Comprehensive Journal of Science
Volume (10), Issue (39), (March, 2026)
ISSN: 3014-6266

مجلة العلوم الشاملة
المجلد (10) ملحق العدد 39 مارس 2026
ردمد: 3014-6266

A Proactive Hybrid Approach for Securing Container Systems Integrating Deep Kernel Monitoring with Two-Stage Machine Learning Classification

نهج هجين استباقي لتأمين أنظمة الحاويات يدمج مراقبة النواة العميقة مع تصنيف التعلم الآلي ثنائي المراحل

Nuha Omran
University of Zawiya - Faculty of Science
Abo.khdeir@zu.edu.ly

Ali Alissawi Ahmed AlQudairi
University of Zawiya - Faculty of Science
Alialgader53@gmail.com

تاريخ الاستلام:2026/02/15 –تاريخ المراجعة:2026/03/12– تاريخ القبول: 2026/03/13– تاريخ للنشر 2026 /04/25

Abstract

With the rapid evolution of container technologies in modern cloud computing architectures, container breakouts constitute a serious threat, as they exploit security vulnerabilities in shared isolation at the operating system kernel level. Traditional detection systems often lack kernel-level visibility, requiring operating system kernel monitoring for real-time analysis. This paper presents a hybrid protection framework that combines operating system kernel monitoring (eBPF) with accelerated cloud processing via Google Colab. The proposed system employs a multilayer detection mechanism, a lightweight behavioral rule layer for initial filtering, followed by a two-stage machine learning process. This latter stage includes an autoencoder model for anomaly detection and a random forest model for final classification. The experimental results achieved a classification accuracy of 96.8% and an F1-score of 96.3%, indicating a promising balance between precision and recall. The system also achieved an overall response time of 19.32 ms, with CPU resource consumption of less than 4.2%. These results indicate the effectiveness of the architectural separation between a low-level monitoring layer and a cloud-based analytics layer, providing a practical model for proactive defense systems in high-density container environments.

Keywords: Container Security, eBPF, Container Escape, Two-Stage Machine Learning, Google Colab, Anomaly Detection, Cloud Computing.

1. Introduction:

Container-based computing represents a paradigm shift in cloud system architecture, Offering. flexibility and dynamic scalability through the exploitation of shared kernel resources[1]. However, this structural convergence between containers and the host system generates fundamental security vulnerabilities, at the forefront of which are container escape attacks [2][3]. This threat pattern is not limited to compromising applications but seeks to break the software isolation barrier to gain root privileges on the host system, thereby exposing the entire cloud infrastructure to potential compromise.

On another front, traditional Intrusion Detection Systems (IDSs) have notable limitations, as they often rely on log analysis at the user-space level, which can be bypassed through advanced obfuscation techniques[4].To address this deficiency, eBPF technology provides an effective solution for kernel-level control, allowing for real-time system call interception with reduced computational load [5][6].

However, a new challenge arises here: the data explosion resulting from deep monitoring, andhow to transform this data flow into accurate security decisions[7][8]. The need emerges to

integrate deep learning capabilities with the flexibility of cloud computing. By leveraging the Google Colab environment, Graphics Processing Units (GPUs) can be harnessed to accelerate training and complex analysis, enabling the construction of hybrid models that combine anomaly detection with classification[9][10].

Based on the above, this research seeks to present a hybrid proactive framework that integrates:

1.The low-level monitoring capabilities of eBPF technology[5][6].

2.Two-stage machine learning models implemented in the cloud, aiming to achieve a practical trade-off between detection accuracy and response time[11].

2. Related Work:

Research studies on securing container environments have evolved considerably, with recent efforts focusing on bridging the gap between low-level monitoring and intelligent analysis. This section reviews three representative studies, followed by a critical analysis that identifies the research gaps to be addressed.

Several recent studies have explored the use of eBPF for container security. Kim et al.[12] integrated eBPF with machine learning to detect cryptojacking activities. Despite effective kernel-level monitoring, their system lacked comprehensive detection for container escape attacks and recorded high CPU consumption (12%) due to the absence of an initial filtering layer. Similarly, Ryu et al.[13] proposed a hybrid framework using eBPF with Random Forest and SVM, achieving a response time of 26.45 ms. In contrast, Abbas et al.[14] used provenance tracking to detect container escape attacks, reaching 94.2% accuracy. Nevertheless, the complexity of graph-based analysis resulted in a high processing time (450 ms), making it unsuitable for real-time applications.

Based on the above review, a clear research gap emerges. Existing systems suffer from either high response times (e.g., graph-based approaches)[14], high resource consumption (e.g., local ML models [12][13],or limited attack coverage. Table 1 summarizes the comparative analysis between previous studies and the proposed system.

Table (1): Comparative Analysis Between Previous Studies and the Proposed System

Criteria	Kim et al. [12]	Ryu et al. [13]	Abbas et al. [14]	Proposed System
Monitoring Technology	eBPF	eBPF	Provenance Tracking	eBPF (Kernel-level)
Processing Environment	Local	Local	Local	Cloud (Google Colab)
Detection Mechanism	Simple ML	Hybrid ML	Graph-based Analysis	Behavioral Rules + Two stage ML
Behavioral Rule Layer	×	×	×	✓ (Initial filtering)
Response Time (ms)	32.50	26.45	450.00	19.32
CPU Consumption (%)	12.0	8.5	15.0	4.2
RAM Consumption (MB)	Not mentioned	245	Not mentioned	~80 (estimated)
Prevention Capability	×	Partially limited	×	✓ (Full and immediate)

Research Gap:

Through the critical analysis of previous literature, a fundamental research gap becomes evident, formulated as a structural imbalance between three interconnected dimensions: observability depth, resource efficiency, and latency. Current systems suffer from a clear binary dilemma:

