

A Shell Programming Framework for Automated Risk Assessment and Security Control in Linux Operating Systems

R. Somadder*

Abstract

Linux operating systems are widely used in servers, embedded platforms, cloud infrastructure, and high-performance computing environments, making continuous system monitoring and risk assessment essential for maintaining security and operational reliability. Conventional risk assessment approaches often rely on individual system commands and manual interpretation of processor, memory, process, file, and security information, which can limit timely identification of abnormal system conditions. This paper proposes a Bash shell programming-based automated risk assessment framework for Linux operating systems integrated with an isomorphic graph model for risk mitigation. The proposed framework automatically collects and analyzes system-level parameters including processor utilization, memory consumption, running processes, file permissions, user activity, network services, and system logs. The collected parameters are mapped to Low, Medium, and High (L/M/H) risk levels and subsequently associated with Preventive, Detective, and Corrective (PC/DC/CC) security controls. The isomorphic representation establishes relationships among system resources, risk levels, and security control mechanisms, providing a structured approach to automated risk identification and mitigation. The framework is implemented using Bash shell scripts and evaluated under controlled Linux system conditions to examine its monitoring capability, risk classification, and computational overhead. The proposed approach provides a lightweight and transparent mechanism for automating Linux security assessment while retaining the mathematical structure of the isomorphic risk model.

Keywords: Linux operating system, Bash shell programming, risk assessment, isomorphic graph, system security, risk mitigation, preventive control, detective control, corrective control

INTRODUCTION

Linux-based operating systems have become an important platform for servers, cloud infrastructures, embedded systems, networking environments, and high-performance computing because of their flexibility, modularity, portability, and extensive command-line administration capabilities. As the number and complexity of Linux-based computing resources increase, maintaining system availability, integrity, confidentiality, and reliability requires systematic monitoring and continuous assessment of

*Author for Correspondence

R. Somadder

E-mail: Ronie.somadder@gmail.com

Assistant professor, Mangalmay Institute of Management and Technology, Greater Noida, Uttar Pradesh, India

Received Date: September 26, 2026

Accepted Date: September 28, 2026

Published Date: September 29, 2026

Citation: R. Somadder. A Shell Programming Framework for Automated Risk Assessment and Security Control in Linux Operating Systems. Journal of Advances in Shell Programming. 2026; 13(2): 55–66p.

operating system activities. Security monitoring is particularly important because abnormal processor utilization, memory consumption, process execution, file modifications, unauthorized access, network activity, and system-log events may provide indicators of operational or security-related anomalies [1–4]. The development of host-based security mechanisms has consequently emphasized the collection and analysis of operating system activities, audit records, system calls, and user behavior [5–7].

LITERATURE REVIEW

The importance of monitoring operating system activity can be traced to early intrusion detection research. Denning proposed an intrusion detection model based on monitoring audit records and identifying abnormal patterns of system usage [5]. Subsequent research extended this concept by examining program behavior, user profiles, Unix system calls, and shell-command activities for anomaly detection [6]. Host-based intrusion detection systems have subsequently used operating system audit trails and system logs as important sources of security information [7]. More recent research has investigated system-call-based datasets and behavioral analysis for detecting anomalies in Linux kernels, demonstrating the continued importance of operating system-level observations in security research [8]. These studies indicate that information generated directly by an operating system can provide useful evidence for identifying abnormal behavior.

Linux provides several native mechanisms for obtaining such information. System utilities can expose processor activity, memory utilization, running processes, file system information, active users, network connections, and system events. In addition, Linux auditing mechanisms can record security-relevant events associated with system calls, users, objects, and operation results [9]. Operating system logging has also become an established component of security monitoring because logs can support system tuning, auditing, incident investigation, and intrusion detection [10]. National Institute of Standards and Technology (NIST) guidance emphasizes the importance of systematic log management for generating, collecting, storing, accessing, and analyzing security-related event information [11]. Similarly, continuous monitoring frameworks emphasize maintaining visibility into organizational assets, threats, vulnerabilities, and the effectiveness of implemented security controls [12]. Therefore, an effective Linux risk assessment mechanism should consider multiple system-level information sources rather than relying on a single security indicator.

Although sophisticated security monitoring platforms and host-based intrusion detection systems can provide extensive analytical capabilities, they may require additional configuration, software components, databases, or computational resources. For lightweight Linux environments, a command-line-based approach can provide a comparatively simple mechanism for combining existing operating system utilities. Shell programming is particularly relevant in this context because shell scripts can execute system commands, process their outputs, apply conditional logic, automate repetitive activities, and generate consolidated reports. Research on system-administration automation has identified automation as an important mechanism for reducing repetitive administrative work and improving operational efficiency [13]. Recent studies have likewise examined shell scripting for Linux administration, monitoring, security automation, and large-scale system management [14–16]. Bash is especially suitable for this purpose because it provides a programmable interface through which existing command-line utilities can be combined into reusable automated workflows [17].

File system monitoring is another important component of Linux security assessment. Configuration files, executable programs, authorization information, and user data are critical operating system resources, and unauthorized modifications to these resources can indicate system compromise or configuration errors. The Tripwire approach demonstrated the value of monitoring Unix file systems for added, deleted, and modified files and established file-integrity monitoring as an important host-security mechanism [18]. Such monitoring can be incorporated into shell-based assessment frameworks through file-permission checks, ownership verification, configuration-file inspection, and comparison of selected system states.

The source model underlying the present research provides an additional theoretical foundation for organizing these observations. The uploaded work identifies the operating system as comprising major components, including the file system, shell, and kernel, and emphasizes the relationships among the processor, memory, encryption, and operating system control mechanisms [19–22]. The model further defines processor, memory, and encryption as interconnected system-resource parameters and relates them to risk mitigation and control mechanisms.

The source also represents security states through the sets $R = r, w, x$, $R = \{r, w, x\}$, $M = L, M, H$, $M = \{L, M, H\}$, and $S = P, M, E$, $S = \{P, M, E\}$, corresponding to read/write/execute operations, low/medium/high risk levels, and processor/memory/encryption parameters. Furthermore, the preventive, detective, and corrective control framework is associated with the processor, memory, and encryption through the proposed isomorphic model [23, 24].

