort, and it can incur a significant overhead on the system operations when performed systematically on several jobs. The work of [5] proposes the use of cryptographic hash to classify jobs into applications, with the aim of predicting job power consumption.

The main differences between our work and the previous studies can be summarized as follows. First, the prediction tasks are different: our approach is the first to classify rogue jobs involving AI operations in their workload, while the cited approaches all focus on the characterization of the job application. Nevertheless, our approach can be easily configured to predict also the job application (we provide this capability in the setup code for RoWD), as such information is already present in SCRIPT-AI. Second, we are the first to publish the dataset used for model training, and to provide a public and reproducible tool. The other approaches rely on unpublished private datasets, making the solutions not reproducible in other systems. Third, while RoWD is trained on a publicly available dataset (SCRIPT-AI), we validate its robustness and generality on an unrelated dataset extracted from a production supercomputer, obtaining an accuracy of 95%. In the other works, since training and testing data originate from the same system, robustness and generality of the results are not clear.

3. RoWD framework

In this section, we describe the functioning and setup of RoWD. The framework is designed to work in an *online* context, where the job data are live and streaming in time. RoWD has no implementation requirements from the system, apart from a software which is able to collect the job data (e.g. job scripts or aggregated *performance metrics*). Fig. 1 shows the high-level functioning of RoWD. The framework takes as input a job script, and it is composed of several software components for rogue job detection. In the figure, the blocks represent the software components, while the lines represent the actions. The solid lines show the workflow of the framework, explaining the input/output of each component and the corresponding data movement. Conversely, the dotted lines refer to the feedback mechanisms which are not directly dependent on the framework workflow, rather on external actions, such as the system administrator validating the predictions to update the classifiers.

RoWD is designed to improve the security of the job submission and execution pipeline of an HPC system, by mitigating possible threats. To this end, we define a threat a model (following the STRIDE methodology [10]) that identifies four threat categories:

- Spoofing: a malicious user obtains another whitelisted user's credentials to execute a non-authorized workload.
- Tampering: a malicious user uses the system resources and data to execute an unauthorized workload.
- Repudiation: a malicious user disguises its activities by masking the job script and results, or by declaring compliance with a certain project.
- Information Disclosure: A user may obtain, leak, or leverage sensitive information for malicious intents.

With RoWD, we are able to mitigate the tampering, repudiation and information disclosure threats. The handling of the spoofing threat is outside the scope of RoWD, and it is discussed in Section 6.

3.1. Framework components

Here we detail the framework components, namely the *Script Preprocessor*, the *Script Encoder*, the *Submission-time Classifiers*, the *Rogue Detector*, the *Execution-time Classifiers* and the *RoWD Logs*. Both classification components serve the same purpose, however they perform the classification using different job data (i.e., submission-time data and execution-time data, respectively), and at different steps of the job execution (i.e., at submission time and completion time, respectively). Both methodologies are useful to achieve accurate classifications, and depending on the task and the type of data available in a system, one may obtain better classification performance than the other. Moreover, the *Execution-time Classifiers* can be instrumental to correct the misclassifications of the *Submission-time Classifiers*, as the classifiers can leverage additional information on the job execution (e.g. aggregated *performance metrics*, *power consumption*, *duration*, etc).

Script preprocessor. For accurate classification, it is important to extract the necessary information from the job script data. The *Script Preprocessor* takes as input a raw job script and pre-processes it to isolate the relevant data for classification. This is important as noisy information can lead to misclassifications.

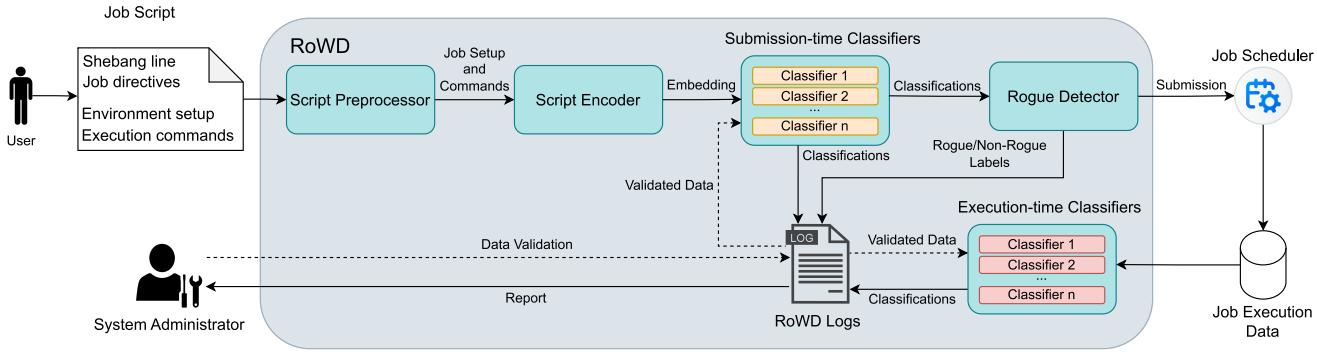

Fig. 1. High-level functioning of the RoWD framework.

A job script is a bash script, which typically follows a structured format composed of 4 sections, namely the shebang line, the job directives, the environmental setup, and the execution commands. The shebang line is the first line that specifies the interpreter for the script, e.g. `#!/bin/bash` or `#!/usr/bin/env bash`. Following this, the script reports the job directives, which are prefixed with a keyword specific to the job scheduler software, such as `#SBATCH` for Slurm or `#PBS` for PBS. The job directives define the job execution setup, such as the *job name*, *output file*, and the *requested resources*. After setting job directives, the script includes environmental setup commands, which load necessary modules, set environment variables, and activate specific software environments to ensure the correct runtime conditions. This part can already provide hints on the kind of workload and application executed by the job. However, the actual programs or applications to run are specified in the execution commands, along with their input files and parameters.

In such a context, the shebang line and the job directives do not provide additional information on the actual workload of the job, hence can be filtered out. To do so, the *Script Preprocessor* splits the job script in lines, and removes all the empty lines, and those starting with #. This allows to remove the shebang line, the job directives, and also the user comments, which can be misleading for the workload recognition. The *Script Preprocessor* output is a string containing only the environmental setup and job execution commands.

Script encoder. To feed the *Script Preprocessor* output into a data-driven classifier, we need to parse the string to a suitable numerical format, which we refer to as embedding. Since the job script is written in natural language and contains lines of code, we use Natural Language Processing (NLP) tools, such as Large Language Models (LLMs).

Submission-time classifiers. This component contains an ensemble of classification models working in parallel to perform several classification tasks, one for each type of operations that needs to be detected in the job activities (e.g., AI activities, cybercrime, cryptomining, etc.). Data-driven models leverage the embedding generated by the *Script Encoder* for classification. The usage of submission-time data (e.g. job script, *requested resources*, *job environment*) allows to classify at job submission time prior to execution. Hence, the *Submission-time Classifiers* target the mitigation of the tampering and information disclosure threats discussed at the beginning of Section 3.

The dangerous activities specific to a system are defined when RoWD is deployed for the system. The classifiers need to be trained on annotated historical data, to then classify new unseen jobs. To this end, any data-driven classification model can be employed, such as ML models, neural networks, or even heuristic algorithms.

Rogue detector. Once job classifications are obtained, it is necessary to understand if the job is authorized to perform its operations, or if it is a rogue one. To do so, the *Rogue Detector* relies on an internal whitelist of groups which are allowed to execute specific types of operations on the system. This list needs to be filled in by the system administrators upon deployment of RoWD, and it has to be continuously updated based on the new projects accepted for execution on the system. For each job, the *Rogue Detector* takes as input the job classifications provided by the *Submission-time Classifiers*), and the group information derived from the original job script. If the group is not in the whitelist for the identified job's operations (from the classifications), the job is labeled as *Rogue*, otherwise it is labeled as *Non Rogue*. The *Non Rogue* jobs proceed to be submitted to the job scheduler, without interferences, while the *Rogue* jobs get reported to the system administrator for further analysis via *RoWD Logs*.

