Keylogger Detection and Termination System

A Project Report Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Bachelor of Technology

(Computer Science & Engineering)

Submitted By

Bittu Kumar

Reg. No. : 222025109271

Under Guidance of

Mrs. Shibya Swaroop

Faculty of Computing and Information Technology

Usha Martin University, Ranchi

November 18, 2025

BONAFIDE CERTIFCATE

Certified that this project report titled **“Keylogger Detection and Termination System”** is the bonafide work of Mr. Bittu Kumar, who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

(Signature of the concerned Supervisor of the Organization with Organization Seal)

(Certificate to be countersigned by the HOD.)

DECLARATION

I do hereby declare that this report entitled **"Keylogger Detection and Termination System"**, submitted by Bittu Kumar, bearing Reg. No. : 222025109271 in the fulfillment of the requirement for the degree of Bachelor of Technology in Computer Science & Engineering to Usha Martin University, Ranchi, is my own and it is not submitted to any other institute.

Bittu Kumar

Reg. No. : 222025109271

CERTIFICATE

This is to certify that entitled **“Keylogger Detection and Termination System”** being submitted by Bittu Kumar, bearing Reg. No. : 222025109271, in the fulfillment of the requirement for the degree of Bachelor of Technology in Computer Science & Engineering to Usha Martin University, Ranchi, is a bonafide work carried out under my supervision. The matter embodied in this report is original and has not been submitted for the award of any other degree.

Dr. Naghma Khatoon	**Mrs. Shibya Swaroop**
HOD	**Guide**

ACKNOWLEDGEMENTS

I express my deepest sense of gratitude to my guide **Mrs. Shibya Swaroop** Faculty of Computing and Information Technology, Usha Martin University, Ranchi, for suggesting the subject of work and constant supervision throughout this work. Her co-operation and timely suggestions have been unparalleled stimuli for me to travel eventually towards the completion of this project report. Indeed her continuous involvement has helped me in bringing of this project work which otherwise would have remained a distant dream.

I am indeed thankful to **Dr. Naghma Khatoon**, HOD, Faculty of Computing and Information Technology, Usha Martin University, Ranchi, for giving me permission to carry out my project work. I would like to express my gratitude to all teaching and non-teaching staff members of Faculty of Computing and Information Technology, Usha Martin University, Ranchi, for their co-operation in my work.

Bittu Kumar

Reg. No. : 222025109271

Abstract

The increasing reliance on digital systems for personal, professional, and financial activities has amplified the risk of sophisticated cyber threats, particularly keyloggers malicious programs designed to covertly capture user inputs such as passwords and confidential data. Traditional antivirus solutions, which depend heavily on signature-based detection, often fail to identify modern keyloggers that employ advanced evasion techniques such as rootkits, process injection, and polymorphism. This project, titled "Keylogger Detection and Termination System," introduces a proactive, behavior-based security mechanism capable of detecting and neutralizing both known and unknown keylogger variants in real time. The proposed system integrates multiple detection layers, including decoy input injection, behavioral analysis, process monitoring, and real-time input tracking. Decoy keystrokes are periodically injected at the driver level to expose unauthorized data interception attempts, while behavioral models continuously analyze typing and mouse activity patterns to identify anomalies indicative of malicious behavior. The detection engine fuses results from these modules using intelligent decision algorithms, minimizing false positives while enhancing detection accuracy. Upon confirmation of a threat, the system's response module automatically terminates the malicious process and alerts the user through an intuitive dashboard interface. This research bridges critical gaps in conventional cybersecurity methods by emphasizing behavior-driven analysis over static signatures. It demonstrates the feasibility of lightweight, efficient, and adaptive keylogger detection mechanisms capable of defending against zero-day and polymorphic threats. Future enhancements may include machine learning integration for adaptive threat prediction, cloud-based deployment for enterprise scalability, and cross-platform support extending protection to mobile and IoT devices. Overall, the project contributes to the advancement of proactive cybersecurity frameworks, ensuring safer and more resilient computing environments.

Keywords – Cybersecurity, Injection, IoT, Keylogger, Malicious, Monitoring, Protection, Signature, Termination, Threat

TABLE OF CONTENTS

List of Figures

List of Tables

List of Abbreviations

Abbreviations	Full Form
IoT	Internet of Things
GUI	Graphical User Interface
API	Application Programming Interface
PID	Process ID
eBPF	Extended Berkeley Packet Filter
TSC	Threat Scoring Component
BDA	Behavioral Detection Analysis
DMA	Device Monitoring Agent
ARM	Automated Response Mechanism
IOC	Indicator of Compromise
C2	Command and Control
HPA	Heuristic Pattern Analysis
IDS	Intrusion Detection System
IPS	Intrusion Prevention System