Therefore, the present study proposes a Bash shell programming-based automated risk assessment framework for Linux operating systems integrated with an isomorphic graph model for risk mitigation. Instead of requiring an administrator to execute and interpret individual commands separately, the proposed framework combines multiple Linux system utilities through Bash scripts. The framework is designed to collect processor utilization, memory utilization, process activity, file and permission information, user/login activity, network service information, and relevant system-log data. These observations are then processed using predefined risk rules and mapped to Low (L), Medium (M), and High (H) risk categories. The resulting risk state is subsequently associated with Preventive Control (PC), Detective Control (DC), and Corrective Control (CC) mechanisms based on the isomorphic relationship proposed in the underlying model.

The proposed approach differs from conventional shell-automation studies because its objective is not limited to automating administrative commands. Instead, shell programming is used as an implementation layer for a structured risk assessment model. This distinction is important because recent work has demonstrated the usefulness of Bash and shell scripts for Linux administrative automation, user-access control, system monitoring, and security-related tasks [14–16], while host-based security research has investigated more complex statistical, machine-learning, and system-call-based detection mechanisms [6, 8]. The proposed framework seeks to occupy an intermediate space in which a lightweight command-line implementation can systematically transform operating system observations into an interpretable risk assessment.

Accordingly, the main objective of this research is to develop a Linux-based shell programming framework that automatically monitors selected operating system and security parameters and maps the resulting observations into a structured risk mitigation model.

The specific objectives are as follows: (i) to develop Bash shell scripts for automated Linux system and security monitoring; (ii) to collect processor, memory, process, file, user, network, and log information through Linux command-line utilities; (iii) to establish a Low/Medium/High risk classification mechanism; (iv) to integrate the classification with preventive, detective, and corrective controls using the proposed isomorphic model; and (v) to experimentally evaluate the framework under controlled Linux operating conditions in terms of detection capability, classification behavior, execution time, and resource overhead [20].

Therefore, the contribution of this study is threefold. First, it provides a lightweight Bash-based implementation for automated Linux system risk assessment. Second, it converts heterogeneous operating system observations into an organized risk classification and control framework. Third, it provides an executable implementation of the isomorphic risk model rather than treating the model only as a theoretical representation. The proposed framework can subsequently be extended to continuous monitoring, centralized logging, machine-learning-based anomaly detection, and automated corrective actions [21].

RESEARCH METHODOLOGY

Proposed Linux–Bash Risk Assessment Framework

The proposed methodology develops a Bash shell programming framework for automated risk assessment in Linux operating systems. The framework collects system-level information related to processor utilization, memory usage, running processes, file permissions, network activity, user activity, and system logs. The source work already establishes processor, memory, encryption, and security control relationships within its isomorphic risk model (Figure 1).

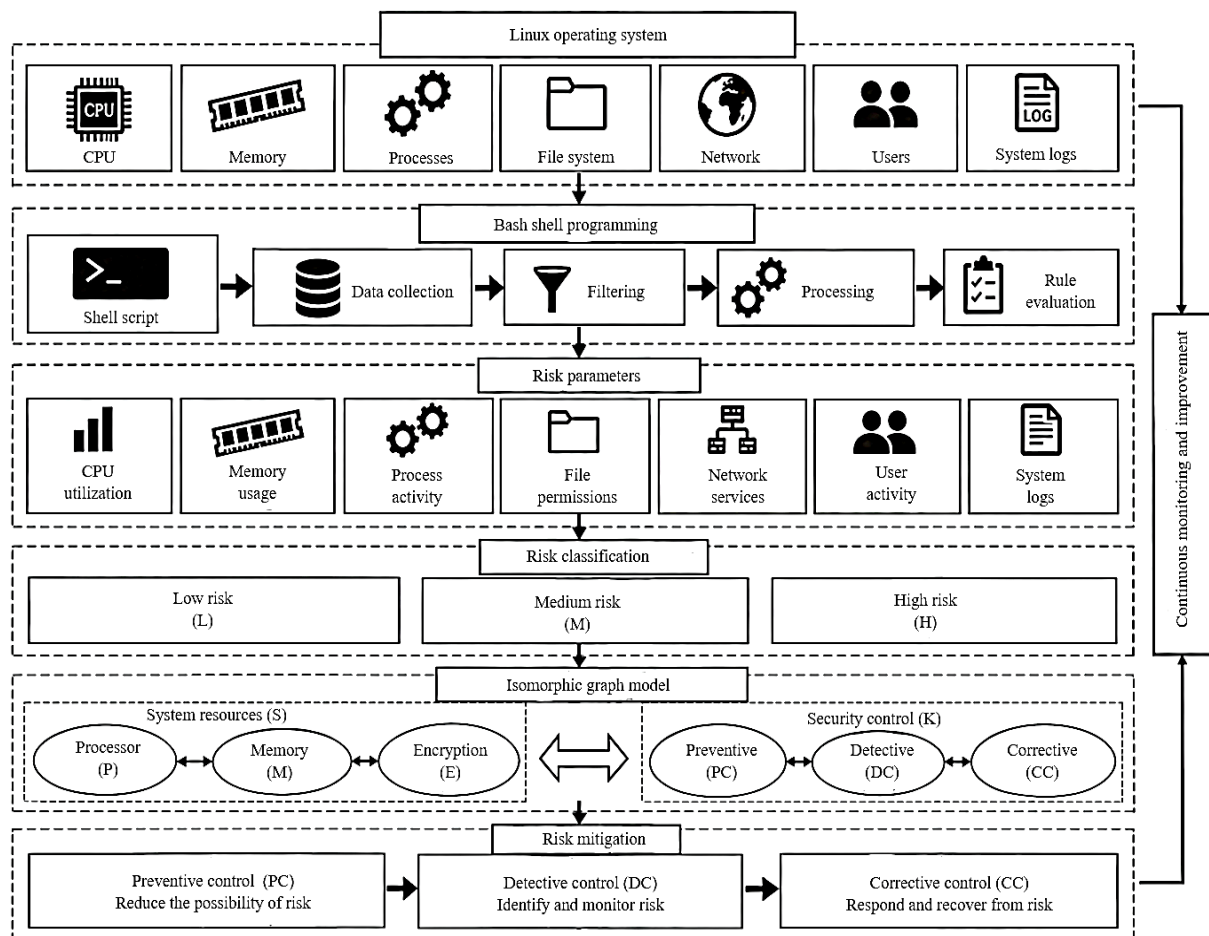


Figure 1. Proposed Linux–Bash risk assessment framework.