The policy to handle the *Rogue* jobs is system-specific, hence we deliberately keep it out of the scope of our detection framework. For example, a policy could be to continuously check the output of the detection pipeline and automatically cancel the jobs labeled as *Rogue*. Another possibility is to let the jobs run and perform a periodic manual check on the *Rogue* jobs to inspect their actual workload.

Execution-time classifiers. Similarly to the *Submission-time Classifiers*, this component classifies jobs, but this time using the job execution data, such as *performance metrics*, *duration*, *power consumption*. Differently from the *Submission-time Classifiers*, the classification can be performed only after the job execution, or at best during execution (with possible significant overhead on the system operations). The component employs several classifiers for different classification tasks, as many as the classifiers in the *Submission-time Classifiers* component. Again, any data-driven classification model can be employed, such as ML models, neural networks, or even heuristic algorithms.

As mentioned previously, *Execution-time Classifiers* can be instrumental to correct the misclassifications of the *Submission-time Classifiers*. For instance, if a malicious user thoroughly disguises the job operations in the job script, the *Submission-time Classifiers* might misclassify the job. However, the job execution data might reveal typical execution patterns of specific operations, allowing the right classification of the job and correcting the initial misclassification. Therefore, the *Execution-time Classifiers* address the repudiation threat discussed the beginning of Section 3.

RoWD logs. The framework relies on these logs, which are used to report the classifier results and the detection of rogue jobs. They mainly serve as an interface for the system administrator to check the potentially rogue jobs and make informed decision on their execution. Once the classification results are validated, the information can be used to evaluate and update the *Submission/Execution-time Classifiers* (e.g. by retraining the classifiers with additional and more recent data), so as to employ the classifiers in a continuous learning context. All the ids of the jobs categorized as *Rogue* can be saved to the system logs, or published to a streaming messaging interface like Kafka or MQTT.

3.2. RoWD deployment

We implement RoWD as a Python backend, with configurable components and software modules. The implementation can be found in the GitHub repository of the project. (see footnote 1)

At the time of first deployment, RoWD needs to be setup by identifying a list of dangerous workloads for the system (e.g. AI, cryptomining, etc), which are the target of the classification tasks. Then a whitelist for the *Rogue Detector* needs to be created, for each of the dangerous workloads. Once the framework is in place, the inference on a job can be performed in two different ways depending on how and when the classification is needed: at each new job submission, or by periodically querying the data storage to retrieve the accumulated new job scripts.

The integration of RoWD into the job submission pipeline of a system can be achieved through the development of a software plugin for the job scheduler. This plugin can be developed in different programming languages, depending on the job scheduler used. Popular scheduling software, e.g. SLURM [11], FLUX [12] and PBS [13], support the integration of external plugins written in C/C++ and Python. Such plugins are usually event-based, meaning that they are triggered by specific events in the job scheduling process (e.g. job submission, job start, job completion, etc). For our purposes, the *Submission-time Classifiers* can be triggered when a job is submitted to the scheduler, while the *Execution-time Classifiers* can be triggered at job completion. The implementation choices depend on the system policy for software integration in the job submission pipeline, as well as on the job scheduler software installed on the machine.

4. RoWD deployment on fugaku

We deploy RoWD on the Supercomputer Fugaku,[4] a production HPC system hosted at the RIKEN Center for Computational Science, in Japan. The system was developed by RIKEN and Fujitsu and was launched in production in 2020. Thanks to more than 150K interconnected computational nodes, each endowed with 48 Arm cores and 32 GiB of high-bandwidth memory (HBM), the machine can reach a peak performance of 537 PFlops/s, which allows it to rank as the 7*th* most powerful supercomputer in the world.[5]

Discussions with RIKEN researchers highlighted the urgent need to detect rogue AI job executions. While recent AI models have shown outstanding generative and analytics capabilities, if used maliciously or improperly, they can perform cybercrime (e.g. malware development, social engineering attacks and identity theft) and lead to sensitive data leakage or fraud [14–17]. Detecting unauthorized AI jobs is thus fundamental to guarantee the safety and security of the system. To implement RoWD to classify AI jobs, we create SCRIPT-AI, the first public dataset(see footnote 3) of job scripts and use it to train the *Submission-time Classifiers*. Then, we implement an *online* predictive algorithm for the *Execution-time Classifiers*. We start this section by describing SCRIPT-AI and then we give the implementation choices for the different software components of RoWD.

4.1. SCRIPT-AI dataset

Dataset overview. SCRIPT-AI consists of two types of data. The first is the `job_scripts` folder, which contains 652 raw job scripts collected from online sources such as data center documentation and GitHub repositories. The second is the `features.csv` file, which lists six features for each of the scripts in the `job_scripts` folder. The features capture the information that we can extract from the raw job scripts regarding the workload executed by the job (namely the *ai_label*, the *application*, and the *domain*), and job specific details (namely the *plang* and *jsched*). The full list of features and their descriptions are given in Table 1.

Table 1
Features associated to each job script of SCRIPT-AI.

Feature	Description	Type
ID	The job script ID	Int
ai_label	If the job involves AI operations (*AI*/*Non-AI*)	String
source	The job script source	String
application	The application executed by the job	String
domain	The domain of the job's workload	String
plang	The used programming languages, libraries or APIs	String
jsched	The scheduler used for the job submission	String

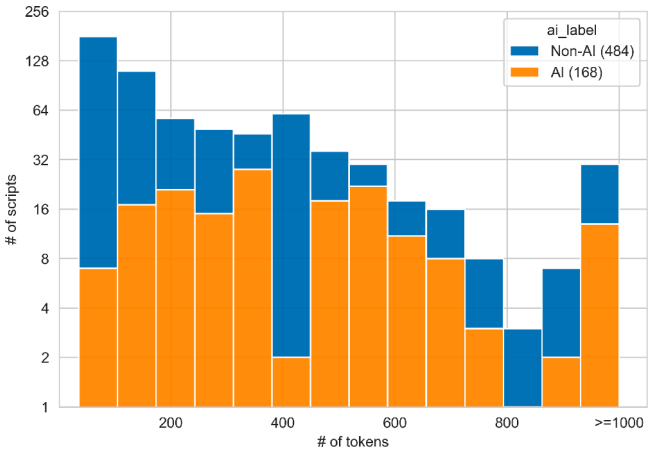

Fig. 2. Distribution of the job script length (in # of tokens), divided by the *ai_label*. The orange bar is in front of the blue one.

The *ID* feature is generated sequentially, depending on the order of the fetched job script. The *application* executed by the job is derived manually by the information provided by the job script *source*. When this information is not available, we report the name of the executable which is launched by the job. The *application* affects the *domain*, which represents the field the workload applies to. Value instances of this feature are *Machine Learning*, *Bioinformatics*, *Chemistry*, *Math*, *Engineering*. The *plang* feature refers to the programming languages and APIs used in the job, such as *Python*, *R*, *OMP*, *Fortran*. This data is extracted by inspecting the job execution commands to understand which programming frameworks are used. Each job is written for a specific job scheduler software (e.g. SLURM, PBS), and we include this information in the *jsched* feature. This feature is generated by checking the directives in the job script's header (e.g.`#SBATCH` means the job is for SLURM, `#PBS` means the job is for PBS). Finally, the *ai_label* feature indicates whether the applications executed by the job involves AI operations, such as ML models or neural networks. If this is the case, the feature is set to *AI*, otherwise to *Non-AI*.