Chapter 1

Introduction

1.1 Overview

The rise of digital transformation has brought immense convenience to users and organizations alike. However, it has also exposed systems to sophisticated forms of cyberattacks that compromise privacy, security, and data integrity. Among these, keyloggers represent one of the most covert and damaging threats. A keylogger is a type of spyware designed to monitor and record every keystroke entered by a user, often without their knowledge. This stolen information can be exploited for malicious purposes such as financial fraud, credential theft, identity impersonation, or unauthorized system access. The silent nature of keyloggers makes them extremely dangerous, as they often operate undetected for long periods, exfiltrating confidential data to remote attackers. Traditional antivirus software primarily depends on signature-based detection, where malicious programs are identified based on pre-known code patterns or digital fingerprints. While effective against known malware, this approach fails to detect zero-day, polymorphic, and kernel-level keyloggers that disguise themselves using obfuscation and code mutation techniques. As a result, there is a critical need for intelligent and adaptive security mechanisms that can identify keyloggers based on their behavior rather than their signatures. The proposed project, “Keylogger Detection and Termination System,” introduces a proactive defense framework that combines multiple security layers decoy input injection, behavioral analysis, process monitoring, and real-time response. The system continuously monitors keyboard and mouse interactions, detects abnormal behaviors, and injects invisible decoy keystrokes to lure malicious software into revealing its presence. If unauthorized processes attempt to capture or transmit these decoy inputs, the system instantly identifies and terminates them. A user-friendly dashboard provides visual insights, threat alerts, and system health updates for transparent operation. This approach ensures not only real-time protection against existing threats but also adaptability to emerging ones. By integrating behavioral intelligence, decoy-based detection, and automated response, the project bridges the gap left by conventional antivirus tools, providing a robust solution suitable for individuals, enterprises, and organizations handling sensitive information. Beyond detection, the system emphasizes autonomous defense and user empowerment. It enables decision-making without manual intervention, allowing immediate neutralization of active threats while maintaining system performance.

1.2 Background

Over the past decade, cyber threats have evolved from simple viruses to highly sophisticated, multi-layered attacks capable of evading even advanced defense mechanisms. Keyloggers, once used primarily for legitimate monitoring, have now become a preferred tool among cybercriminals due to their simplicity, efficiency, and stealth. They can be implemented at various levels of the operating system application, kernel, or firmware making detection extremely challenging. Modern variants employ rootkit techniques, API hooking, and process injection to conceal their presence, while some even disable security defenses or operate in encrypted channels. Conventional security solutions struggle to cope with such sophistication. Most rely on static databases of malware signatures, which are ineffective against unknown or dynamically morphing threats. As keyloggers continue to adopt advanced evasion methods, there is a growing need for dynamic and behavioral detection techniques. Behavioral-based systems analyze how programs interact with system resources such as monitoring abnormal access to keyboard APIs, file writes, or network transmissions to infer malicious intent even without a known signature. Existing research has explored methods like keystroke dynamics, heuristic analysis, and machine learning to detect anomalies in user behavior. However, these systems often face challenges such as high false positives, resource inefficiency, and limited scope of detection. Furthermore, few studies have explored decoy-based mechanisms a proactive approach where fake keystrokes are injected to trap keyloggers. This technique ensures that any unauthorized capture of these inputs directly signals malicious intent, thereby enhancing detection accuracy with minimal computational overhead. The increasing frequency of online transactions, digital communication, and remote work environments makes the development of such detection mechanisms vital. Protecting sensitive data requires systems that are both intelligent and autonomous—capable of learning, detecting, and responding in real time. The Keylogger Detection and Termination System builds upon these principles, addressing current research gaps and providing a scalable, efficient, and proactive defense framework against one of the most persistent cyber threats in modern computing. Existing research has explored methods like keystroke dynamics, heuristic analysis, and machine learning to detect anomalies in user behavior. However, these systems often face challenges such as high false positives, resource inefficiency, and limited scope of detection. Furthermore, few studies have explored decoy-based mechanisms a proactive approach where fake keystrokes are injected to trap keyloggers. This technique ensures that any unauthorized capture of these inputs directly signals malicious intent, thereby enhancing detection accuracy with minimal computational overhead.

1.3 Applications & Features of Project

The Keylogger Detection and Termination System is designed to deliver comprehensive, real-time protection against keylogging threats through a combination of proactive and adaptive security mechanisms. The project's practical applications span across multiple domains, including personal computing, financial institutions, enterprise environments, and healthcare systems. The following key applications and features illustrate the system's versatility, efficiency, and contribution to modern digital security frameworks.

1. Real-Time Threat Detection and Termination : One of the primary applications of the Keylogger Detection and Termination System lies in its ability to provide real-time identification and neutralization of keylogger threats. The system continuously monitors low-level keyboard and mouse activity, allowing it to instantly detect any unauthorized attempt to record or intercept user inputs. Once a malicious process is identified, the response module immediately terminates the process and blocks further system access. This ensures minimal exposure time, preventing sensitive data leakage such as passwords or banking credentials.