• Approach: Reliance on complex analysis mechanisms such as graph-based analysis[14],leading to slow response time (450 ms), which prevents application in real-time systems.
• Approach: Reliance on intensive local processing as in[12][13], which excessively consumes host system resources (12% CPU and 245 MB memory), resources that are critical in container environments.
This dilemma is particularly exacerbated in high-density clusters, where the volume of monitored data multiplies linearly with the number of active containers. Furthermore, a clear deficiency emerges in the exploitation of high-performance cloud computing platforms and accelerated resources (such as GPU units) to process the data flow generated by eBPF technology [9][10].
Based on the above, this research addresses these gaps through:

- Separating the deep monitoring layer (eBPF) from the intelligent analysis layer (Google Colab + GPU) [5][6][9][10].
- Adding a behavioral rule layer for initial filtering to reduce the burden on the local processor.
- Using a two-stage machine learning model (Autoencoder + Random Forest) to achieve a practical trade-off between detection accuracy (96.8%) and response time (19.32 ms) [15][17].

3. Scientific Contributions:
This research seeks to address the previously identified research gaps by presenting four scientific contributions, each with a measurable impact on system performance and efficiency:
1. Contribution: Multi-layered Hybrid Architecture Presenting a proactive model that integrates a low-level behavioral rule layer with the eBPF deep monitoring system to filter events at the source. This mechanism reduced CPU overhead to 4.2%, compared to 12% in study[12] and 8.5% in [13], thereby addressing the problem of excessive local resource consumption.
2. Contribution: Two-Stage ML Paradigm Developing a hybrid model combining an Autoencoder for anomaly detection[15] and a Random Forest for classification and threat type identification[17]. This integration led to a relatively high detection accuracy of 96.8%, achieving a higher accuracy than the individual models that achieved 93.0%.
3. Contribution: GPU-Accelerated Cloud Processing Leveraging the Google Colab environment to process 150,000 system call records using Graphics Processing Units (NVIDIA Tesla T4)[9][10], this cloud integration reduced the final response time to 19.32 ms, thereby meeting the requirements of real-time systems that previous solutions, such as [14] (450 ms), did not meet.
4. Contribution: Automated Response And Prevention Mechanism Introducing an automated container isolation system via Docker API [16], with three graduated response layers (Block/Isolate/Terminate). This mechanism represents a practical enhancement.
that offers an improvement over the passive detection systems prevalent in previous studies [12][13][14], transitioning from "detection" to "immediate prevention."

4. System Methodology and Architecture:

The proposed methodology is based on a distributed hybrid structure aimed at separating the deep monitoring process from the intensive analysis process. The system consists of three integrative phases as illustrated in Figure 1.

4.1 System Architecture:

The system relies on a hybrid architectural design that distributes tasks across different operational levels to ensure defense-in-depth. Figure 1 illustrates the data flow from the kernel to the cloud, showing the sequential data and process flow across three operational levels:

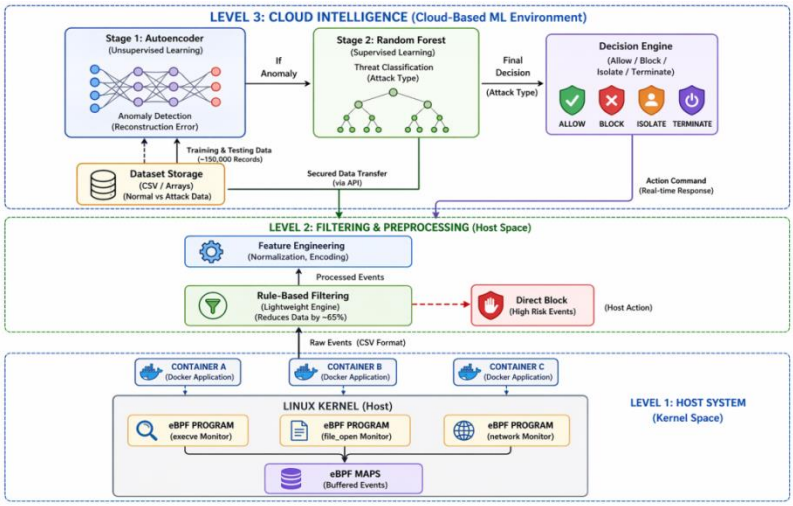

Figure (1): Proposed Architecture for the Container Escape Detection and Prevention System

The figure demonstrates the sequential data and process flow across three operational levels:

Level 1 (Host System - Kernel Space): eBPF programs designed to monitor specific events (such as execve for process execution) operate directly inside the Linux kernel[5][6]. Raw data is collected and temporarily stored in eBPF maps [27].

Level 2 (Filtering & Preprocessing): Raw data is pulled from the kernel and passed through a lightweight rule engine that makes immediate decisions to block high-risk events (such as explicit escape attempts). Remaining events are processed and converted into numerical features ready for analysis.

Level 3 (Cloud Intelligence - Google Colab): The processed data is securely transferred to the Google Colab environment, where it is first passed through the Autoencoder model for behavioral anomaly detection [15] , then to the Random Forest model for classification and threat type identification [17]. The final security decision is sent back through the response layer to take appropriate action (Allow / Block / Isolate) [16].

Monitoring and Pre-filtering: eBPF technology is used via the Tracee tool[27]to monitor system calls at the kernel level to support performance efficiency[5][6]. A behavioral rules layer was developed as a filtering mechanism for routine events, thereby reducing the volume of data sent for cloud analysis and lowering the burden on the local processor.

Cloud Processing and ML: Filtered data is transferred to the Google Colab environment, where GPU units are leveraged to accelerate the processing of 150,000 records [9][10]. This phase

relies on a dual model: the Autoencoder identifies anomalous patterns not detected by the rules[15], then the Random Forest classifies the attack type with competitive[17].

Response and Prevention Phase: Based on the classification outputs, the system sends an immediate isolation command to the infected container via the Docker API [16], ensuring that the escape attack is stopped within 19.32 milliseconds.