Data analysis. We now analyse the dataset features, inspecting their distribution and correlation. In Fig. 2, we show the distribution of the job script length, divided by the *ai_label*. As a proxy for the length, we tokenize each job script with an instance of a pre-trained BERT tokenizer [18] and count the # of generated tokens. We observe that most job scripts contain fewer than 100 tokens. However, SCRIPT-AI includes both short and long scripts (over 100 entries have more than 500 tokens), making the dataset representative of various real-world job script types. Overall, the dataset contains more *Non-AI* scripts (484) than *AI* ones (168). This can be explained by the fact that *AI* applications are relatively new to the HPC world, and the typical HPC workload is *Non-AI*. While this trend will change in the future, this is still the case for many systems, as witnessed also by the distribution of *AI* and *Non-AI* jobs in the Fugaku dataset (Section 5.1).

Figs. 3 and 4 depict the distribution of the *domain* feature of the job scripts, divided by *ai_label* and *plang*, respectively. The former figure

[4] https://www.fujitsu.com/global/about/innovation/fugaku/
[5] https://top500.org/lists/top500/2025/06/

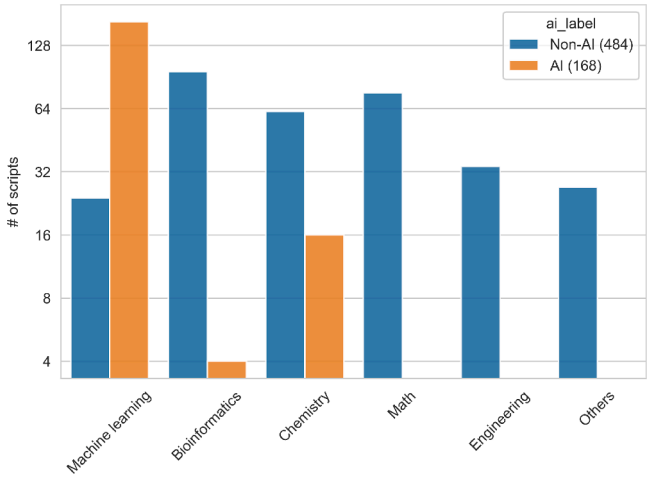

Fig. 3. Distribution of the job script *domain*, divided by the *ai_label*.

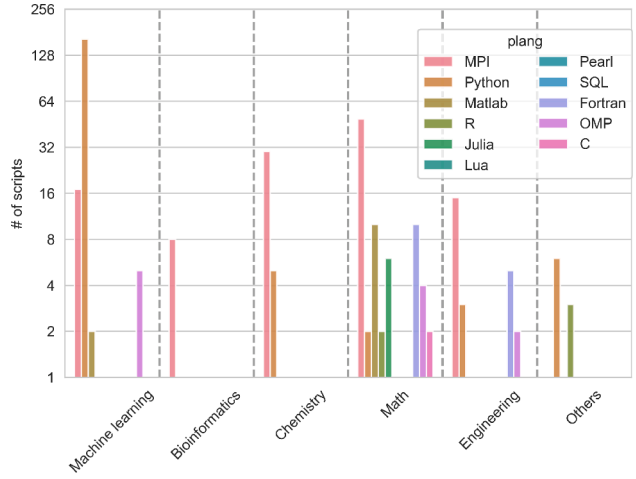

Fig. 4. Distribution of the job script *domain*, divided by the *plang*.

shows that the majority of AI jobs are present in the *Machine Learning* domain, as expected. We note that not all *Machine Learning* jobs are AI. This is because we consider in this domain also the jobs that perform setup (e.g. data pre-processing or environmental setup) for AI modeling, without actually using AI models. This job type is present also in the *Chemistry* domain, where AI models are widely used for simulation and modeling [19]. This figure provides insight into the composition of SCRIPT-AI, which is predominantly made up of Machine Learning jobs. However, SCRIPT-AI also represents other domains, each with more than 20 job scripts.

Fig. 4 presents how programming languages, libraries or APIs are used in different domains. We observe that *OMP* and *MPI* are employed in all the domains. This is justified by the fact that they allow to handle the workload execution in a distributed system, independently of the language used to write the executables. In the *Machine Learning* domain, the majority of the jobs leverage *Python*, which is generally the most used language for ML workload [20]. Whereas in the *Math* and *Engineering* domains, we notice a heavy use of *Matlab* and *Fortran*, respectively, as expected.

In Table 2, we show the # of job scripts per *jsched* value. The majority of the scripts are written for PJSUB, which is also the job scheduler software of Fugaku [21]. However, the dataset is well balanced, as it also contains 191 job scripts for Univa Grid Engine (UGD) [22], 149 for SLURM [11], 85 for PBS [13], and 15 for LFS [23]. Due to the lack of public resources available for COBALT [24] and FLUX [12], these job schedulers are under represented (we foresee the extension the dataset

Table 2
# of job scripts in SCRIPT-AI per *jsched* value.

Job Scheduler	# of job scripts
PJSUB	205
UGD	191
SLURM	149
PBS	85
LSF	15
FLUX	6
COBALT	1

Fig. 5. Word cloud of *application*. Larger fontsizes correspond to higher # of occurrences.

with new resources in future work). An important observation is that SCRIPT-AI is representative of jobs written both for open source schedulers (SLURM, PBS, LFS, FLUX and COBALT) and proprietary ones (PJSUB and UGD). The variety of the target job scheduler makes the dataset a fundamental and unique resource for scheduler-agnostic analysis and predictive modeling.

Finally, we show in Fig. 5 a word cloud generated from the *application* feature, where larger fontsizes correspond to a higher # of occurrences. We see that *application* can be a well-known benchmark or application, such as *eigenexa*, *laamps* and *vasp*, as well as a description of the operations performed by the job, such as *build python*, *train bert* or *torch setup*.

For this work, SCRIPT-AI is essential for training a model to distinguish between AI and Non-AI jobs, as it provides examples of typical HPC jobs in both categories, along with their characteristics, written for different job schedulers. It also provides examples of correct job scripts, which can be used as a reference to write job scripts in a real scenario, or to train ML models to do so automatically (this evaluation is outside the scope of this work).

4.2. Components' implementation

For the Fugaku deployment, we define only one classification task, which is the classification of AI jobs. Thus, the *Submission-time* and *Execution-time Classifiers* components contain only one classifier each.

Script preprocessor. As explained in Section 3, the *Script Preprocessor* filters out unnecessary information from the raw job script. This process is expected to reduce the data dimensionality. We show an evidence of this in Fig. 6, where we notice a significant reduction in the number of tokens after preprocessing, especially for shorter scripts. The preprocessing is expected also to improve classification performance, as only the relevant information is kept for model training and inference. We validated this by measuring the change in accuracy and inference without the *Script Preprocessor*. We observed that, the accuracy worsens by more than 10% on average, and the inference time increases by up to 50%.

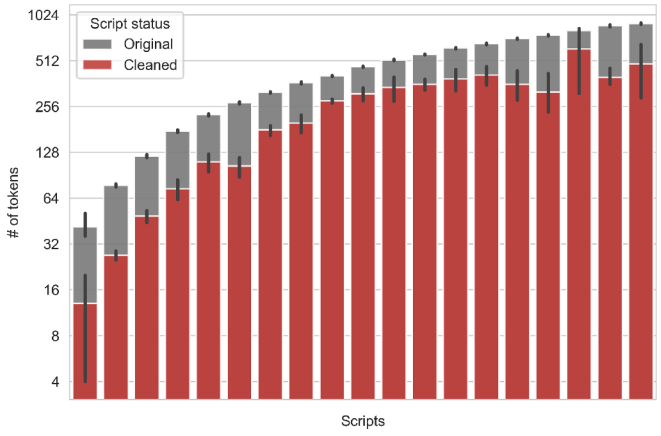

Fig. 6. Job script length (in # of tokens) before and after using the *Script Preprocessor*. The scripts are divided in 80 subsets, based on their original # of tokens.