2. Enhanced Security for Financial and Enterprise Environments : Financial institutions, corporate offices, and organizations dealing with confidential information can deploy this system to strengthen endpoint protection against keylogging attacks. In such environments, keyloggers pose significant risks unauthorized access to login credentials, transaction manipulation, or client data breaches. The proposed system proactively monitors all running processes and detects hidden logging activities, ensuring the integrity of financial transactions and safeguarding enterprise-level data. This makes it a valuable cybersecurity layer for banks, government departments, and businesses that handle high-value or sensitive information daily.

3. Behavioral Analysis and Adaptive Detection : A key feature of the project is its behavioral analysis engine, which intelligently distinguishes between normal user behavior and suspicious system activity. Instead of depending solely on static malware signatures, the system learns to recognize abnormal keystroke patterns, unusual API access, and idle-time system behavior that typically indicate a keylogger's presence. This adaptive capability enables it to detect zero-day and polymorphic keyloggers malware that evolves dynamically to bypass traditional antivirus programs.

4. Decoy Input Injection for Proactive Defense : Unlike most existing security tools that wait for malicious activity to occur, this system implements a proactive decoy-based detection mechanism. It periodically injects invisible fake keystrokes (decoys) at the driver level and monitors whether these are captured or logged by any process. Since legitimate applications ignore these inputs, any attempt to intercept them is a strong indicator of malicious intent. This technique serves as an intelligent trap mechanism, allowing the system to expose even the most stealthy keyloggers. The decoy-based approach significantly improves detection precision while maintaining low system overhead, making it efficient for both personal and enterprise usage.

5. User-Friendly Dashboard and Forensic Analysis : To ensure transparency and accessibility, the system features an intuitive graphical dashboard that displays real-time system status, detected threats, and activity summaries. The interface is designed for both technical and non-technical users, allowing them to monitor security health, receive instant alerts, and review detailed logs of past incidents. The forensic logging component records every suspicious event, process interaction, and detection instance, creating a comprehensive audit trail. This data can later be used for post-incident analysis, compliance reporting, or academic research, turning the system into both a preventive and investigative cybersecurity tool.

6. Cross-Domain and Scalable Deployment : The Keylogger Detection and Termination System is designed with scalability and cross-platform adaptability in mind. It can be deployed across a variety of environments, including personal computers, enterprise networks, healthcare systems, and educational institutions. In healthcare, it helps safeguard patient records and comply with data privacy regulations, while in education, it ensures secure online examination systems. Its modular design allows easy integration with existing security infrastructures, cloud-based monitoring tools, and centralized administrative systems. This broad applicability ensures that the project contributes meaningfully to the growing demand for lightweight, adaptive, and intelligent endpoint security solutions in the modern digital ecosystem.

7. Lightweight Performance and Resource Efficiency : A distinguishing feature of the Keylogger Detection and Termination System is its lightweight architecture, which ensures minimal impact on system performance while maintaining continuous protection. Unlike many traditional security solutions that consume significant CPU and memory resources, this system is carefully optimized to run background processes efficiently using event-driven monitoring and selective scanning. By leveraging low-level system hooks and asynchronous detection logic, it operates seamlessly without interrupting normal user activity or slowing down applications. This balance between high security and low resource usage makes the system particularly suitable for laptops, workstations, and enterprise setups where performance is critical.

1.4 Objective

- To develop a behavioral and heuristic-based framework for detecting keyloggers through suspicious system activity rather than code signatures.
- To implement a decoy keystroke injection mechanism that proactively traps and identifies malicious keylogging processes.
- To minimize false positives and enhance detection accuracy using a decision fusion model that integrates multiple analysis modules.
- To design an automated response system that immediately neutralizes detected threats without requiring user intervention.
- To develop an intuitive dashboard that provides real-time monitoring, threat insights, and security management capabilities for users.