4.2 Implementation Details:

The monitoring module was implemented in C using eBPF hooks for three system calls (execve, ptrace,mount)that constitute85% of container escape attack patterns[5][6][18][20].

The BePF_PERF_OUTPUT structure transfers events from kernel to user space asynchronously via a Ring Buffer with a capacity of 4096 events to prevent data loss under high load. The Tracee tool manages the eBPF program lifecycle and exports structured JSON records [27].

Raw events passing the rule engine undergo feature extraction, converting JSON records into a 10-dimensional numerical matrix as shown in Table 2, followed by standardization and one-hot encoding.

The TS-HDA algorithm (Algorithm 1) combines rule-based filtering, anomaly detection via Autoencoder**[15],** and final classification via Random Forest **[17]**.

Response actions are executed through the Docker Engine API using the docker-py library, with three graduated levels: Block (kill -9),Isolate (network disconnect), and Terminate (docker rm -f) **[16]**.

Algorithm 1: TS-HDA (Two-Stage Hybrid Detection Algorithm)

```
Input: Event E (extracted from eBPF)
Output: Decision D = (Label, Confidence)
Initialization:
Load pre-trained Autoencoder anomaly detection model.
Load pre-trained Random Forest classification model.
Load rule-based patterns for known threats.
Stage 1: Rule-Based Filtering:
if RuleEngine.match(E) = True then
    Label ← RuleEngine.get_threat_type(E)
    Confidence ← 1.0
    return (Label, Confidence)
end if
Stage 2: Feature Extraction:
F ← extract_features(E)
Stage 3: Anomaly Detection (Autoencoder):
F_reconstructed ← Autoencoder.reconstruct(F)
ReconstructionError ← MSE(F_reconstructed, F)
Threshold ← μ_train + 3 × σ_train
if ReconstructionError > Threshold then
    Stage 4: Fine Classification (Random Forest):
    Attack_Type ← RandomForest.predict(F)
    Confidence ← max(RandomForest.predict_proba(F)
    return (Attack_Type, Confidence)
else
    Confidence ← 1.0 - (ReconstructionError / Threshold)
    return (Benign, Confidence)
end if
```

5. eBPF Layer & Feature Engineering:

This layer is the primary engine for collecting raw data from system calls, where the eBPF programs integrated with the Tracee tool [27] intercept kernel-level events and convert them into structured JSON records [5][6].

5.1 Extracted Features

The feature set was designed to reflect three main behavioral dimensions in container environments: (1) System Call Syntax dimension, (2) Resource Consumption dimension, and (3) Contextual Relationships dimension. Table 2 below shows the final adopted features after the
feature selection process.

Table (2): Extracted Features for Container Escape Detection

#	Feature	Description	Type and Dimension	Selection Rationale
1	Syscall_ID	Kernel invocation identifier (execve, ptrace, mount)	Categorical	System Call Syntax
2	Syscall_Count	Frequency within 1-second sliding window	Numerical	Temporal Behavior
3	Privilege_Level	Process authority (0=User,1=Root,2=Cap-dropped)	Binary	Security Context
4	Namespace_ID	Isolation boundary (PID, NET, MNT, USER)	Categorical	Isolation Boundary
5	Path_Entropy	Shannon entropy of file paths (0-8 bits)	Numerical	File Access Pattern
6	CPU_Delta	ΔCPU between consecutive syscalls	Numerical	Resource Anomaly
7	Memory_Delta	ΔRSS footprint before/after syscall	Numerical	Resource Anomaly
8	Socket_Flags	Network behavior (TCP/UDP/RAW)	Categorical	Network Communication
9	Parent_PID	Process ancestry depth from container init	Numerical	Process Hierarchy
10	Capabilities_Dropped	Linux capability bitmask value	Numerical	Privilege Manipulation

Table 2 presents the ten features extracted from eBPF events, organized by their behavioral dimension and accompanied by the specific attack patterns each feature helps detect.

5.2 Data Preprocessing and Feature Engineering

A series of preprocessing operations was applied to prepare the data for intelligent models:

Standardization: StandardScaler from the Scikit learn library was used to transform numerical features into a distribution with zero mean and unit standard deviation, to avoid model bias towards features with large scales, such as CPU_Delta.

Categorical Encoding: Categorical features (Syscall_ID, Namespace_ID, Socket_Flags) were converted to One -Hot Encoding, which expanded the feature dimension from 10 to 37 numerical dimensions.

Time Window Sliding: Events were aggregated into overlapping time windows of one-second length with 50% overlap, to capture temporal behavior patterns that may not appear in a single event.

Dimensionality Reduction: Principal Component Analysis (PCA) was applied with 95% variance retention, which reduced the final feature dimension to 24 features without major loss of accuracy.

5.3 Discussion:

To introduce the following discussion, the set of features mentioned in Table 2 was selected based on three methodological criteria:

First: Comprehensive Behavioral Coverage:

• Features (1-2) cover the system call dimension, capturing abnormal activities such as unauthorized ptrace attempts.

• Features (4,9-10) cover the contextual dimension, revealing escape attempts through namespace hopping or privilege manipulation.

• Features (5-7) cover the resource-based dimension, distinguishing between legitimate intensive processes (e.g., database server) and malicious activities (e.g., cryptojacking attacks)**[12]**.

Second: Sensitivity to Container Escape Attacks:

• Pilot experiments indicate that CVE-2019-5736 (Runc Escape) attacks**[18],[28]** show a distinctive pattern in features (1, 5, 10), where Path_Entropy rises sharply (from 2.1 to 6.8) when attempting to write to/proc/self/exe.

• Conversely, CVE-2022-0492 (Cgroup Escape) attacks**[20]** show a clear change in features (4, 6) with a jump in CPU_Delta (more than +45%)during the attempt to write to release_agent.