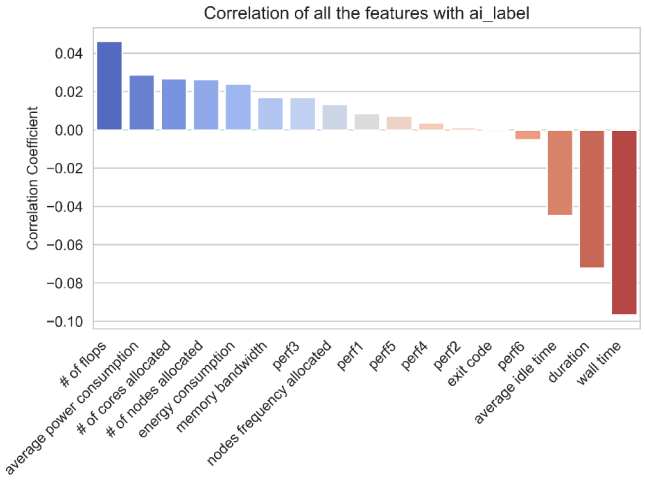

Fig. 7. Feature correlation with the *ai_label*.

Script encoder. We implement the *Script Encoder* using an instance of pre-trained Sentence-Bert (SBert) [25], a sentence embedding model developed by fine-tuning the pre-trained BERT (Bidirectional Encoder Representations from Transformers) [18] on sentence similarity tasks. The original BERT model, as recent LLMs like Llama, is effective in generating word-level embeddings. However, this representation is not ideal to encode full text sequences, as it would result in a high-dimensional matrix (number of tokens in the string $\times$ 768 for BERT) for each sequence encoding. Instead, SBert is specifically designed to generate meaningful sentence-level embeddings, resulting in more informative sequence representation and a one-dimensional embedding. The output of the SBert-based job script encoding is a 384-dimensional floating-point vector, which retains the information contained in the script. The choice of SBert is also motivated by an initial experimentation with LLMs (i.e., SBert, Llama3.1-8B, Llama3.2-1B and Llama3.2-3B, chosen due to their relatively small size for embedding purposes) employed as *Script Encoder*, where SBert showed the best classification performance. We only considered small LLMs because we want to keep the inference time as short as possible and the hardware requirements minimal. The results of the evaluation of different encoders are reported in Table 4 in Appendix.

Submission-time classifier. For this classifier, we train an ML-based model on the job scripts of the SCRIPT-AI dataset. To choose an appropriate classification model, we experimented with different ML models widely used for classification tasks, including Random Forest (RF) [26], k-nearest neighbors (KNN) [27], Support Vector Machine Classifier (SVC) [28], and Logistic Regression (LR) [29]. Based on our initial results (available in Table 4 in Appendix), we picked LR, which obtained the best classification performance. During training, the model learns the relationship between one or more independent variables and a binary outcome using the logistic function, which outputs values between 0 and 1. The model estimates the coefficients through maximum likelihood estimation, with the aim of finding the best fit that maximizes the likelihood of observing the given data. A key feature of LR is its lightweight nature, which makes it suitable for production environments where the model overhead should be as low as possible.

We favor traditional ML models over more sophisticated Deep Learning (DL) solutions, as ML models offer extremely fast inference, which is crucial to avoid impacting job scheduling latency. In contrast, DL models, especially those using high-dimensional embeddings or multi-layer architectures, can introduce non-negligible runtime overhead. Moreover, as we will show in Section 5, ML models already obtain highly accurate classification performance.

Execution-time classifier. For the *Execution-time Classifier*, we rely on a state-of-the-art *online* algorithm proposed in our previous work on job classification [9,30]. The algorithm implements an ML classifier, which is re-trained every β days, on the data of the jobs executed in the previous α days. Updating the model periodically is fundamental to adapt the model to the workload change in the system, and guarantee optimal prediction performance, as discussed in [9,30,31]. To find the best algorithm setup, we experimented with different ML models, such as RF, KNN, SVC and LR, using different combinations of values for α (namely 1, 5, 10, 30, 45, and 60) and β (namely 1, 2, 5, and 10). Based on our results (reported in Table 5 in Appendix), we chose RF model with $\alpha = 5$ and $\beta = 1$, which resulted in the best prediction performance. A low α value is fundamental to reduce the amount of data and computational time needed for model training.

We train the models on the historical traces of Fugaku, using the job execution data. The feature set is selected based on an initial empirical evaluation, identifying the subset that offers better predictive performance using metrics such as correlation and feature importance. In Fig. 7, we show the correlation values of the execution time features with the *ai_label*. Starting from this set, we remove the features where $|correlation\ coefficient| < 0.015$ (decided based on the correlation coefficient of the central values), namely *perf3*, *nodes frequency allocated*, *perf1*, *perf5*, *perf4*, *perf2*, *perf6* and *exit code*. Then, we filter out the repeated information (i.e. features which are the same or directly proportional), such as *# of cores*. The final feature set for model training and prediction is composed of the following features: *energy consumption*, *average power consumption*, *# of cores allocated*, *# of nodes allocated*, *duration*, *wall-time*, *average idle time*, *memory bandwidth* and *# of flops*.

We note that we deliberately exclude user-related information from the feature set, so as to make the classification not bound to the specific user but rather on the job activities.

5. Experimental validation on fugaku

We validate RoWD on unseen Fugaku data to assess the generalization capability of the classifiers, their robustness to adversarial behavior, and the runtime overhead of our approach. In this section, we first report the experimental setup and then present our results.

The experiments are conducted on a machine with 2 AMD EPYC 9534 CPUs and 4 NVIDIA H100 GPUs. For the implementation of the ML models, we rely on the *scikit-learn* library[6] and use the default implementations. The SBert model is provided by the *sentence transformers* library,[7] while the weights are pulled from Huggingface.[8] We use the pre-

[6] https://scikit-learn.org/stable/
[7] https://www.sbert.net
[8] https://huggingface.co

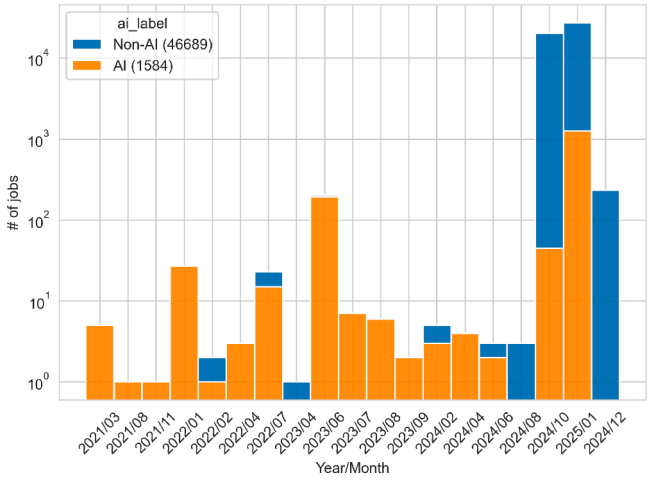

Fig. 8. Distribution of the Fugaku job executions in time, divided by the *ai_label*. The orange bar is in front of the blue one.

trained model *all-MiniLM-L6-v2*,[9] since it has the best trade-off between prediction quality and speed [25]. We increase the `max_seq_length` parameter value to 512, from the original 256, which controls the maximum number of tokens in input to generate the embedding. If a script length is longer than `max_seq_length`, the exceeding tokens are truncated. In Figs. 2 and 6, we observe that a significant portion of the job scripts contain more than 200 tokens. The new `max_seq_length` value allows the model to use more tokens for the embedding generation and minimize information loss due to the truncation. Finally, the Llama models are pulled from Huggingface, and the Gemini model is queried through the official API interface Google Gen AI.[10]

5.1. Experimental setup

For the experiments, we create a new dataset of job scripts and execution data extracted from Fugaku. In the following, we describe this dataset, along with the methodology used to train and evaluate the classifiers, the evaluation metrics, and the classification baselines.