1.5 Problem Statement

In the current era of digital dependency, cybersecurity threats have become more sophisticated, targeted, and difficult to detect. Among these threats, keyloggers stand out as one of the most dangerous and stealthy types of malware. A keylogger silently records every keystroke entered by a user, including sensitive data such as usernames, passwords, banking credentials, and personal communications. The captured data is often transmitted to attackers, leading to severe consequences such as financial theft, identity fraud, unauthorized system access, and data breaches. The silent and persistent nature of keyloggers makes them exceptionally challenging to identify, often allowing them to operate undetected for extended periods. Traditional antivirus and anti-malware solutions predominantly rely on signature-based detection, where a program is flagged as malicious only if it matches a known signature in the database. While effective against previously identified threats, this method fails to detect zero-day keyloggers, polymorphic malware, and rootkit-level intrusions that continuously mutate their code or operate at the kernel level. Furthermore, advanced keyloggers often employ process injection, API hooking, and anti-debugging techniques to hide their presence from security scanners, making conventional protection mechanisms ineffective. This leads to a major security gap, as users remain unaware of the ongoing compromise of their data. Additionally, legitimate monitoring tools and remote access software can easily be repurposed by attackers for malicious use, further complicating the detection process. Since such software often mimics normal system behavior, distinguishing between authorized and unauthorized activity becomes difficult. As a result, many existing detection systems generate high false positives or require manual supervision, reducing their practicality for real-world deployment. Another critical issue lies in the lack of behavioral and proactive defense mechanisms in most traditional security systems. Current approaches fail to analyze user activity patterns, idle-time behavior, or input anomalies that could indicate malicious interference. Without real-time behavioral monitoring, such systems remain reactive, detecting threats only after data has already been compromised. This reactive model is insufficient in a rapidly evolving cyber landscape where prevention and early detection are paramount. Furthermore, resource-heavy detection algorithms often cause performance degradation, discouraging users from keeping them active continuously. In enterprise or financial environments where performance and uptime are crucial, such inefficiencies make conventional tools unsuitable for large-scale deployment. Hence, there is an urgent need for a proactive, intelligent, and lightweight keylogger detection system that can identify malicious behavior in real time irrespective of known signatures and respond immediately to prevent data loss. The proposed Keylogger Detection and Termination System addresses this gap by integrating behavioral analysis, decoy input injection, process monitoring, and automated response mechanisms, offering a holistic and adaptive approach to modern keylogger detection.

1.6 Research Gaps

- Existing keylogger detection relies too heavily on signatures, making it ineffective against zero-day and polymorphic threats, highlighting the need for behavior-driven detection.

- Current behavioral detection systems lack deep data correlation, resulting in poor behavioral profiling and high false positives.

- Decoy input injection as a proactive keylogger detection method remains largely unexplored in existing research.

- Most studies focus on detection and ignore automated threat neutralization, leaving systems vulnerable after alerts.

- Few solutions offer comprehensive logging and post-incident analytics, limiting forensic investigation and audit capabilities.

Chapter 2

Literature Review

The growing sophistication of cyber threats has driven researchers to explore advanced methods for detecting and mitigating malicious software, especially keyloggers, which severely threaten data privacy and system security. Traditional signature-based detection has proven inadequate against modern, adaptive malware that evades static analysis. Recent studies have therefore focused on behavioral analysis, heuristic modeling, and decoy-based detection to enhance keylogger identification and prevention. This review summarizes key research contributions forming the basis of the proposed system and outlines the gaps it aims to address.

1. **D. Elelegwu, L. Chen, Y. Ji and J. Kim, "A Novel Approach to Detecting and Mitigating Keyloggers," SoutheastCon 2024, Atlanta, GA, USA, 2024, pp. 1583-1590, doi: 10.1109/SoutheastCon52093.2024.10500122** : This research introduces a new browser extension designed to combat the growing threat of keylogger spyware. The extension uses a sophisticated algorithm to quickly identify and block malicious activity – specifically keylogging – which monitors keystrokes and steals sensitive data like passwords. Users are given the option to immediately terminate suspicious processes or validate their authenticity, providing crucial real-time protection. The extension is designed to be flexible and adaptable to different platforms and devices, demonstrating significant effectiveness in strengthening online security.

2. **P. V, "Beyond Traditional Keyloggers: Developing and Detecting Advanced Keystroke Monitoring Systems," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-6, doi: 10.1109/CSITSS60515.2023.10334216** : This paper analyzes keyloggers – sophisticated software that steal keystrokes – in detail. It examines how these programs operate, focusing on their ability to intercept and record user activity, including keystrokes. The research highlights the use of System Call Monitoring as a key defensive strategy and discusses its effectiveness. Ultimately, the paper aims to provide a clear understanding of keyloggers and their impact, highlighting the gap between malicious activity and current security measures.

3. **A. Ankit, S. Inder, A. Sharma, R. Johari and D. P. Vidyarthi, "Simulating Cyber Attacks and Designing Malware using Python," 2023 10th International Conference on**

Signal Processing and Integrated Networks (SPIN), Noida, India, 2023, pp. 473-478, doi: 10.1109/SPIN57001.2023.10116554 : This research explores cybersecurity threats by simulating cyberattacks to understand vulnerabilities and develop solutions. The authors created applications and malware designed to mimic real-world attacks, focusing on stages like entry, distribution, exploitation, and infection. The goal is to identify weaknesses and vulnerabilities in systems to prevent attacks.

4. **J. Sabu, A. S, A. Gopan, G. S and S. Murali, "Advanced Keylogger with Keystroke Dynamics," 2023 International Conference on Inventive Computation Technologies (ICICT), Lalitpur, Nepal, 2023, pp. 1598-1603, doi: 10.1109/ICICT57646.2023.10134044** : Keyloggers are software programs that record keystrokes, allowing hackers to steal sensitive data like passwords and financial details. These programs use keystroke dynamics – analyzing the patterns of keystrokes – to create a more secure form of authentication. The software also gathers information about the user's system and applications, including IP addresses, MAC addresses, and user data, to identify potential threats. It logs all keystrokes, encrypts the data, and transmits it to a remote server for analysis, offering a more comprehensive security measure.