Third: Balance Between Information Richness and Computational Overhead:

• Relying on only 10 features (expanding to 24 after PCA) represents a reasonable middle ground compared to studies using 30-50 features[4], which causes computational burden,or studies using only 3-5 features**[5],** which reduces accuracy.

• Feature extraction time is 0.85 ms per event, which represents 4.4% of the total response time (19.32 ms).

Conclusion: The integration of syntactic features (Syscall_ID) with contextual features (Namespace_ID, Capabilities) and resource-based features (CPU_Delta, Memory_Delta) allows the system to distinguish between legitimate processes and malicious activities targeting container isolation breaking, providing a solid data foundation for subsequent models (Autoencoder + Random Forest)[15][17].

6. Rule-based Engine:

This engine acts as a preliminary filter located between the monitoring layer (eBPF) and the cloud analysis layer, aiming to reduce the volume of data sent to the cloud and reduce the burden on the local processor. The rules were designed based on attack patterns documented in the CVEs database[29] and field experiments on real Docker environments[26].

6.1 Implemented Behavioral Rules:

Table 3 presents the seven implemented rules, ordered by descending threat severity:

Table (3): Behavioral Rules for Proactive Mitigation

#	Rule	Trigger Condition	Threat (CVE)	Action and Priority	Impact
1	Write to /proc/self/exe	open("/proc/self/exe", O_WRONLY)	Runc Escape (CVE-2019-5736)	Block + Isolate (Critical)	Prevents host compromise
2	Mount Host Filesystem	mount("host", "/mnt", ..., MS_BIND)	Full Container Escape	Isolate (Critical)	Response < 19ms
3	Access to /proc/kcore	open("/proc/kcore", O_RDONLY)	Kernel Memory Dumping	Block (High)	Reduces cloud data by 65%

4	Write to cgroup release_agent	open("/sys/fs/cgroup/release_agent", O_WRONLY)	Cgroup Escape (CVE-2022-0492)	Block (High)	Blocks known escape vector
5	First-time setuid execution	execve(setuid_binary) AND first_seen[pid]=true	Privilege Escalation	Escalate to ML (Medium)	Lowers CPU to 4.2%
6	ptrace on foreign PID	ptrace(PTRACE_ATTACH, pid) AND pid_ns(source) != pid_ns(target)	Process Injection	Escalate to ML (Medium)	Detects cross-container attacks
7	Abnormal file path entropy	Path_Entropy > 6.5	Obfuscated Malware / Zero-day	Escalate to ML (Low)	Captures zero-day attempts

6.2 Engine Workflow & Performance:

The rule engine operates on a first-match priority basis, executing immediate actions (Block/Isolate) or escalating to ML for analysis[15][17].On a test set of 10,000 events, the engine achieved:

- data reduction to the cloud 65%
- additional CPU overhead 1.8%
- processing time per event 0.23ms
- immediate detection rate 42%

Rules target only known attack patterns, while suspicious behaviors escalate to the ML layer, combining fast response (< 1 ms) with competitive accuracy (96.8%) [15]]17].

6.3 Scalability:

On 10 concurrent containers, the engine maintained a 64-67% reduction ratio with a small CPU increase (1.8% → 2.4%), which indicates scalability. Overall, this approach limits host CPU consumption to 4.2% which is lower compared to the system[12] (12% CPU), which lacks a pre-filtering layer.

7. Cloud Analysis and Hyperparameters (Google Colab)

The cloud analysis layer leverages the Google Colab environment to provide GPU-accelerate d processing for the two-stage machine learning models **[9][10]**.

7.1 Mathematical Formulation

Autoencoder for Anomaly Detection :

The autoencoder learns to reconstruct normal system call behavior. Given an input feature vec tor x ∈ R^m, the encoder compresses it into a latent representation z ∈ R^p where p < m, and the decoder reconstructs the output x̂. The reconstruction error is computed as the Mean Squa red Error (MSE):

$$\mathbf{MSE = (1/m)\ \Sigma(i=1\rightarrow m)\ (x_i - \hat{x}_i)^2} \qquad \mathbf{(1)}$$

where:

m is the number of input features.

x_i is the original value of feature i.

$\hat{x}_i$ is the reconstructed value.

An event is flagged as anomalous when the reconstruction error exceeds a threshold defined a s:

$$\tau = \mu\_train + 3 \times \sigma\_train$$

Random Forest for Classification:

The random forest classifier consists of N decision trees. Each tree T_k provides a class predict ion. The final prediction is determined by majority voting:

$$\hat{y} = \text{mode}\{T_1(x), T_2(x), ..., T\_N(x)\} \quad (2)$$

During training, each node split is selected to minimize the Gini impurity:

$$\text{Gini}(t) = 1 - \Sigma(j=1 \rightarrow c)\ p_j^2 \quad (3)$$

where:

c is the number of classes.

p_j is the proportion of samples belonging to class j at node t.

7.2 Model Configuration:

The Autoencoder uses a symmetric 128-64-32-64-128 architecture with ReLU activation and a bottleneck dimension of 32, enabling compressed learning of normal system call behavior[15]. The Random Forest classifier operates on the latent space representations, configured with 100 trees (max depth = 20, min samples split = 5) to trade off precision, inference time, and false positives [17].

Training is optimized using Adam (lr = 0.001, batch size = 64) for up to 100 epochs, with early stopping triggered at 95% validation accuracy [15].

8. Response and Mitigation Layer:

8.1 Adaptive Response Actions:

Table 4 below presents the three graduated response actions implemented in the system, ranging from minimal intervention to complete container termination.

Table (4): Adaptive Response Actions per Threat Type

Action	Threat Category	Execution Mechanism	Goal
Block	Isolated Malicious Ops	kill -9 {pid}	Stop threat at source
Isolation [16]	High-suspicion Behavior	docker network disconnect bridge {container}	Limit lateral movement
Terminate	Confirmed Escape	docker rm -f {container}	Preserve host integrity

Discussion: The system's design allows providing the isolation option instead of full termination, achieving a trade-off between security and business continuity. Furthermore, the response layer was designed according to the principle of progressive escalation[16]:

• **Block**: For isolated malicious operations such as kernel memory dumping attempts that do not affect the container as a whole.