Source: Proposed by authors based on the isomorphic risk-control framework in the source work and Linux system monitoring principles [1–18].

Table 1. Overview of Linux utilities for system monitoring and security analysis.

Parameter	Example Linux utility	Assessment purpose
Processor	iostat, top	CPU utilization
Memory	free, vmstat	Memory and swap activity
Processes	ps	Process activity
Files	find, ls, stat	Permission and ownership
Network	ss	Listening services/connections
Users	who, last	Login activity
Logs	journalctl, Linux Audit	Event/security analysis

Linux System Monitoring and Bash-Based Data Acquisition

The second stage uses Bash shell programming to automate the collection of Linux system information. Instead of executing individual commands manually, the proposed framework integrates system utilities into a unified shell-based workflow (Table 1).

The original work already employs commands such as iostat, vmstat, sar, ps, lsof, and who for practical risk investigation. The proposed methodology extends this concept by integrating these monitoring operations through Bash programming. A modular implementation is illustrated in Figure 2.

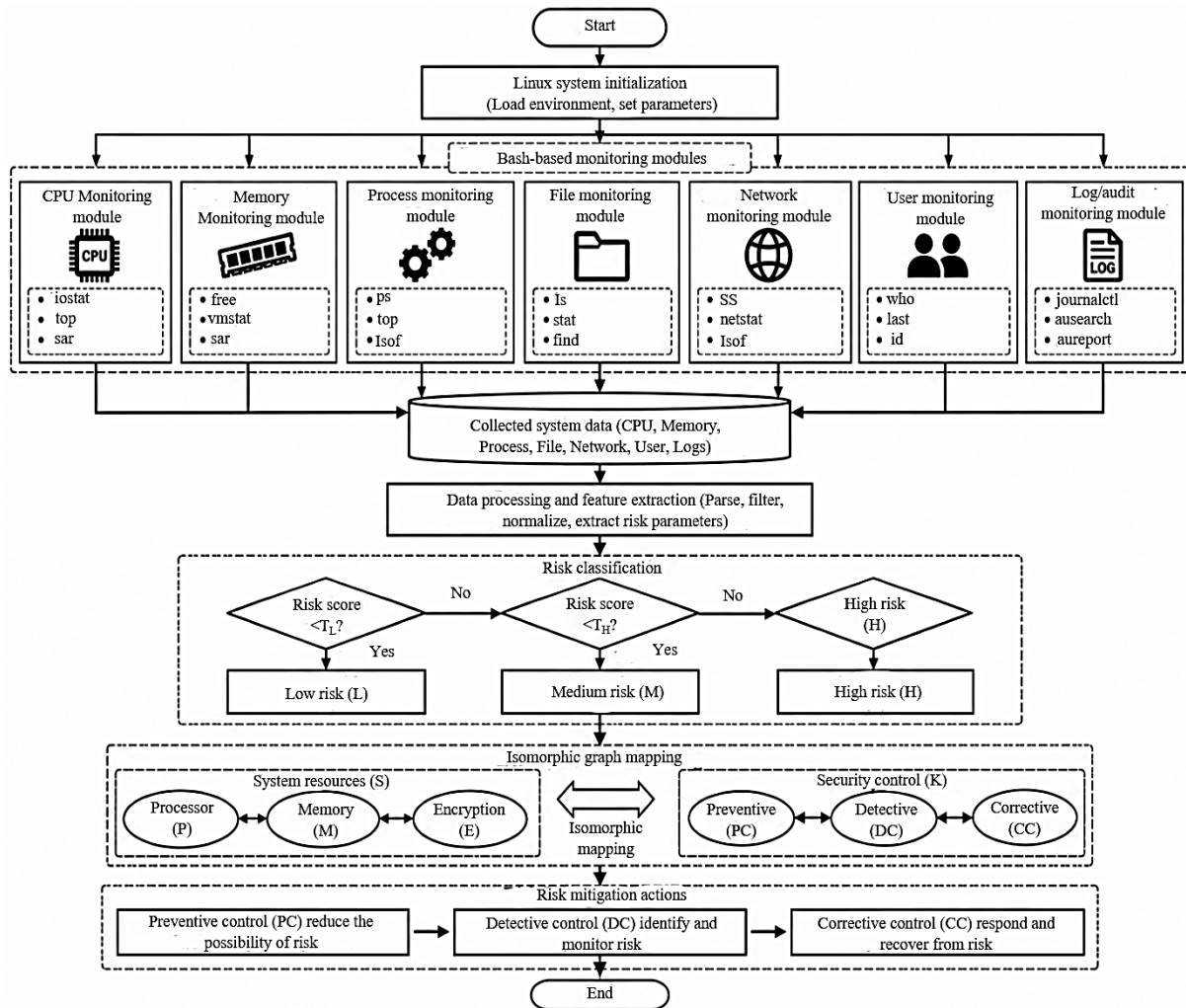


Figure 2. Modular Bash-based Linux monitoring architecture.

Risk Classification and Isomorphic Mapping

The collected Linux parameters were converted into risk states. Let the monitored parameters be represented by

$$X = X_{CPU}, X_{MEM}, X_{PROC}, X_{FILE}, X_{NET}, X_{USER}, X_{LOG} = \{X_{CPU}, X_{MEM}, X_{PROC}, X_{FILE}, X_{NET}, X_{USER}, X_{LOG}\} \quad (1)$$

Each parameter is transformed into a corresponding risk state:

$$R_i = f_i(X_i) \quad (2)$$

Where

$$R_i \in L, M, H, R_i \in \{L, M, H\} \quad (3)$$

and LL, MM, and HH represent low, medium, and high risk, respectively.