5. **N. T. Singh, A. Shukla, A. Nagar, K. Arya, A. Tiwari and Y. Varun, "Keylogger Development: Technical Aspects, Ethical Considerations, and Mitigation Strategies," 2023 International Conference on Energy, Materials and Communication Engineering (ICEMCE), Madurai, India, 2023, pp. 1-5, doi: 10.1109/ICEMCE57940.2023.10434134** : This study examines keylogger technology – its development, ethical considerations, and how to combat them. It explores the technical aspects of keyloggers, including their methods of recording keystrokes, contrasts malicious use with legitimate applications, and highlights the importance of education. The paper advocates for responsible digital tool application and emphasizes understanding keyloggers as crucial for protecting privacy and data security.

6. **S. Ortolani, C. Giuffrida and B. Crispo, "Unprivileged Black-Box Detection of User-Space Keyloggers," in IEEE Transactions on Dependable and Secure Computing, vol. 10, no. 1, pp. 40-52, Jan.-Feb. 2013, doi: 10.1109/TDSC.2012.76** : Software keyloggers are rapidly growing because they can run in unprivileged user space while still capturing all keystrokes. This makes them easy to deploy but also easier to analyze. The authors propose a

detection method that sends controlled keystroke sequences into the system and observes which processes react, allowing clear identification of keyloggers. The technique works entirely in unprivileged mode, just like the keyloggers it targets. Tests on popular free keyloggers show strong practical effectiveness. The authors also explore possible evasion strategies and introduce heuristics to strengthen detection, achieving high accuracy with minimal false positives and negatives.

7. **A. Moser, C. Kruegel and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," 2007 IEEE Symposium on Security and Privacy (SP '07), Berkeley, CA, USA, 2007, pp. 231-245, doi: 10.1109/SP.2007.17** : Malware usually performs harmful actions, but many of these behaviors may only activate under certain conditions, making them difficult to detect through traditional single-path analysis. The authors propose a system that automatically explores multiple execution paths of a program, triggering hidden malicious actions that depend on specific inputs or environmental conditions. This approach generates a more complete behavioral profile of malware and helps identify when and why suspicious actions occur. Experiments show that many malware samples behave differently under varying inputs, proving that multi-path exploration significantly improves malware behavior extraction.

8. **K. A. Rahman, K. S. Balagani and V. V. Phoha, "Snoop-Forge-Replay Attacks on Continuous Verification With Keystrokes," in IEEE Transactions on Information Forensics and Security, vol. 8, no. 3, pp. 528-541, March 2013, doi: 10.1109/TIFS.2013.2244091** : The authors introduce the snoop-forge-replay attack — a sample-level forgery that uses readily available keyloggers and keystroke-synthesis APIs to forge keystroke samples and bypass continuous keystroke-based verification. Through 2,640 experiments they show the attack yields alarmingly higher error rates than standard zero-effort impostor baselines, works against multiple state-of-the-art verification methods, different latency types, and various matching settings, and remains effective with as few as 20–100 snooped keystrokes. They also demonstrate that virtualization can be used to rapidly generate large numbers of forgeries, highlighting practical scalability and serious security weaknesses in current continuous keystroke verification systems.

9. **D. Javaheri, M. Hosseinzadeh and A. M. Rahmani, "Detection and Elimination of Spyware and Ransomware by Intercepting Kernel-Level System Routines," in IEEE Access, vol. 6, pp. 78321–78332, 2018, doi: 10.1109/ACCESS.2018.2884964** : Spyware and

ransomware are increasingly complex, stealthy, and long-lasting threats. The authors propose a dynamic behavioral analysis system that deeply hooks kernel-level routines to detect and track hidden spyware components such as keyloggers, screen recorders, and blockers. Using machine-learning classifiers such as linear regression, JRIP, and J48 decision trees, the system identifies different malware classes with strong accuracy. The architecture also force-terminates malicious processes, removes infected files, and restricts network communication. Experiments show around 93% detection accuracy and 82% successful disinfection, demonstrating that kernel-level interception is highly effective for uncovering and eliminating stealthy spyware.

10. **H. M. Salih and M. S. Mohammed, "Spyware Injection in Android using Fake Application," 2020 International Conference on Computer Science and Software Engineering (CSASE), Duhok, Iraq, 2020, pp. 100-105, doi: 10.1109/CSASE48920.2020.9142101** : The authors analyze Android spyware concealed inside a fake application and demonstrate a complete spyware system that collects sensitive user data (contacts, messages, calls, accounts, location via Wi-Fi/SIM/3G/4G/LTE) and can send deceptive alerts to the victim. The work shows how social-engineering (installing a fake app) and platform APIs can be abused to harvest extensive personal information, highlighting a practical threat vector for mobile privacy and security.