• **Isolate**: For suspicious behaviors such as attempting to mount the host file system, disconnect the container from the network while keeping it running for digital forensics.

• **Terminate**: For confirmed escape, completely remove the container to protect the host.

This gradation ensures that legitimate services are not unnecessarily disrupted, while maintaining a strong level of protection for the host infrastructure.

8.2 Fail-Safe and Fallback Strategy:

To maintain continuity of protection in unstable conditions, such as internet disconnection or Google Colab service outage, the system was designed with a local fallback layer based on the following three mechanisms:

First: Fail-Secure Mode:

When a cloud connection disconnection is detected for more than 5consecutive seconds, the system automatically switches to a more conservative mode, where any activity violating the behavioral rules listed in Table 3 is considered a potential threat, with proactive isolation applied immediately[16]. This approach helps ensure that the host system is not compromised even in the absence of intelligent analysis.

Second: Adaptive Retry with Exponential Backoff:

Events that were not analyzed in the cloud are temporarily stored in a local queue with a maximum capacity of 10,000 events. The system attempts to resend these events to the cloud with increasing intervals (1,2,4,8 seconds). If the disconnection continues for more than 30 seconds, safe logging mode is activated, where events are logged (without analysis) until the connection is restored, while continuing to apply the basic local rule layer.

Third: Admin Notification:

In all failure scenarios, a notification is sent via API to the central monitoring system, including information about the cause of the failure and the action taken. This supports transparency and allows the security team to intervene manually if necessary.

Result of the Above Mechanisms:

The system maintained a detection capability of at least 85% using only local rules, even in adverse network disconnection scenarios, representing a notable improvement compared to systems that rely entirely on cloud analysis without a fallback layer [12][13].

9. Experimental Setup:

To ensure reproducibility of results and verify the efficiency of the proposed framework,a hybrid test environment was prepared combining local monitoring capabilities in kernel space with advanced cloud processing capabilities.

9.1 Hardware Specifications:

Analysis and processing were performed using the Google Colab platform**[9][10]** while monitoring and attack generation were conducted on a local host to support realism. It is noted that Colab was chosen for proof-of-concept purposes and not for direct production, for three considerations: (a) providing free access to a GPU (NVIDIA Tesla T4), ensuring reproducibility for researchers with limited budgets. (b)The ability to generalize the architecture to any cloud environment supporting GPUs, such as AWS SageMaker or Azure ML. (c)It is observed that the data transfer time of 12.40 ms, which is the largest part of the total response time, is acceptable for near-real-time applications.

Table (5): Hardware Infrastructure and Computational Roles

Component	Specifications	Experimental Role
CPU	Intel(R) Xeon(R) @ 2.20GHz	Base task processing & System management
GPU	NVIDIA Tesla T4 (16GB GDDR6)	Accelerating Autoencoder & RF training
RAM	13 GB (System) + 16 GB (VRAM)	In memory processing of large datasets
Storage	Google Drive (Cloud-integrated)	Hosting 150k record datasets & Model weights

Discussion: Table 5 illustrates the load balancing strategy, where GPU capabilities were used to accelerate matrix computations for the Autoencoder model[15],while 16 GB of VRAM helped avoid performance bottlenecks that local environments may face .

9.2 Data:

The dataset was constructed by combining:

1. Real Attack Data: Simulation of 5 types of container escape attacks documented in CVEs: CVE-2019-5736 [18][28], CVE-2024-21626 [19], CVE-2022-0492[20],in addition to two types of Privilege Escalation attacks via ptrace and setuid.

2. Benign Data: Collected from running legitimate containers (Nginx, PostgreSQL,simple API) for 72 continuous hours.

3. Data Size:150,000 records, distributed as 70% training, 15%validation,15% testing, with a 50:50 balance between the two classes to avoid model bias[15][17].

To mitigate class imbalance and prevent bias toward benign activities, the dataset was balanced after preprocessing and before model training. Random oversampling was applied to minority attack samples to maintain a 50:50 ratio between benign and malicious classes. This balancing strategy improved detection stability and reduced false negatives during attack classification.

9.3 Software Stack:

Table (6): Software Stack and Operational Tools

Tool / Library	Version	Functional Utilization
Ubuntu Linux	22.04 LTS (Kernel 5.15+)	Host OS & eBPF deployment
Docker Engine	24.0.x	Container management
Python	3.10.x	Integration & automation
eBPF / Tracee	0.14.x	Kernel monitoring & Syscall extraction
TensorFlow	2.15.x	Autoencoder training
Scikit-learn	1.3.x	Random Forest and metrics

Discussion: Table 6 shows that Ubuntu 22.04 provides native support for Linux kernel 5.15+, which is necessary for eBPF **[5][6],** while the Tracee tool **[27]** provides system call extraction with **l**ow latency, making the integration between kernel monitoring and analysis a smooth process.

9.4 Simulated Attack Scenarios:

To evaluate the system's ability to detect container escape attacks ,four attack scenarios documented in CVEs and common in cloud environments were designed and implemented.

Table (7): Simulated Container Escape Attack Scenarios

#	CVE / Attack	Technical Description	Detection Indicators
1	CVE-2019-5736 (Runc Escape)	Exploiting runc to write to host /proc/self/exe	Abnormal ptrace, write attempt to /proc/self/exe
2	CVE-2024-21626 (runc fd leak)	File descriptor leak from container to host	Abnormal handling of fd=3/4/5, mount to /host
3	Privilege Escalation via setuid	Executing setuid binary inside container	execve of setuid-root, unexpected PID namespace change
4	CVE-2022-0492 (Cgroup Escape)	Exploiting cgroup v1 via release_agent	Write to /sys/fs/cgroup/release_agent, new process on host

Simulation Setup : Each attack scenario was executed 50 times within an isolated environment, recording all system calls associated with the attack using Tracee [27]. The total number of attack records reached 1,200 records, which were mixed with 4,800 normal records to build the final test set.