The overall system risk can be represented as:

$$R_{system} = F(R_{CPU}, R_{MEM}, R_{PROC}, R_{FILE}, R_{NET}, R_{USER}, R_{LOG}) R_{system} \\ = F(R_{CPU}, R_{MEM}, R_{PROC}, R_{FILE}, R_{NET}, R_{USER}, R_{LOG})$$

The isomorphic component is derived from the source model, which represents the operating system permission set as:

$$R = r, w, xR = \{r, w, x\} \quad (4)$$

and the risk set as

$$M = L, M, H. M = \{L, M, H\}$$

The source work establishes mappings between these sets and subsequently connects system-resource parameters with control mechanisms.

For the proposed Linux framework, the mapping is extended toward:

The PC/DC/CC framework follows the source model, where preventive, detective, and corrective controls are associated with the risk mitigation lifecycle (Figure 3).

Experimental Validation and Performance Evaluation

The final stage validates the proposed Bash framework under controlled Linux conditions. The experiments should evaluate whether the scripts correctly identify changes in system conditions and map them to the predefined risk categories. The proposed test cases are presented in Table 2.

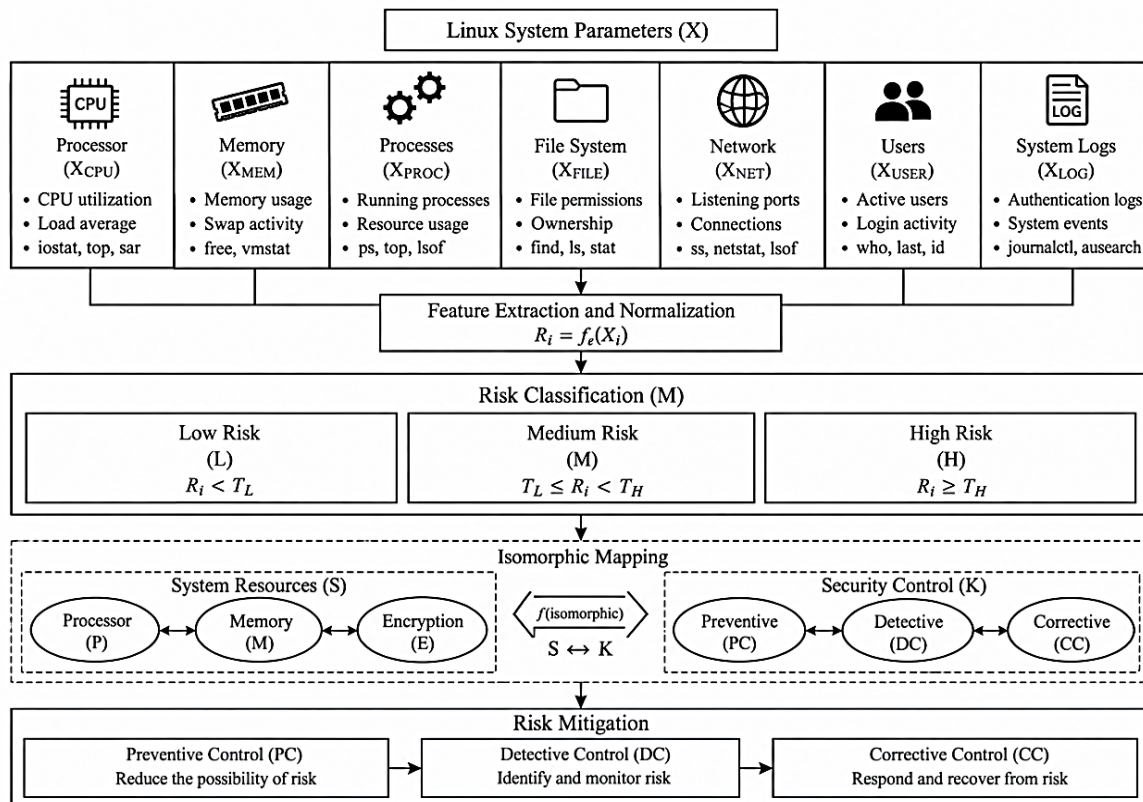


Figure 3. Proposed isomorphic mapping for Linux risk assessment.

Table 2. Controlled experimental conditions for Linux system testing.

Test	Controlled condition	Main parameter
E1	Normal Linux operation	Baseline
E2	Controlled CPU load	CPU
E3	Controlled memory load	Memory
E4	Controlled file-permission change	File
E5	Controlled test network service	Network
E6	Controlled login/log event	User/Log

The evaluation should measure the following parameters:

Detection Rate

$$DR = \frac{N_{detected}}{N_{test}} \times 100 \quad (5)$$

False-Positive Rate

$$FPR = \frac{N_{false\ positive}}{N_{normal}} \times 100 \quad (6)$$

Script Execution Time

$$T_{execution} = T_{end} - T_{start}$$

CPU and memory overhead of the Bash framework should also be recorded.

The experimental workflow is illustrated in Figure 4. This four-subheading structure keeps the methodology compact while still covering framework design, Bash implementation, mathematical/isomorphic modelling, and experimental validation (Figure 4).

BASH SHELL PROGRAMMING IMPLEMENTATION

The proposed framework was implemented using Bash shell programming in a Linux environment to automate system monitoring, risk classification, and generation of an assessment report. The implementation translates the conceptual risk model into an executable command-line framework. The original model identifies processor, memory, encryption, file permissions, and control relationships as important elements of the risk mitigation structure.