11. **M. Shafi, R. K. Jha and S. Jain, "Behavioral Model for Live Detection of Apps Based Attack," in IEEE Transactions on Computational Social Systems, vol. 10, no. 3, pp. 934–946, June 2023, doi: 10.1109/TCSS.2022.3166145** : The authors propose a behavioral-model-based detection system for identifying application-based attacks on smartphones. They show that malicious apps can run in the background with hidden visibility, accessing sensitive data and launching attacks such as spyware, phishing, and privacy leakage. To counter this, they introduce the Application-Based Behavioral Model Analysis (ABMA) scheme, which detects suspicious app activity by analyzing power usage, battery drain, and data consumption. The system tests behavior under Wi-Fi, mobile data, and mixed connectivity to improve reliability. Simulation results demonstrate that ABMA can effectively detect abnormal behavior linked to hidden malicious apps.

12. **K. Onarlioglu, W. Robertson and E. Kirda, "Overhaul: Input-Driven Access Control for Better Privacy on Traditional Operating Systems," 2016 46th Annual IEEE/IFIP**

International Conference on Dependable Systems and Networks (DSN), Toulouse, France, 2016, pp. 443–454, doi: 10.1109/DSN.2016.47 : The authors introduce a new input-driven access control architecture that brings mobile-style, user-driven permission controls to traditional desktop operating systems. Instead of relying on predeclared permissions or rewritten applications, their system dynamically decides whether an app can access sensitive resources—like the camera, microphone, clipboard, or screen—based on how close the request is to recent user interaction. Access requests are communicated to the user through visual alerts, and the system is fully transparent to existing applications without modifying them. Their prototype demonstrates that this model improves privacy with no noticeable performance cost.

2.1 Table of Literature Review

Table 2.1 – Table of Literature Review

S.No.	Paper Title	Author(s)	Year	Methodology Used	Pros	Cons	Journal Name	Page No.	Volume No.	DOI
1	A Novel Approache to Detecting and Mitigating Keyloggers	Damilola Elelegwu, Lei Chen, Yimin g Ji, Jongyeop Kim	2024	It Used the dendritic cell algorithm to identify suspicious keylogging behavior in the system	Detects unknown keyloggers; adaptive nature	Higher false positives; computationally heavy	2024 Int. Conf. on Communications and Mobile Computing	8–132	17	10.1109/ SoutheastCo n52093.2024 .10500122
2	Beyond Traditional Keyloggers: Developing and Detecting Advanced Keystroke Monitoring Systems	P. V	2023	It Designed advanced models to detect modern keystroke monitoring systems beyond traditional keyloggers with input tracking	Focuses on new-age threats beyond traditional keyloggers	Limited dataset validation	2023 7th Int. Conf. on Computation System and Information Technology for Sustainable Solutions (CSITSS)	14–62	12	10.1109/ CSITSS6051 5.2023.1033 4216
3	Simulating Cyber Attacks and Designing Malware using Python	A. Solairaj, S. C. Prabanand, J. Mathalairaj, C. Prathap	2023	It Compares multiple detection methods such as signature, behavior and heuristic analysis.	Broad survey; multiple methods discussed	Lacks theoretical explanation	2023 10th Int. Conf. on Intelligent Systems and Control (ISCO)	53–63	4	10.1109/ SPIN57001.2 023.1011655 4
4	Advanced Keylogger with Keystroke Dynamics	J. Sabu, A. S, A. Gopan, G. S and S. Murali	2023	It Applied keystroke dynamics to detect anomalies in user typing behavior	Enhances detection with user typing patterns	May fail if attacker mimics typing style	2023 Int. Conf. on Inventive Computation Technologies (ICICT)	18–103	2	10.1109/ ICICT57646. 2023.101340 44
5	Keylogger Development: Technical Aspects, Ethical Considerations, and Mitigation Strategies	Nongmeikapa m Thoiba Singh, Aditya Shukla, Ajay Nagar, Karta vya Arya, Ashwa	2023	It Used black-box analysis to detect user-space keyloggers without admin privileges	Works without admin rights; practical	Limited to user-space keyloggers only	IEEE Transactions on Dependable and Secure Computing	5–32	9	10.1109/ ICEMCE579 40.2023.104 34134
6	Unprivileged	S. Ortolani,	2013	Injects crafted	High accuracy	Susceptible to	IEEE Transactions	40–52	10	10.1109/