Fugaku test data. To generate our test set, we randomly sample the scripts and execution data of ~50K jobs run on Fugaku between 2021 and 2025. The extracted data are manually annotated as *AI/Non-AI* depending on whether the job execution involves AI operations. To this end, we apply the same methodology used to annotate SCRIPT-AI, as described in Section 4. In Fig. 8, we show the distribution of *AI/Non-AI* jobs over time, where we witness a growing presence of AI jobs in recent times. This is explainable by the increasing usage of HPC systems to execute AI operations, which are now becoming the predominant workload in modern systems [32]. However, their overall presence in the dataset is limited, with 1585 jobs (3%) being labeled as *AI*, in comparison to 46,689 (97%) labeled as *Non-AI*. This makes the systematic detection of AI jobs nontrivial. We note that this sample is a small subset of all the jobs executed on Fugaku and thus is not intended to be representative of the whole system workload.

Classifier training and testing. To evaluate the *Submission-time Classifier*, we start by training it on SCRIPT-AI. This operation takes less than 20 seconds on the machine used for the experiments. For testing, we take the Fugaku data as a whole and employ the classifier to classify all the job scripts as *AI/Non-AI*.

To train and test the *Execution-time Classifier*, we use the job execution data of the Fugaku dataset. To this end, we need a portion to use as the first training set, to then start testing and retraining on the remaining data. When working with historical data, it is not realistic to do inference on a job by learning from the data of the future jobs submitted at a later time [30]. This is because in a real system, job data can be evaluated and collected correctly only after the job ends and the data collection process finishes successfully. Thus, whenever a job is submitted, we can use only the data of the jobs which are already finished to perform the learning phase. For this reason, we order the Fugaku job data chronologically based on their *submission_time*, to ensure that the training data always come chronologically before the test set one.

Since the *Execution-time Classifier* is retrained periodically, we have to create different training and test sets. We consider as the first training set all the jobs that are submitted in the first α days and finished after the αth day. Starting from the *submission_time* of the first job not present in the first training set, we divide the data in batches in chronological order, where each batch contains the jobs submitted in the next β days. We then iterate over each batch, considering it as a new test set. At each iteration, the training set is updated with the data of the last α days and the model is retrained. To guarantee soundness of the setting, we use the time information provided by the *end_time* of the jobs to ensure that all the jobs in the training splits end before the submission of the jobs in the test sets. As explained in Section 4, we set $\alpha = 5$ and $\beta = 1$ after an initial empirical investigation of the best algorithmic setting.

Evaluation metrics. The prediction accuracy is measured using the F1-macro average score [33], which is a widely used metric for classification problems. It is computed as the mean of the F1-score obtained on specific *AI* and *Non-AI* classes. The F1-score on a single class is computed as the harmonic mean between the precision and recall on the target values. Hereafter, we will refer to the F1-macro average as F1.

Moreover, we are interested in evaluating the rates of misclassifications, namely the false positives (FPs) and false negatives (FNs). FPs are the jobs classified as *AI* that are in fact not, while FNs are the AI jobs classified as *Non-AI*. The latter category is most relevant for our purposes, as it quantifies the amount of undetected AI jobs that can ultimately be rogue ones. Hence, to have an effective detection mechanism, the FN value should be as low as possible.

Baselines. For the *Submission-time Classifier*, we define three baselines. The first is an LLM, specifically Gemini 2.0 flash (Gemini) [34] developed by Google, which provides open APIs. We employ it in a few-shot fashion, meaning that the model relies on its internal knowledge for classification plus some information retrieved from SCRIPT-AI on typical AI and Non-AI applications. We note that Gemini has also internet access to obtain additional information. To this end, the Fugaku job scripts are anonymized w.r.t. the user information and fed into the model along with a prompt which instructs the model on the task to perform. The full prompt is:

"Classify the following HPC job scripts based on whether they involve AI tasks. AI tasks typically include commands or operations related to machine learning, neural networks, data processing, or models like 'pytorch', 'tensorflow', 'keras', 'transformers', or 'scikit-learn'. Examples of AI tasks are: 'bert', 'dpgen', 'deepmd', 'gpt'. Examples of non-AI tasks are: 'vasp', 'laamps', 'gromacs'.
Return only a string 'Non-AI' or 'AI'.
Do not provide additional text."

The second baseline is the related work [2] discussed in Section 2, where an RF model is trained on fuzzy hashes of the job binaries. As we do not have this type of data, we apply RF to the raw job scripts (referred as ssdeep + RF). The final baseline is named keywords matching. We define a list of keywords related to AI workload, namely *"pytorch", "keras", "train", "tensorflow"*, and *"jax"*, and it classifies a job as *AI* if one or more keywords are present in the job script, otherwise it classifies the job as *Non-AI*.

[9] https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2
[10] https://pypi.org/project/google-genai/

Table 3
The classification performance of the RoWD classifiers against the baselines. For the Inference Time, False Positives and False Negatives, lower values correspond to better results. For all the other metrics, higher values correspond to better results. The best results are highlighted in bold.

Approach	F1	True Positives	True Negatives	False Positives	False Negatives	Inference Time (in S)
RoWD StC	**0.95**	**1,296**	**46,685**	4	288	$< 1 * 10^{-3}$
Gemini	0.58	1360	39,135	7554	**224**	$7.5 * 10^{-1}$
Ssdeep + RF	0.50	6	46,689	**0**	1578	$< 1 * 10^{-3}$
Keywords Matching	0.85	973	46,517	172	611	$< 1 * 10^{-3}$
RoWD EtC	**0.90**	**872**	39,800	12	**448**	$< 1 * 10^{-3}$
MFREQ	0.50	5	**39,809**	**3**	1315	$< 1 * 10^{-3}$
MSIMILAR	0.83	772	39,583	229	548	$< 1 * 10^{-3}$

Concerning the *Execution-time Classifier*, we identify two baselines. The first one classifies based on the most frequent label in the training set, ignoring the input job data (referred to as MFREQ). The other baseline returns the class of the most similar job (the one for which the cosine similarity between the input features is the lowest) in the training set (referred to as MSIMILAR). For a fair comparison, we employ the baselines in the same *online* algorithm as the *Execution-time Classifier*, i.e., by updating the models every $\beta = 1$ day, with the data of the last $\alpha = 5$ days of job executions.

5.2. Experimental results

In Table 3, we report the results of the *Submission-time Classifier* (StC) and *Execution-time Classifiers* (EtC) of RoWD compared to the baselines.

Submission-time classifiers. Our solution outperforms all the other methods with an F1 score of 0.95 against 0.85, the best baseline result obtained by the keywords matching method. We observe that RoWD misclassifies very few instances, namely 4 FP and 288 FN. Instead, the other methods can only classify correctly one class at a time. For instance, while Gemini achieves the lowest number of FN (224), it gives a high number of FP (7,554). Similarly, ssdeep + RF achieves 0 FP, but it scores an high number of FN (1,587). These results make the alternative approaches unreliable, as many AI jobs are not detected or some users might see their jobs flagged as rogue *AI*, while in fact they are not. The comparison with the baselines reveals that the task requires a trained model, and using general knowledge of a pre-trained LLM in a zero/few-shot learning setting is not enough to achieve accurate classification. Overall, the results prove RoWD to be the most suitable approach for the detection of potentially rogue jobs at submission-time, and to mitigate the tampering and information disclosure threats (Section 3) for the Fugaku use-case.