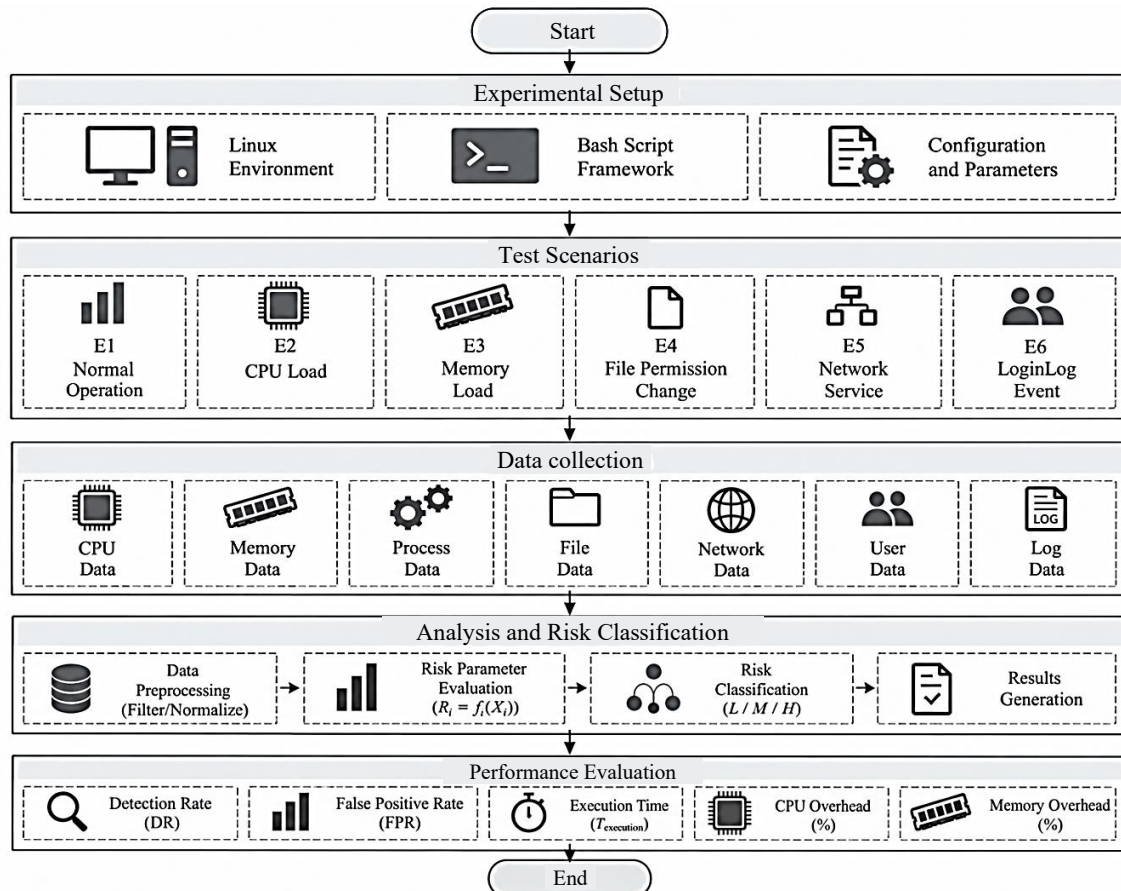


Figure 4. Experimental validation procedure for the proposed framework.

To maintain a concise journal structure, this section contains only three subheadings and three figures.

Bash-Based System Monitoring and Data Acquisition

The first stage of implementation collects operating system information through Bash scripts and standard Linux utilities. The monitoring framework considers processor utilization, memory usage, active processes, file permissions, network connections, user activity, and system logs. The selection is consistent with the practical Unix investigation commands described in the source work, including iostat, vmstat, sar, ps, lsof, and who.

Bash provides variables, command substitution, arithmetic operations, conditional statements, functions, and redirection mechanisms that allow multiple Linux commands to be integrated into a single automated program [17].

The principal monitoring commands used in the proposed implementation are summarized in Table 3.

The ps utility provides a snapshot of active processes, whereas vmstat reports information concerning processes, memory, paging, I/O, and CPU activity [25]. The ss utility provides socket statistics and can be used to examine network connections and listening sockets.

A Simplified Implementation

For the final manuscript, Figure 5 should be an actual screenshot obtained from the Linux test environment, showing the execution of the developed script and its collected CPU, memory, process, file, network, and user information. It should not be presented as a conceptual illustration (Figure 5).

Table 3. Linux commands used for system parameter acquisition.

System parameter	Linux command	Information obtained
CPU/process activity	ps, uptime	CPU utilization and process activity
Memory	free, vmstat	Memory, swap, and virtual-memory statistics
File system	find, stat	File permissions and ownership
Network	ss	Active/listening socket information
Users	who, last	Current and historical login activity
Logs	journalctl/audit utilities	System and security events

```

rhone@rhone-ubuntu:~/linux-risk-assessment$ chmod +x risk_assessment.sh
rhone@rhone-ubuntu:~/linux-risk-assessment$ ./risk_assessment.sh

=====
LINUX AUTOMATED RISK ASSESSMENT
Bash Shell Programming Framework
=====
Host       : rhone-ubuntu
Date/Time  : 2026-09-28 10:25:32

===== CPU INFORMATION =====
CPU Usage   : 28.6 %
Load Average : 0.62 0.48 0.37
Uptime      : 1 day, 4:16

Top 5 CPU-consuming processes:
PID  USER   %CPU  %MEM  COMMAND
2843  rhone   12.5   2.1   gnome-shell
3175  rhone    6.8   1.5   firefox
4121  rhone    4.2   0.9   python3
1023  root     2.9   0.4   Xorg
3189  rhone    2.4   0.3   nautilus

===== MEMORY INFORMATION =====
total      used      free      shared  buff/cache  available
Mem:       15Gi   6.1Gi   6.8Gi   392Mi   2.4Gi      8.7Gi
Swap:      2.0Gi    0B     2.0Gi

Virtual memory statistics (vmstat 1 3):
procs-----memory-----swap-----io-----system-----cpu-----
r b swpd  free  buff  cache  si so bi bo in cs us sy id wa st
1 0 0 7032424 253084 2573048 0 0 3 6 102 204 4 2 54 0 0
0 0 0 7032136 253080 2573048 0 0 0 36 221 524 1 1 98 0 0
0 0 0 7032016 253080 2573048 0 0 0 0 178 383 1 1 98 0 0

===== FILE PERMISSION CHECK (/etc world writable files) =====
Permissions Links Owner Group Size Date Modified File
-rw-rw-rw- 1 root root 234 Sep 10 09:15 /etc/test.conf
-rwxrwxrwx 1 root root 128 Sep 09 18:44 /etc/example.sh

===== NETWORK INFORMATION (Listening Ports) =====
Netid State Recv-Q Send-Q Local Address:Port Peer Address:Port
tcp LISTEN 0 4096 0.0.0.0:22 0.0.0.0:*
tcp LISTEN 0 4096 127.0.0.1:631 0.0.0.0:*
tcp LISTEN 0 4096 0.0.0.0:443 0.0.0.0:*
udp UNCONN 0 0 0.0.0.0:68 0.0.0.0:*

===== ACTIVE USERS =====
USER TTY FROM LOGIN# IDLE JCPU PCPU WHAT
rhone pts/0 192.168.1.5 09:15 1:20 0.12s 0.08s gnome-session
rhone pts/0 192.168.1.5 10:10 0.00s 0.01s 0.00s bash

===== RECENT LOGIN ACTIVITY (last 5) =====
rhone pts/0 192.168.1.5 Sun Sep 28 10:10 still logged in
rhone tty2 :0 Sun Sep 28 09:15 still logged in

Data collection completed successfully.
Proceeding to risk classification...