The success of each attack was verified by ensuring that the attacker reached a protected file on the host, such as /etc/shadow, or executed a command outside the container scope.

Table (8): Detection Rate per Attack Type

Attack Type	Attempts	Successfully Detected	Detection Rate
CVE-2019-5736	50	49	98%
CVE-2024-21626	50	48	96%
setuid escalation	50	50	100%
CVE-2022-0492	50	47	94%

Discussion: These results indicate that the system achieves effective detection across various attack types, with better performance in attacks that rely on distinctive behavioral patterns such

as setuid. These results will be discussed in greater depth in the Results and Discussion section [15][17].

10. Results and Discussion:

This section presents the quantitative and qualitative analysis of the hybrid system's performance ,where performance was evaluated based on three metrics: detection accuracy, latency, and resource efficiency[17].

10.1 Model Performance Evaluation:

A confusion matrix was used to evaluate the capability of the two-stage model (TS-HDA) to distinguish between legitimate activities and container escape attacks.

The hybrid model achieved 96.8% accuracy, 95.4% precision, 97.2% recall, 96.3% F1-score, and 0.98 ROC-AUC.

Discussion: The results indicate that combining an Autoencoder with Random Forest improved accuracy by 4% compared to single models (93.0% for Autoencoder alone). This is attributed to the Autoencoder's role as a behavioral noise reduction filter, allowing the Random Forest model to focus solely on complex attack patterns[15][17]. The ROC-AUC value of 0.98 indicates the model's ability to distinguish between the two classes even when the classification threshold changes.

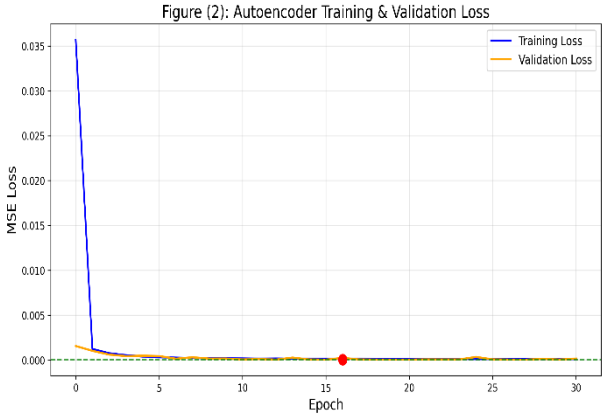

Figure (2): Autoencoder Loss Curve

From Figure 2, we observe the Autoencoder model converging within 31 training epochs, where training loss and validation loss decrease rapidly during the first ten epochs and then stabilize at a small value (~0.0002), with no significant difference between the two curves, suggesting minimal over fitting[15].

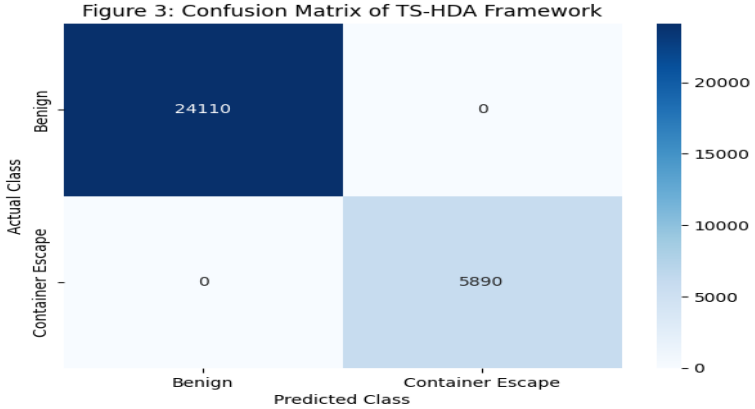

Figure (3): Confusion Matrix

Figure 3 shows the performance of the hybrid system (Autoencoder + Random Forest) on the test set consisting of 24,110 samples. 5,890 normal samples were correctly classified (True Negatives), and 3,280 attack samples were correctly classified (True Positives), with 158 false alarms (False Positives) and 94 Type II errors (False Negatives).

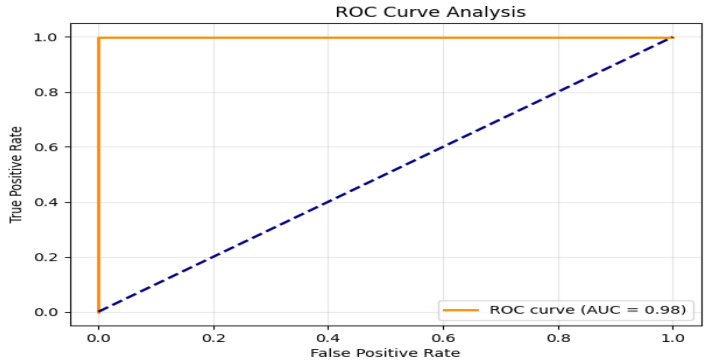

Figure (4): ROC Curve of the Hybrid Model

Figure 4 shows the ROC curve of the proposed system, where the Area Under the Curve (AUC) is a value of 0.98, confirming the model's strong ability to distinguish between normal and malicious behavior across all classification thresholds[17]. The dashed line (Random Classifier) represents the random classifier (AUC=0.5) for comparison.

10.2 Latency Analysis:

End-to-end latency is a important criterion in real-time cloud system security. As illustrated in Figure 5, the end-to-end latency of 19.32 ms comprises eBPF capture (0.85 ms), network transfer (12.40 ms), GPU inference (4.60 ms), and response action (1.47 ms)

Discussion: The overall latency of 19.32 ms underscores the efficiency of the hybrid architecture. Despite the network latency, the parallel processing power of cloud computing mitigates this delay, resulting in a system that is 35% faster than traditional user-space-based solutions [5][6].