Additionally, we evaluate the overhead on the system operations incurred by the use of the approach, which is quantified as the time (in seconds) required to classify a job script (inference time). Again, RoWD is the best performing solution, as it classifies a job in less than 10^{-3} seconds. This time is negligible w.r.t. the average waiting time of Fugaku jobs in the job queue, which is around 3 minutes (as also discussed in [9]). The overhead of our approach is suitable for a production system, as it would not affect its normal workload submission pipeline.

Execution-time classifiers. Again, RoWD scores the best F1 (0.90), outperforming both MFREQ (0.50) and MSIMILAR (0.83). The MFREQ approach obtains only 3 FP, yet it gives more than 1300 FN and only 5 TP, making the model not reliable. Similar conclusions can be drawn for MSIMILAR, which returns 772 TP, but a high FP (229). Conversely, RoWD scores the lowest number of FN (448) and highest TP (872), while still obtaining a low number of FP (12), making it the most reliable and robust to use in production.

As discussed in Section 4, the *Execution-time Classifier* requires job execution data for training. Since some of the Fugaku data are used for this purpose, we could not classify them. For this reason, we cannot compare the *Submission-time Classifier* and *Execution-time Classifier* fairly.

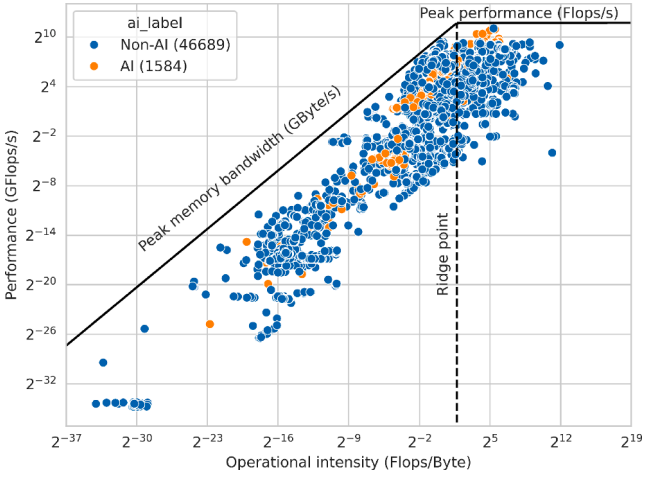

Fig. 9. *Roofline* plot of the Fugaku jobs, divided by the *ai_label*.

However, from the comparison of the results obtained by the two, we can conclude that the *Submission-time Classifier* is more accurate for our data and use-case. A possible explanation can be obtained by inspecting Fig. 9, which shows the *roofline* [35] plot of the Fugaku jobs, a visual model of the computational performance of jobs. The plot illustrates the maximum achievable performance based on operational intensity (ratio between *flop/s* and *memory bandwidth*) and the machine's hardware limits, such as the *peak memory bandwidth* and *peak flop/s*. We observe that there is no clear pattern in the distribution of AI and Non-AI jobs, showing the non-triviality of the task when using *performance metrics* of the job execution. In an initial experimental study, we observed that using only *performance metrics* as input for the classifiers does not allow to obtain accurate classification. By adding the other job execution features (i.e., *energy consumption, average power consumption, # of cores allocated, duration*, and *wall-time, average idle time*) we provide more informative features to the classifiers and we ultimately reach the 0.90 F1 score. Hence, in this task, the job scripts are more informative on the job operations, and they represent a better input for a classification model. Nevertheless, the *Execution-time Classifiers* can be a good alternative to *Submission-time Classifiers* when they are not deployable, as in our case, for instance, they still reach an F1 score of 0.90.

Robustness against adversarial behavior. We conduct additional experiments to assess the robustness of the approach against realistic rogue or adversarial job behaviors. Specifically, we perform a controlled system-level security validation in which job scripts are intentionally disguised. We define a disguise technique that replaces a given % of the job script lines (from 10% to 70%) with random strings. We then apply it to job scripts from the Fugaku dataset to evaluate how the classification performance of the trained classifiers change. We observe that up to 50% disguising, the *Submission-time Classifier* remains robust, achieving an F1 score of 0.90 (only 0.05 lower than when using the original, non-

disguised scripts). When more than 50% of the script is disguised, performance degrades, dropping to an F1 score of 0.17 at 70% disguising (a decrease of 0.78 from the original 0.95). With more than 70% of the script lines disguised, the model is no longer able to classify AI jobs. This behavior is expected, as most of the informative content of the job script is removed. Nevertheless, these results demonstrate that the *Submission-time Classifier* is robust to intentional and malicious disguising of substantial portions of job scripts, a scenario typical of rogue workloads.

Instead, the *End-time Classifier* is unaffected by job-script disguising, consistently achieving an F1 score of 0.90 regardless of the extent of malicious modification applied to the script. Consequently, even if a user disguises the job script to conceal its activities, RoWD's *Execution-time Classifiers* can still reliably classify AI jobs and mitigate repudiation threats in the Fugaku use case.

Runtime overhead. The runtime overhead depends largely on deployment choices, such as hardware and framework configuration (e.g., the number of jobs processed concurrently and whether the system is triggered at each job submission or operates asynchronously in batches). Here, we report an analysis to quantify the runtime overhead.

As reported in [9], Fugaku handles approximately 25K job submissions per day, corresponding to about 17 submissions per minute, or roughly one job every 3 seconds. As discussed in Section 5, RoWD runs on a dedicated management server, and the execution time of the full pipeline (including pre-processing and model inference) is approximately ∼1,ms per job for both the *Submission-time* and *Execution-time Classifiers*. This overhead is negligible compared to the average job waiting time in the Fugaku queue, which is nearly 3 minutes [9]. Even if the inference pipeline is triggered at every job submission, no bottleneck arises between successive inferences, and the management server utilization would be approximately 0.03% of a single CPU core. The total daily overhead amounts to roughly 25 seconds for the entire Fugaku workload. Moreover, since RoWD does not require GPUs (as the employed ML models are CPU-based), inference pipelines for different jobs can run in parallel on a single CPU node using multiple cores. This enables efficient handling of near-concurrent job submissions. The inference performance can be further improved on more powerful or distributed systems by increasing the degree of parallelism.

6. Discussion

SCRIPT-AI is relatively a small-size dataset. While this may seem like a limitation, it is in fact a strength. RoWD achieves accurate classification on unseen data despite being trained with little data, proving the robustness and generalizability of the approach. Moreover, the fact that SCRIPT-AI is a public resource allows to apply RoWD to other real HPC environments without the need for domain adaptation (to the target system) or additional hand-labeled data, which can be expensive or challenging to obtain (due to, for instance, privacy concerns or limited data availability).

We acknowledge, however, that RoWD retains some limitations despite showing a strong potential for detecting rogue jobs in HPC environments. In this first version, RoWD analyzes only the submitted shell scripts and does not inspect the actual job code nor deeply nested commands. To this end, an attacker could craft a script that merely renames or obfuscates executable files to conceal malicious behavior, bypassing RoWD's detection logic. We will investigate possible solutions in future work. However, inspecting such data raises significant privacy concerns, as jobs may contain proprietary models or confidential data that the operators are not authorized to access. In addition, an extensive analysis of job code may require the inspection of thousands of lines of code, from dozens of different files, possibly written in diverse languages or frameworks (e.g. binary files, executables or scripts), ultimately making automatic deep analysis dangerous, unreliable and often infeasible at scale.