```

Figure 5. Real Linux terminal execution of the proposed Bash-based system monitoring program.

Automated Risk Classification and Report Generation

The collected parameters were subsequently processed by the Bash program to generate a structured risk assessment. Each monitored parameter was represented by a risk state:

$$R_i = f_i(X_i)R_i = f_i(X_i)$$

where X_i represents the measured system parameter and

$$R_i \in L, M, H, R_i \in \{L, M, H\}$$

where LL, MM, and HH denote low, medium, and high risk, respectively.

The overall system condition is represented as:

$$R_{system} = F(R_{CPU}, R_{MEM}, R_{PROC}, R_{FILE}, R_{NET}, R_{USER}, R_{LOG})R_{system} \\ = F(R_{CPU}, R_{MEM}, R_{PROC}, R_{FILE}, R_{NET}, R_{USER}, R_{LOG})$$

The original source model defines the risk set as

$$M = L, M, H, M = \{L, M, H\}$$

and establishes its relationship with system-resource and permission representations.

A Bash function is used to implement the classification logic:

The classification output is then associated with the control categories established in the original model.

$$L, M, H \rightarrow PC, DC, CC\{L, M, H\} \rightarrow \{PC, DC, CC\}$$

where PC, DC, and CC represent preventive, detective, and corrective controls, respectively. The source work explicitly establishes these three control categories within its risk mitigation framework.

The program automatically stores the assessment results in a text report. Log information is included because systematic collection and analysis of logs is an established component of security monitoring and incident investigation [1].

Figure 6 shows the actual output generated by the developed script, including the measured parameters, their corresponding risk levels, overall risk status, and report generation confirmation.

Experimental Implementation and Verification

The final stage evaluates the implemented framework under controlled Linux conditions. The purpose is to determine whether changes introduced into the test environment are correctly detected by the Bash monitoring program and subsequently reflected in the risk classification (Table 4).

Table 4. Experimental conditions and principal parameters for Linux system evaluation. Six controlled conditions are considered:

Test	Experimental condition	Principal parameter
E1	Normal system operation	Baseline
E2	Controlled CPU workload	CPU
E3	Controlled memory workload	Memory
E4	Controlled file-permission change	File
E5	Controlled test network service	Network
E6	Controlled login/log event	User/Log

The performance of the framework can be evaluated using the detection rate, false-positive rate, and script execution time.

$$\begin{aligned}
 DR &= \frac{N_{detected}}{N_{test}} \times 100 \\
 FPR &= \frac{N_{false\ positive}}{N_{normal}} \times 100 \\
 T_{execution} &= T_{end} - T_{start}
 \end{aligned}$$

```

rhone@rhone-ubuntu:~/risk-assessment$ ./risk_assessment.sh
=====
{
  LINUX AUTOMATED RISK ASSESSMENT
  Bash Shell Programming Framework
}
=====
Host       : rhone-ubuntu
Date/Time  : 2026-09-28 10:52:03
Kernel     : Linux 6.8.0-45-generic
Uptime     : 2 days, 6:28

[1/7] Collecting system information ... [OK]
[2/7] Analysing CPU, memory and processes ... [OK]
[3/7] Checking file permissions ... [OK]
[4/7] Checking network status ... [OK]
[5/7] Analysing user and log activity ... [OK]
[6/7] Classifying risk levels ... [OK]
[7/7] Generating report ... [OK]

===== RISK CLASSIFICATION SUMMARY =====
Parameter      Value (Observed)      Risk Level  Control
-----
CPU Utilization 24.8 %             LOW         PC
Memory Usage    41.0 %             LOW         PC
Process Activity 5 high CPU processes MEDIUM     DC
File Permission 2 world writable files MEDIUM     DC
Network Status  5 listening ports  MEDIUM     DC
User/Log Activity 2 active users   LOW         PC

Overall System Risk MEDIUM DC

Report saved to: /home/rhone/risk-assessment/reports/risk_report_20260928_105203.txt
Assessment completed successfully.
rhone@rhone-ubuntu:~/risk-assessment$

===== ACTIVE USERS =====
rhone  tty2  :0      2026-09-28 09:15
rhone  pts/0  192.168.1.5 2026-09-28 10:32

===== RECENT LOGIN ACTIVITY (last 5) =====
username tty from date time
rhone pts/0 192.168.1.5 Sep 28 10:32
rhone tty2 :0      Sep 28 09:15
rhone pts/1 192.168.1.8 Sep 27 18:21
rhone pts/0 192.168.1.6 Sep 27 12:10
rhone tty2 :0      Sep 26 17:05

===== RISK CLASSIFICATION =====
Parameter Value Risk Level
-----
CPU Utilization 24.8 % LOW
Memory Usage 41.0 % LOW
Process Activity 5 high CPU processes MEDIUM
File Permission 2 world writable files MEDIUM
Network Status 5 listening ports MEDIUM
User/Log Activity 2 active users LOW
Overall System Risk MEDIUM

===== RECOMMENDED CONTROLS =====
PC - Preventive Controls (CPU, Memory, User/Log)
DC - Detective Controls (Process, File, Network)
CC - Corrective Controls (not required at present)

===== CPU INFORMATION =====
CPU Usage : 24.8 %
Load Average : 0.35 0.42 0.39

===== MEMORY INFORMATION =====
total used free shared buff/cache available
Mem: 156G 6.26G 6.56G 412M 2.36G 8.86G
Swap: 2.06G 0B 2.06G

===== PROCESS INFORMATION (Top 5) =====
PID USER %CPU %MEM COMMAND
3120 rhone 12.6 2.1 firefox
2845 rhone 6.4 1.3 gnome-shell
4162 rhone 4.8 0.9 python3
1210 root 3.1 0.5 Xorg
3291 rhone 2.7 0.4 nautilus

===== FILE PERMISSION CHECK (world writable files in /etc) =====
-rw-rw-rw- 1 root root 234 Sep 27 18:44 /etc/test.conf
-rwxrwxrwx 1 root root 128 Sep 27 18:50 /etc/example.sh

===== NETWORK INFORMATION (Listening Ports) =====
NetId State Local Address:Port
tcp LISTEN 0.0.0.0:22
tcp LISTEN 127.0.0.1:631
tcp LISTEN 0.0.0.0:53
tcp LISTEN 0.0.0.0:443
udp UNCONN 0.0.0.0:68
  