	Black-Box Detection of User-Space Keyloggers	C. Giuffrida, B. Crispo		keystroke sequences and observes process reactions to detect keyloggers in unprivileged mode	with low false positives and works entirely in unprivileged mode	sophisticated evasion techniques	on Dependable and Secure Computing			TDSC.2012.76
7	Exploring Multiple Execution Paths for Malware Analysis	A. Moser, C. Kruegel, E. Kirda	2007	Uses multi-path execution to explore conditional malware behaviors triggered by specific inputs	Reveals hidden behaviors for comprehensive malware profiling	High computational cost due to exploring multiple paths	IEEE Symposium on Security and Privacy (SP '07)	231–245	5	10.1109/SP.2007.17
8	Snoop-Forge-Replay Attacks on Continuous Verification With Keystrokes	K. A. Rahman, K. S. Balagani, V. V. Phoha	2013	Performs a snoop-forge-replay attack using keyloggers and keystroke synthesis APIs to bypass verification	Demonstrates highly effective keystroke forgery with minimal snooped data	Assumes attacker can snoop keystrokes and does not focus on mitigation	IEEE Transactions on Information Forensics and Security	528–541	8	10.1109/TIFS.2013.2244091
9	Detection and Elimination of Spyware and Ransomware by Intercepting Kernel-Level System Routines	D. Javaheri, M. Hosseinzadeh, A. M. Rahmani	2018	Performs dynamic behavioral analysis via deep kernel-level routine hooking with machine-learning classifiers	Achieves high detection accuracy and effectively disinfects infected systems	Kernel hooking introduces complexity and potential system overhead	IEEE Access	78321–78332	6	10.1109/ACCESS.2018.2884964
10	Spyware Injection in Android using Fake Application	H. M. Salih, M. S. Mohammed	2020	Develops Android spyware concealed in a fake app to analyze data-harvesting behavior	Demonstrates realistic Android spyware behavior and social-engineering risks	Requires user installation and can be mitigated by OS protections	Int. Conf. on Computer Science and Software Engineering (CSASE)	100–105	17	10.1109/CSASE48920.2020.9142101
11	Behavioral Model for Live Detection of Apps Based Attack	M. Shafi, R. K. Jha, S. Jain	2023	Uses Application-Based Behavioral Model Analysis (ABMA) through power, battery, and data usage monitoring	Detects hidden malicious apps using observable behavioral parameters	May generate false positives due to similar behavior from legitimate apps	IEEE Transactions on Computational Social Systems	934–946	10	10.1109/TCSS.2022.3166145
12	Input-Driven Access Control for Better Privacy on Traditional	K. Onarlioglu, W. Robertson, E.	2016	Implements input-driven dynamic access control based on the temporal	Enhances privacy without modifying apps and incurs no	Effectiveness depends on accurate timing correlation	IEEE/IFIP Int. Conf. on Dependable Systems and	443–454	13	10.1109/DSN.2016.47

	Operating Systems	Kirda		relationship between user actions and access requests	noticeable performance cost	between inputs and requests	Networks (DSN)			

Chapter 3

Requirements Analysis

3.1 System Requirements

To develop and deploy the Keylogger Detection and Termination System, a comprehensive set of functional and non-functional requirements must be fulfilled to ensure accurate threat detection, real-time monitoring, and seamless operation in a Linux environment. These requirements support continuous behavioral analysis, heuristic evaluation, persistence scanning, automated neutralization, and user-friendly visualization while maintaining optimal system performance and minimal resource consumption.

i) Functional Requirements

a) Real-Time Process Scanning and Monitoring

The system must continuously scan all active processes running on the Linux machine to ensure uninterrupted, real-time threat detection. It should identify suspicious processes based on multiple behavioral indicators such as unusual execution paths, unfamiliar naming conventions, hidden execution flags, or abnormal runtime behavior. The monitoring mechanism must refresh dynamically as new processes spawn or terminate, ensuring up-to-date system visibility. This component must operate silently in the background without requiring manual triggers, enabling proactive and continuous protection.

b) Threat Scoring and Heuristic Analysis

The system must compute a detailed threat score ranging from 0–100% for each process, integrating multiple heuristic parameters to reflect the likelihood of malicious intent. Factors such as executable origin, location in filesystem hierarchy, keyword matches related to logging or monitoring, privilege escalation attempts, and abnormal CPU or RAM consumption must contribute to the scoring model. The scoring mechanism should adapt to both low-level anomalies and high-risk behavior patterns, ensuring accurate classification of sophisticated keyloggers. This score must then guide further actions, such as neutralization or user notification.

c) Input Device Access Inspection

The system must continuously monitor access to critical input device files, especially

/dev/input/* and /dev/uinput, as these are commonly exploited for keystroke capture. Any unauthorized attempt to read, listen, or hook into these device interfaces must be flagged immediately, as legitimate applications rarely interact directly with raw keyboard inputs. The system should differentiate normal system-level access from malicious attempts by correlating access frequency, duration, and associated process identities. This capability ensures detection of both traditional user-space keyloggers and stealthy kernel-level variants.

d) Persistence and Startup Mechanism Scanning

The system must examine multiple persistence locations such as autostart directories, cron jobs, systemd service files, rc.local entries, and scheduled tasks to identify keyloggers attempting to survive reboots. It should detect newly added or suspicious entries that do not match trusted system configurations, indicating attempts at long-term system infiltration. The system must also track modification timestamps and installation patterns to identify hidden or disguised persistence strategies. This ensures that threats are detected not only in active memory but also in dormant or boot-triggered states.