Additionally, RoWD can be possibly vulnerable to the spoofing threat (Section 3), i.e., if a malicious user logs in with leaked credentials from a whitelisted user. In fact, this represents a security breach for the whole system, which cannot be detected by RoWD, nor by other security screening systems working at the job level. To this end, we transfer the responsibility of the spoofing to the users, as they are not supposed to disclose or share their credentials with other users. Nevertheless, RoWD can be leveraged to mitigate this issue in two ways. First, as RoWD's detection is enabled for all the submitted jobs, the system administrator can anyway perform a sanity check for the jobs classified as AI, regardless of the whitelisted users. Second, RoWD's classifications can be used to detect a suspicious change in the workload executions of a user. For instance, if a user, who does not normally execute AI jobs, suddenly starts to submit several AI jobs, it could be that the user's credentials have been compromised and the account is used by someone else with malicious intentions.

Another limitation can be that RoWD is currently validated only for the detection of AI jobs. However, for the time being, there are no other public datasets containing job scripts labeled based on the presence of rogue operations or other possibly dangerous activities (e.g. cryptocurrencies mining, cybercrime, etc). Hence, we could not extend the evaluation to other tasks, which we foresee as a future development for this work.

7. Conclusions

We presented RoWD, the first security-screening framework for the job-submission workflow in HPC systems. Given the thousands of jobs submitted daily to modern HPC systems, detecting rogue executions without automated and systematic support is impractical. RoWD addresses this need by automatically identifying suspicious job executions and reporting them to system administrators for further investigation. RoWD relies on an ensemble of classification models that operate either at job submission time using job scripts (*Submission-time Classifiers*) and/or during or after execution using job execution data (*Execution-time Classifiers*). This design makes the detection pipeline flexible and customizable, allowing it to adapt to system-specific constraints such as data availability and compliance with data-management policies.

We deployed RoWD on the Fugaku supercomputer to address the problem of classifying AI job executions. To this end, we created and released SCRIPT-AI, the first publicly available dataset comprising approximately 650 job scripts, enriched with features describing each workload (i.e., the *application* and its *domain*), job specifics (i.e., the *job scheduler software* and the *programming language*), and whether the job involves AI operations. We used this dataset to train a *Submission-time Classifier*, implemented as a Logistic Regression (LR) model receiving SBert-based encodings of job scripts as input. In contrast, we implemented the *Execution-time Classifier* using a Random Forest (RF) model, employed within an *online* algorithm that periodically retrains the model to maintain optimal prediction performance. We evaluated both classifiers on data from approximately 50K jobs executed on Fugaku between 2021 and 2025. Our results show that RoWD (i) accurately classifies these jobs, achieving an F1 score of 0.95 with the *Submission-time Classifier* and 0.90 with the *Execution-time Classifier*, (ii) is robust against adversarial behavior (i.e., disguisal of significant portions of job scripts), (iii) incurs low runtime overhead on the system operations, making it suitable for strengthening the security of HPC environments and for real-time deployment in production systems.

In future work, we plan to extend RoWD to other classification tasks, such as the detection of cybercrime, cryptomining, or data leaking jobs. To this end, we plan to investigate other classification techniques, such as using more sophisticated DL methods or fine-tuning LLMs on large datasets. Moreover, we are looking into obtaining data from other systems to test RoWD on different environments. Finally, we aim to deploy it in production to enhance the security of the job submission pipeline of a real HPC system.

CRediT authorship contribution statement

Francesco Antici: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Data curation; **Jens Domke:** Supervision, Conceptualization; **Andrea Bartolini:** Supervision, Funding acquisition; **Zeynep Kiziltan:** Supervision, Writing – review & editing; **Satoshi Matsuoka:** Supervision, Conceptualization.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Andrea Bartolini reports financial support was provided by Horizon Europe. Andrea Bartolini reports financial support was provided by European High Performance Computing Joint Undertaking. Andrea Bartolini reports financial support was provided by Horizon Europe. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research has been partially funded by the EU Pilot for exascale EuroHPC EUPEX (g.a. 101033975), and the EuroHPC JU SEANERGYS (g.a. 101177590) projects.

Appendix A.

In this appendix we report the results obtained during the initial development phase of RoWD.

B. Execution-time classifier setup

B. Submission-time classifier setup

In Table 4 we show the results obtained by different combinations of models and techniques for the *Script Encoder* and the *Submission-time Classifier*. We report the F1 (global and per-class), the accuracy and the TP, TN, FP, FN and Embedding Size. The encoding size refers to the size of the vector outputted by the *Script Encoder*. We can observe that using SBert as *Script Encoder*, in combination with LR allows to obtain the best result overall, with an F1 of 0.95. Using SBert as a *Script Encoder* allows to obtain better prediction performance, since such model is trained for sentence embedding purposes, rather than for word embedding ones, like llama models. Moreover, it allows to achieve the smallest encoding size, as it generates a vector of only 384 elements, w.r.t. the llama models which generate at least 2048-dimensional vectors. Generally speaking, the RF model obtains better results with the llama-based encodings, especially with the llama-3.2-1B (F1 of 0.86).

In Table 5, we report the classification accuracy (F1 score) of several *Execution-time Classifiers*, with different ML models, and *online* algorithmic setting (namely the values of α and β). We can observe that the SVC and LR models are not suited for the task, as they obtain an F1 not greater than 0.64 and 0.50, respectively. The KNN achieves an F1 of 0.82 with just 1 day of training data ($\alpha = 1$). However, it is outperformed by the RF, which obtains an F1 of 0.90 with α as low as 5. Since the RF's accuracy does not improve by increasing α, we stick to the lowest α value for the algorithm, which allows to minimize the amount of data needed to train the model. In general, we observe that for both the KNN and the RF, setting $\beta > 1$ leads to a degradation of the classification accuracy. This can be explained by the fact that the models are re-trained less frequently, and to achieve optimal prediction performance they need to be re-trained daily ($\beta = 1$).

Table 4
The extended results of the different combinations of LLM Encoders and ML Classifiers. For the Inf Time, FP and FN, lower values correspond to better results. For all the other metrics, higher values correspond to better results. The best results are highlighted in bold.

Script Encoder + ML Classifier	F1	Accuracy	F1 (Non-Ai)	F1 (Ai)	TP	TN	FP	FN	Encoding Size
SBert + RF	0.80	0.98	0.99	0.61	701	46,688	883	**1**	384
SBert + KNN	0.57	0.82	0.90	0.25	**1415**	38,187	**169**	8502	384
SBert + SVC	0.87	**0.99**	0.99	0.74	939	**46685**	645	4	384
SBert + LR	**0.95**	**0.99**	**1.00**	**0.90**	1296	**46685**	288	4	384
llama-3.1-8B + RF	0.80	0.98	0.99	0.62	716	46,664	868	25	4096
llama-3.1-8B + KNN	0.51	0.79	0.88	0.14	796	37,314	788	9375	4096
llama-3.1-8B + SVC	0.81	0.98	0.99	0.62	786	46,528	798	161	4096
llama-3.1-8B + LR	0.75	0.98	0.99	0.51	559	46,660	1025	29	4096
lLama 3.2-1B + RF	0.86	**0.99**	0.99	0.72	892	46,680	692	9	2048
lLama 3.2-1B + KNN	0.69	0.94	0.97	0.42	787	44,659	637	2030	2048
lLama 3.2-1B + SVC	0.81	0.98	0.99	0.64	787	46,586	797	103	2048
lLama 3.2-1B + LR	0.55	0.97	0.98	0.12	103	46,680	1481	9	2048
llama-3.2-3B + RF	0.82	0.98	0.99	0.65	766	46,682	818	7	3072
llama-3.2-3B + KNN	0.77	0.97	0.98	0.55	944	45,793	640	896	3072
llama-3.2-3B + SVC	0.81	0.98	0.99	0.63	792	46,549	792	140	3072
llama-3.2-3B + LR	0.75	0.98	0.99	0.52	561	46,673	1023	16	3072

Table 5
The F1 of the ML models with different values of α and β. The best results per model are highlighted in *italic*, while the best results overall are highlighted in **bold**.