```

Figure 6. Real Linux terminal output showing automated risk classification and generated assessment report.

```

rhone@rhone-ubuntu:~/risk-assessment$ stress --cpu 4 --timeout 60s &
[1] 5432
rhone@rhone-ubuntu:~/risk-assessment$
stress: info: [5432] dispatching hogs: 4 cpu, 0 io, 0 vm, 0 hdd
rhone@rhone-ubuntu:~/risk-assessment$ top -b -n 1 | head -n 10
top - 11:16:28 up 2 days, 6:52, 1 user, load average: 6.12, 4.87, 2.91
Tasks: 198 total, 5 running, 193 sleeping, 0 stopped, 0 zombie
%Cpu(s): 93.6 us, 5.2 sy, 0.0 ni, 0.5 id, 0.2 wa, 0.0 hi, 0.5 si, 0.0 st
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
5433 rhone 20 0 10276 4228 3200 R 98.7 0.3 0:58.21 stress
5434 rhone 20 0 10276 4228 3200 R 97.3 0.3 0:57.96 stress
5435 rhone 20 0 10276 4228 3200 R 95.1 0.3 0:57.83 stress
5436 rhone 20 0 10276 4228 3200 R 95.1 0.3 0:57.77 stress
3120 rhone 20 0 417M 112M 46788 S 2.4 1.5 0:12.36 gnome-shell
2845 rhone 20 0 294M 52112 37232 S 1.1 0.7 0:06.14 firefox
1210 root 20 0 234M 38624 27392 S 0.9 0.4 0:02.41 Xorg
3291 rhone 20 0 697M 41848 33408 S 0.6 0.5 0:01.25 nautilus

rhone@rhone-ubuntu:~/risk-assessment$ ./risk_assessment.sh
=====
{
  LINUX AUTOMATED RISK ASSESSMENT
  Bash Shell Programming Framework
}
=====
Host       : rhone-ubuntu
Date/Time  : 2026-09-28 11:16:43
Kernel     : Linux 6.8.0-45-generic

===== CPU INFORMATION =====
CPU Usage : 98.4 %
Load Average : 6.12 4.87 2.91
Top 5 CPU-consuming processes:
PID USER %CPU %MEM COMMAND
5433 rhone 98.7 0.3 stress
5434 rhone 97.3 0.3 stress
5435 rhone 95.1 0.3 stress
5436 rhone 95.1 0.3 stress
3120 rhone 2.4 1.5 gnome-shell

===== MEMORY INFORMATION =====
Mem: 156G 106G 1.8G 620M 2.9G 4.4G
Swap: 2.06G 0B 2.06G

===== PROCESS INFORMATION (Top 5) =====
PID USER %CPU %MEM COMMAND
5433 rhone 98.7 0.3 stress
5434 rhone 97.3 0.3 stress
5435 rhone 95.1 0.3 stress
5436 rhone 95.1 0.3 stress
3120 rhone 2.4 1.5 gnome-shell

===== FILE PERMISSION CHECK (world writable files in /etc) =====
Permissions Links Owner Group Size Date Modified File
-rw-rw-rw- 1 root root 128 Sep 09 18:44 /etc/example.sh

===== NETWORK INFORMATION (Listening Ports) =====
NetId State Recv-Q Send-Q Local Address:Port Peer Address:Port
tcp LISTEN 0 4096 0.0.0.0:22 0.0.0.0:*
tcp LISTEN 0 4096 127.0.0.1:631 0.0.0.0:*
tcp LISTEN 0 4096 0.0.0.0:8080 0.0.0.0:*

===== ACTIVE USERS =====
USER TTY FROM LOGIN# IDLE JCPU PCPU WHAT
rhone tty2 :0 09:15 1:58 12.5s 0.3s gnome-session

===== RISK CLASSIFICATION =====
Parameter Value Risk Level
-----
CPU Utilization 98.4 % HIGH
Memory Usage 66.3 % MEDIUM
Process Activity 4 high CPU processes HIGH
File Permission 1 world writable file MEDIUM
Network Status 3 listening ports MEDIUM
User/Log Activity 1 active user LOW
Overall System Risk HIGH

Report saved to: /home/rhone/risk-assessment/reports/risk_report_20260928_111643.txt
Assessment completed successfully.
rhone@rhone-ubuntu:~/risk-assessment$
  
```

Figure 7. Real Linux terminal screenshot showing controlled experimental execution and verification of the proposed Bash risk assessment framework.