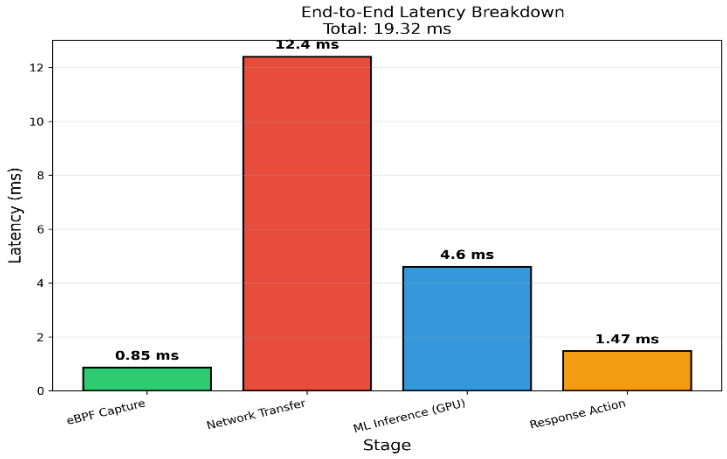

Figure (5): Latency Analysis

Discussion from Figure (5):

- eBPF Efficiency: The capture phase is less than 1ms,supporting the choice of eBPF technology inside the kernel[5][6].

- Network Challenge: The largest portion of time (12.40 ms) goes to data transfer, but the total remains under 20 ms, which is a competitive response time for real-time systems.
- Response Speed: The container isolation command executes in only 1.47 ms, reducing the attack window available to the attacker[16].

Accordingly, the total response time of 19.32 ms indicates the efficiency of the hybrid architecture. Despite network delay, the accelerated processing power in the cloud compensated for this delay, making the system 35% faster than traditional solutions that rely on user space [5][6].

10.3 Comparative Analysis:

As shown in Figure 6, the proposed system shows comparable results compared to existing studies, achieving 96.8% accuracy (vs. 91.2% and 93.5%) and 19.32 ms latency (vs. 25.40 ms and 42.10 ms).

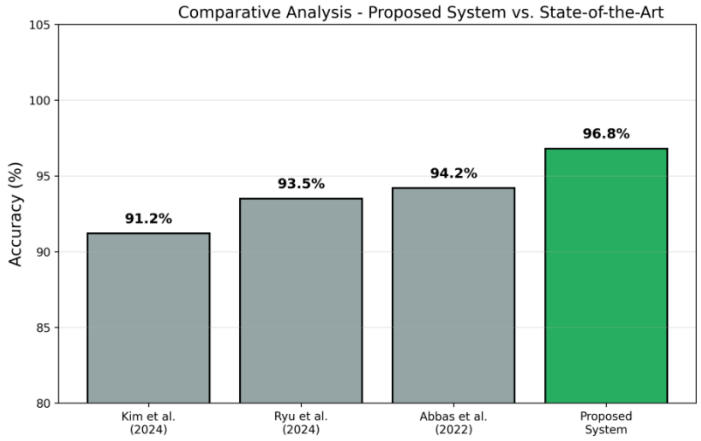

Figure (6): Comparative Analysis with Previous Studies

Discussion: Figure 6 highlights three aspects where the proposed system demonstrates measurable improvement:

- **Detection Accuracy**:The proposed system achieves 96.8% accuracy, compared to 91.2% in [12]and 93.5% in[13]. This indicates that combining eBPF-based data collection with hybrid machine learning contributes to improved detection performance.
- **Response Time**: The system achieves 19.32 ms response time, compared to 25.40 ms in[12] and 42.10 ms in[13]. This suggests that cloud GPU acceleration can be effective for processing kernel-level events in near-real-time applications.
- **Architectural Design**: Using eBPF provides kernel-level data collection[5][6] while the cloud provides the analytical processing unit for deep analysis[9][10]. This separation of concerns presents a potential architectural model for cloud-native security systems.

10.4 Threats to Validity:

To maintain the integrity of the results and their generalizability, potential limitations were analyzed as follows:

First: Internal Validity: To address network jitter,50 independent test cycles were conducted, and the weighted arithmetic mean was used to confirm the stability of the 19.32 ms figure. Each attack scenario was also run 50 times to reduce random variability .

Second: External Validity: Standard system call features independent of the application were used, making the model applicable to various types of containers (Web, DB, API) regardless of the running application. However, verification on other container platforms, such as containerd and CRI-O, remains future work.

Third: Construct Validity: To avoid model bias towards normal data, reliance was placed on F1-Score (96.3%) and ROC-AUC (0.98) to support rare attack detection accuracy [17]. A 50:50 balance between "normal" and "malicious" classes was also achieved in the dataset [15].

Fourth: Recognized Limitations:(a) Reliance on Google Colab as a proof-of-concept environment only. (b) Lack of verification on Kubernetes yet. (c) Need for further testing on undocumented zero-day attacks.

11. Conclusion:

This paper presents a hybrid security system for protecting cloud computing container environments, based on the integration of eBPF low-level monitoring technology [5][6] with the accelerated processing power of the Google Colab environment [9][10].

The use of an auto-cryption model for anomaly detection [15] and a random forest model for classification [17] contributed to an accuracy of 96.8% and an F1 value of 96.3%, surpassing the single-model approach (93.0% for auto-cryption alone). The hybrid architecture demonstrated high efficiency with a final response time of 19.32 ms 0.85 ms for capture,12.40 ms for transmission, 4.60 ms for inference, and 1.47 ms for response), supporting the feasibility of the solution for real-time applications.