α		1				5				15				30				45				60			
β		1	2	5	10	1	2	5	10	1	2	5	10	1	2	5	10	1	2	5	10	1	2	5	10
R	F	0.83	0.79	0.80	0.76	***0.90***	0.87	0.85	0.79	***0.90***	0.87	0.86	0.79	***0.90***	0.87	0.86	0.79	***0.90***	0.87	0.86	0.79	***0.90***	0.87	0.86	0.80
KNN		*0.82*	0.81	0.77	0.72	0.81	0.79	0.74	0.69	0.81	0.79	0.74	0.69	0.81	0.79	0.74	0.70	0.81	0.79	0.74	0.70	0.81	0.79	0.74	0.70
SVC		*0.64*	*0.64*	0.59	0.57	0.49	0.49	0.49	0.49	0.49	0.49	0.49	0.49	0.50	0.49	0.49	0.49	0.50	0.49	0.49	0.49	0.50	0.49	0.49	0.49
LR		0.49	0.49	0.49	0.49	0.49	0.49	0.49	0.49	*0.50*	0.49	0.49	0.49	*0.50*	0.49	0.49	0.49	*0.50*	*0.50*	*0.50*	0.49	*0.50*	*0.50*	*0.50*	*0.50*

References

[1] A. Gangwal, S.G. Piazzetta, G. Lain, M. Conti, Detecting covert cryptomining using hpc, in: Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings 19, Springer, 2020, pp. 344–364.

[2] T. Jakobsche, F.M. Ciorba, Using malware detection techniques for HPC application classification, in: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2024, pp. 1773–1780.

[3] E. Ates, O. Tuncer, A. Turk, V.J. Leung, J. Brandt, M. Egele, A.K. Coskun, Taxonomist: application detection through rich monitoring data, in: Euro-Par 2018: Parallel Processing: 24Th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24, Springer, 2018, pp. 92–105.

[4] Y. Gao, Y. Zhou, R. Li, J. Wei, Y. Wang, J. Lin, Research and development of evaluation tools on user job level index of HPC cluster, in: Proceedings of the 2025 Supercomputing Asia Conference, SCA '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 42-51. https://doi.org/10.1145/3718350.3718356

[5] K. Yamamoto, Y. Tsujita, A. Uno, Classifying jobs and predicting applications in HPC systems, in: High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33, Springer, 2018, pp. 81–99.

[6] P. Haridas, G. Chennupati, N. Santhi, P. Romero, S. Eidenbenz, Code characterization with graph convolutions and capsule networks, IEEE Access 8 (2020) 136307–136315.

[7] T. Jakobsche, N. Lachiche, A. Cavelan, F.M. Ciorba, An execution fingerprint dictionary for hpc application recognition, in: 2021 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2021, pp. 604–608.

[8] V. Ramos, C. Valderrama, S.X. de Souza, P. Manneback, An accurate tool for modeling, fingerprinting, comparison, and clustering of parallel applications based on performance counters, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2019, pp. 797–804.

[9] F. Antici, A. Bartolini, Z. Kiziltan, O. Babaoglu, Y. Kodama, MCBound: An online framework to characterize and classify memory/compute-bound HPC jobs, in: To Appear in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2024.

[10] R. Khan, K. McLaughlin, D. Laverty, S. Sezer, STRIDE-Based threat modeling for cyber-physical systems, in: 2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2017, pp. 1–6. https://doi.org/10.1109/ISGTEurope.2017.8260283

[11] A.B. Yoo, M.A. Jette, M. Grondona, Slurm: simple linux utility for resource management, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2003, pp. 44–60.

[12] T. Patki, D. Ahn, D. Milroy, J.-S. Yeom, J. Garlick, M. Grondona, S. Herbein, T. Scogland, Fluxion: a scalable graph-based resource model for HPC scheduling challenges, in: Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 2077–2088.

[13] H. Feng, V. Misra, D. Rubenstein, PBS: A unified priority-based scheduler, in: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2007, pp. 203–214.

[14] T.F. Blauth, O.J. Gstrein, A. Zwitter, Artificial intelligence crime: an overview of malicious use and abuse of AI, IEEE Access 10 (2022) 77110–77122.

[15] M. Brundage, S. Avin, J. Clark, H. Toner, P. Eckersley, B. Garfinkel, A. Dafoe, P. Scharre, T. Zeitzoff, B. Filar, et al., The malicious use of artificial intelligence: Forecasting, prevention, and mitigation, arXiv preprint arXiv:1802.07228 (2018).

[16] B.A. Juliussen, J.P. Rui, D. Johansen, Algorithms that forget: machine unlearning and the right to erasure, Comput. Law Security Rev. 51 (2023) 105885.

[17] J. Borkar, What can we learn from Data Leakage and Unlearning for Law?, arXiv preprint arXiv:2307.10476 (2023).

[18] J. Devlin, M.-W. Chang, K. Lee, et al., BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 NAACL: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186.

[19] D. Dickel, M. Nitol, C.D. Barrett, LAMMPS Implementation of rapid artificial neural network derived interatomic potentials, Comput. Mater. Sci. 196 (2021) 110481.

[20] D. Talati, Python: the alchemist behind AI's intelligent evolution (2021).

[21] A. Uno, F. Sueyasu, R. Sekizawa, Operations management software of supercomputer fugaku, Fujitsu Technical Rev. (3) (2020) 2003–2020.

[22] A. Engineering, Univa grid engine (2025). https://doi.org/https://altair.com/grid-engine

[23] S. Zhou, X. Zheng, J. Wang, P. Delisle, Utopia: a load sharing facility for large, heterogeneous distributed computer systems, Software: Practice Exper. 23 (12) (1993) 1305–1336.

[24] N. Desai, Cobalt: an open source platform for hpc system software research, in: Edinburgh BG/L System Software Workshop, 2005, pp. 803–820.

[25] N. Reimers, I. Gurevych, Sentence-bert: Sentence embeddings using siamese bert-networks, arXiv preprint arXiv:1908.10084 (2019).

[26] L. Breiman, Random forests, Mach. Learn. 45 (2001) 5–32.

[27] E. Fix, J.L. Hodges, Discriminatory analysis. nonparametric discrimination: consistency properties, Int. Statist. Rev./Revue Internat. Statist. 57 (3) (1989) 238–247.

[28] M.A. Hearst, S.T. Dumais, E. Osuna, J. Platt, B. Scholkopf, Support vector machines, IEEE Intell. Syst. Their Appl. 13 (4) (1998) 18–28. https://doi.org/10.1109/5254.708428

[29] M.P. LaValley, Logistic regression, Circulation 117 (18) (2008) 2395–2399.

[30] F. Antici, A. Borghesi, Z. Kiziltan, Online job failure prediction in an HPC system, in: Euro-Par 2023: Parallel Processing Workshops: Euro-Par 2023 International Workshops, Limassol, Cyprus, August 28–September 1, 2023, Revised Selected Papers, Springer Nature, 2023.

[31] F. Antici, K. Yamamoto, J. Domke, Z. Kiziltan, Augmenting ML-based predictive modelling with NLP to forecast a Job's power consumption, in: Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 1820–1830.

[32] I. Ettifouri, M. Zbakh, C. Tadonki, The need for HPC in AI solutions, in: International Conference of Cloud Computing Technologies and Applications, Springer, 2024, pp. 137–159.

[33] M. Sokolova, N. Japkowicz, S. Szpakowicz, Beyond accuracy, F-Score and ROC: a family of discriminant measures for performance evaluation, Vol. 4304, 2006, pp. 1015–1021. https://doi.org/10.1007/11941439_114

[34] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A.M. Dai, A. Hauth, K. Millican, et al., Gemini: a family of highly capable multimodal models, arXiv preprint arXiv:2312.11805 (2023).

[35] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76.