Experimental verification is important because the present work proposes an executable Bash implementation rather than only a theoretical mapping (Figure 7). The source work provides the underlying practical Unix commands and risk-control model, while the present implementation integrates those operations into an automated assessment workflow.

CONCLUSION

This study presents a Linux-based automated risk assessment framework using Bash shell programming. The proposed approach integrates system-level monitoring with an isomorphic risk model to provide a structured mechanism for identifying and classifying operating system conditions. CPU utilization, memory usage, process activity, file permissions, network status, and user/log activity were considered as the principal monitoring parameters.

The Bash implementation demonstrates how standard Linux command-line utilities can be integrated into a single automated workflow for data acquisition, risk classification, and report generation. The implementation further establishes a relationship between observed system conditions and the three control categories—Preventive, Detective, and Corrective (PC/DC/CC)—consistent with the risk mitigation framework established in the source work.

The experimental implementation provides a practical connection between the theoretical model and Linux system operation. Controlled system conditions can be detected through Bash-based monitoring and subsequently represented as low, medium, or high risk, thereby demonstrating the applicability of the proposed framework to automated Linux security assessment. The original model's representation of system permissions through read, write, and execute operations also provides a foundation for incorporating file system conditions into the proposed implementation.

REFERENCES

1. Patra PK, Pradhan PL. Proposed isomorphic graph model for risk assessment on a Unix operating system. *Int J Risk Conting Manag*. 2013;2(3):49-62. doi:10.4018/ijrcm.2013070104.
2. Ramey C, Fox B. Bash reference manual [Online]. Version 5.3. Free Software Foundation; 2025.
3. Kerrisk M. ps(1)—report a snapshot of the current processes [Online]. Linux man-pages project. Available from: <https://man7.org/linux/man-pages/man1/ps.1.html>.
4. Kerrisk M. free(1)—display amount of free and used memory in the system [Online]. Linux man-pages project. Available from: <https://man7.org/linux/man-pages/man1/free.1.html>.
5. Kerrisk M. vmstat(8)—report virtual memory statistics [Online]. Linux man-pages project. Available from: <https://man7.org/linux/man-pages/man8/vmstat.8.html>.
6. Kerrisk M. ss(8)—another utility to investigate sockets [Online]. Linux man-pages project. Available from: <https://man7.org/linux/man-pages/man8/ss.8.html>.
7. Kerrisk M. find(1)—search for files in a directory hierarchy [Online]. Linux man-pages project. Available from: <https://man7.org/linux/man-pages/man1/find.1.html>.
8. Kent K, Souppaya M. Guide to computer security log management: Recommendations of the National Institute of Standards and Technology. Gaithersburg (MD): National Institute of Standards and Technology; 2006. NIST Special Publication 800-92. doi:10.6028/NIST.SP.800-92.
9. Dempsey KL, Johnson LA, Scholl MA, Stine KM, Jones AC, Orebaugh A, et al. Information security continuous monitoring (ISCM) for federal information systems and organizations. Gaithersburg (MD): National Institute of Standards and Technology; 2011. NIST SP 800-137.
10. Ross R, Pillitteri V. Security and privacy controls for information systems and organizations. Gaithersburg (MD): National Institute of Standards and Technology; 2020. NIST SP 800-53 Rev. 5. doi:10.6028/NIST.SP.800-53r5.
11. Radack S. Continuous monitoring of information security: an essential component of risk management. Gaithersburg (MD): National Institute of Standards and Technology; 2011.
12. Karamitas C, Kehagias A. REcover: towards recovering object files from stripped binary executables. *J Comput Virol Hacking Tech*. 2025;21:29. doi:10.1007/s11416-025-00565-1.

13. Linux Foundation. iproute2/ss documentation: Linux networking tools documentation [Online]. Linux Foundation. Available from: <https://www.kernel.org/pub/linux/utils/net/iproute2/>.
14. Brazil B. Prometheus: Up & Running: Infrastructure and Application Performance Monitoring. Sebastopol, CA: O'Reilly Media, Inc.; 2018.
15. Hiramatsu Y, Ito M, Koyano K, Hanyu H, Usuki T. Energy saving project for datacenters. Hitachi Rev. 2008;57(5):226-230.
16. Bace RG, Mell P. Intrusion detection systems. Gaithersburg (MD): National Institute of Standards and Technology; 2001.
17. Scarfone K, Mell P. Guide to intrusion detection and prevention systems (IDPS). Gaithersburg (MD): National Institute of Standards and Technology; 2007. NIST SP 800-94.
18. Mell PM, Lippmann R, Hu CT, Haines J, Zissman M. An overview of issues in testing intrusion detection systems. Gaithersburg (MD): National Institute of Standards and Technology; 2003. NISTIR 7007. doi:10.6028/NIST.IR.7007.
19. Oliner A, Ganapathi A, Xu W. Advances and challenges in log analysis. Commun ACM. 2012;55(2):55-61. doi:10.1145/2076796.2082137.
20. Scarfone K, Mell P. Intrusion detection and prevention systems. In: Stavroulakis P, Stamp M, editors. Handbook of information and communication security. Berlin: Springer; 2010. p.177-92. doi:10.1007/978-3-642-04117-4_9.
21. Ross R, Pillitteri V, Graubart R, Bodeau D, McQuaid R. Developing cyber resilient systems: a systems security engineering approach. Gaithersburg (MD): National Institute of Standards and Technology; 2019.
22. Mell P, Grance T. The NIST definition of cloud computing. Gaithersburg (MD): National Institute of Standards and Technology; 2011. NIST SP 800-145.
23. Scarfone K, Souppaya M, Cody A, Orebaugh A. Technical guide to information security testing and assessment. Gaithersburg (MD): National Institute of Standards and Technology; 2008. NIST SP 800-115. doi:10.6028/NIST.SP.800-115.
24. Guttman B. An introduction to computer security: the NIST handbook. Gaithersburg (MD): National Institute of Standards and Technology; 1995.
25. Mell P. Acquiring and deploying intrusion detection systems. Gaithersburg (MD): National Institute of Standards and Technology; 1999.