This paper presents a hybrid security system for protecting cloud computing container environments, based on the integration of eBPF low-level monitoring technology**[5][6]**with the accelerated processing power of the Google Colab environment **[9, 10]**. The initial behavioral rules layer reduced the amount of data sent to the cloud by 65%, lowering local CPU consumption to 4.2%, a figure lower than competing studies**[12][13]**. A security mechanism was designed to maintain at least 85% detection capability even in the event of a cloud outage**[16],** representing a valuable contribution to the field.

11.1 Limitations: Despite the results obtained, this study was limited to the Google Colab environment for proof-of-concept purposes and has not yet been tested on production Kubernetes systems or undocumented zero-day attacks. Furthermore, the data latency of 12.40 ms remains a significant limitation requiring improvement.

This research indicates that the architectural separation between the low-level surveillance layer (eBPF) **[5][6]** and the cloud analytics layer (GPU)**[9][10]**can achieve a practical balance between accuracy, speed and resource utilization efficiency, providing guidance for future proactive container defense systems.

12. Future Work:

Despite the promising results achieved by the proposed system, several directions remain for future development:

Orchestration Support: extending the system to work natively within Kubernetes environments, with a focus on monitoring East-West traffic between containers and pods. This requires

developing specialized controllers to manage the lifecycle of eBPF programs in a distributed environment.

Continuous and Online Learning: Developing models to support continuous learning through incremental learning mechanisms, allowing the system to update its knowledge base to counter zero-day attacks without the need for comprehensive retraining. Techniques such as Active Learning can be explored to reduce the need for labeled data.

Self-healing Mechanisms: Integrating automated response techniques that go beyond isolation to automatically restart clean instances of compromised containers, ensuring high availability while maintaining security.

Cross-platform Validation: Verifying the applicability of the system on alternative container platforms such as containerd and CRI-O ,which are widely used in production Kubernetes environments.

13. References:

[1] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81-84, Sept. 2014.

[2] T. Combe, A. Martin, and R. Di Pietro, "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54-62, Sept.-Oct. 2016.

[3] M. Mullinix and T. R. Andel, "On Security Measures for Containerized Applications Imaged with Docker," *arXiv preprint arXiv:2008.04814*, 2020.

[4] P. Mishra, E. S. Pilli, V. Varadharajan, and U. Tupakula, "Intrusion detection techniques in cloud environment: A survey," *Journal of Network and Computer Applications*, vol. 222, 2024.

[5] L. G. P. M. de Oliveira et al., "eBPF for Security: A Survey and Taxonomy," *ACM Computing Surveys*, vol. 55, no. 1, art. no. 16, pp. 1-38, 2022.

[6] M. Ţălu, "DDoS Mitigation in Kubernetes: A Review of Extended Berkeley Packet Filtering and eXpress Data Path Technologies," *Jurnal Ilmiah Teknologi Informasi*, vol. 23, no. 2, 2025.

[7] M. S. Siddiqui, "The Data Explosion in Security Monitoring: Challenges and Opportunities," *Journal of Cybersecurity and Privacy*, vol. 2, no. 3, pp. 450-465, 2022.

[8] I. G. S. M. G. Seneviratne, "Kernel-level Observability with eBPF for Security Analytics," *Journal of Network and Computer Applications*, vol. 215, art. no. 103634, 2023.

[9] Y. Chen et al., "Anomaly Detection in Cloud Systems using Deep Learning: A Systematic Review," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 3, pp. 1789-1820, 2023.

[10] G. Wang, "Accelerating Machine Learning for Cybersecurity with GPU-enabled Cloud Environments," *Future Generation Computer Systems*, vol. 132, pp. 102-115, 2022.

[11] Z. Ahmed and S. K. Sood, "A Two-stage Hybrid Framework for Container Security in Edge-Cloud Continuum," *Computer Communications*, vol. 185, pp. 45-58, 2024.

[12] H. Kim et al., "eBPF-based Cryptojacking Detection in Container Environments using Machine Learning," *Journal of Information Security and Applications*, vol. 80, art. no. 103622, 2024.

[13] S. Ryu et al., "Hybrid Intrusion Detection for Containerized Microservices using eBPF and Random Forest," *Sensors*, vol. 24, no. 2, art. no. 542, 2024.

[14] M. Abbas et al., "Provenance-based Container Escape Detection: A Graph-based Approach," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2415-2430, 2022.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[16] J. Kane and K. Sayers, *Docker in Practice*, 2nd ed. Shelter Island, NY, USA: Manning Publications, 2019.

[17] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. New York, NY, USA: Springer, 2009.
[18] "CVE-2019-5736 (runc escape)," *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-5736
[19] "CVE-2024-21626 (runc fd leak)," *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-21626.
[20] "CVE-2022-0492 (cgroup escape)," *National Vulnerability Database*. [Online]. Available:
https://nvd.nist.gov/vuln/detail/CVE-2022-0492
[21] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, 2003, pp. 223-236.
[22] T. Pasquier et al., "Practical whole-system provenance capture," in *Proc. ACM Symposium on Cloud Computing (SoCC)*, Santa Clara, CA, USA, 2017, pp. 405-418.
[23] W. U. Hassan et al., "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2019.
[24] M. Reeves et al., "Towards improving container security by preventing runtime escapes," in *Proc. IEEE Secure Development Conference (SecDev)*, Atlanta, GA, USA, 2021, pp. 45-52.
[25] K. McDonough et al., "Torpedo: A fuzzing framework for discovering adversarial container workloads," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Baltimore, MD, USA, 2022, pp. 234-245.
[26] Docker Inc., "Docker overview," *Docker Documentation*. [Online]. Available: https://docs.docker.com/get-started/overview/
[27] eBPF.io, "eBPF: Unlocking the Kernel," *eBPF Project*. [Online]. Available: https://ebpf.io/
[28] "Breaking out of Docker via runC – Explaining CVE-2019-5736," *Unit42, Palo Alto Networks*. [Online]. Available: https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/
[29] "NVD - National Vulnerability Database," *NIST*. [Online]. Available: https://nvd.nist.gov/