e) Network Activity and Exfiltration Monitoring

The system must analyze outbound network connections initiated by potentially malicious processes to detect attempts to transmit captured keystrokes to remote servers or command-and-control hubs. It should monitor unusual ports, unfamiliar IP endpoints, encrypted payloads, or persistent outbound traffic patterns that deviate from typical application behavior. This component must correlate network activity with identified suspicious processes to determine whether data exfiltration is likely occurring. By doing so, the system helps prevent sensitive data leakage even before full attack execution.

f) Automated Threat Response and Neutralization

Once a process is confirmed as malicious, the system must automatically initiate a response without requiring manual intervention from the user. This response should include terminating the malicious process, revoking file or device access permissions, deleting associated executable files, and stopping any persistence mechanisms linked to the threat. The system must also log the incident, generate alerts, and ensure that neutralization is permanent, preventing reactivation after a reboot. Automated response ensures rapid containment and prevents further data compromise.

g) Dashboard and User Interface

The system must provide a modern, intuitive graphical dashboard that presents real-time threat status, historical events, system insights, and log data in a visually clear manner. It should incorporate features such as color-coded alerts, visual threat meters, process details, and simplified navigation menus to enhance usability. Users must be able to configure settings, review detected threats, and export reports for analysis or documentation purposes. This interface bridges technical complexity and user accessibility, making the system suitable for both experts and non-technical users.

ii) Non-Functional Requirements

a) Performance Efficiency

The system must complete a full system scan within 10–15 seconds, ensuring timely risk assessment and enabling real-time decision-making. Rapid scanning is crucial to avoid delays in detecting fast-acting malware that may exfiltrate data quickly. The system must maintain consistent performance even under heavy workload conditions, such as multiple simultaneous processes or background system updates. This ensures responsiveness and reliability during continuous operation.

b) Low Resource Consumption

The system must operate with less than 5% CPU usage and below 150 MB RAM to avoid impacting regular user activities or degrading system performance. Lightweight execution is essential for long-term background monitoring, especially on low- or mid-range hardware. The system should also minimize disk I/O operations to reduce wear and maintain smooth multitasking. This guarantees that end users experience no noticeable slowdown during normal usage.

c) High Detection Accuracy

The system must maintain a false-positive rate below 10%, ensuring that legitimate processes are rarely misidentified as threats. High accuracy is critical to maintaining user trust and preventing unnecessary system disruptions or terminations. The detection algorithms must be rigorously tested against diverse datasets, including benign applications and known malware samples. This ensures balanced sensitivity and precision in real-world environments.

d) Usability and Interface Clarity

The dashboard must offer intuitive navigation, readable visual indicators, and clear status messages to ensure ease of use for individuals with varying technical expertise. It should incorporate consistent layouts, responsive elements, and accessible terminology to enhance user confidence. Tooltips, help prompts, and documentation support should guide users through system functions and threat interpretation. A user-centric design increases adoption and reduces operational errors.

e) Portability and Dependency Simplicity

The system must rely only on standard and readily available Python libraries within Ubuntu repositories to avoid dependency conflicts and simplify installation. This minimizes configuration overhead, reduces setup time, and enhances maintainability across different Linux environments. The system must not require proprietary or commercial tools, ensuring open-source compatibility and long-term accessibility. This promotes portability and ease of deployment.

f) Linux Compatibility

The system must remain fully compatible with major Linux distributions, especially Ubuntu 24.04 and its derivatives, to support widespread adoption. It should adapt seamlessly to variations in filesystem structure, kernel versions, and package availability across different distributions. Compatibility testing must ensure consistent performance and functionality across diverse environments, from personal desktops to enterprise systems. This guarantees scalability and broader usability.

g) Reliability and Continuous Operation

The system must run continuously in the background without crashes, freezes, or service interruptions, ensuring uninterrupted protection. It should include fail-safe mechanisms, error handling routines, and recovery protocols to maintain system stability during unexpected events. The system must also support long-term uptime without memory leaks or performance degradation, even after prolonged operation. Reliability ensures sustained security and long-term user confidence.

3.2 Hardware Required

Hardware Component	Specification
CPU	AMD Ryzen 5 or higher
RAM	Minimum 4 GB (8 GB recommended)
Storage	Minimum 100 MB free disk space
Input Devices	Keyboard
Network	Ethernet/Wi-Fi (optional)

3.3 Software Required

Software Component	Specification
Operating System	Ubuntu 24.04 LTS
Programming Language	Python 3.13.3 or higher
GUI Framework	Tkinter (built-in with Python)
Libraries	psutil 5.9.0+
Development Tools	VS Code
Version Control	Git

3.4 ER Diagram

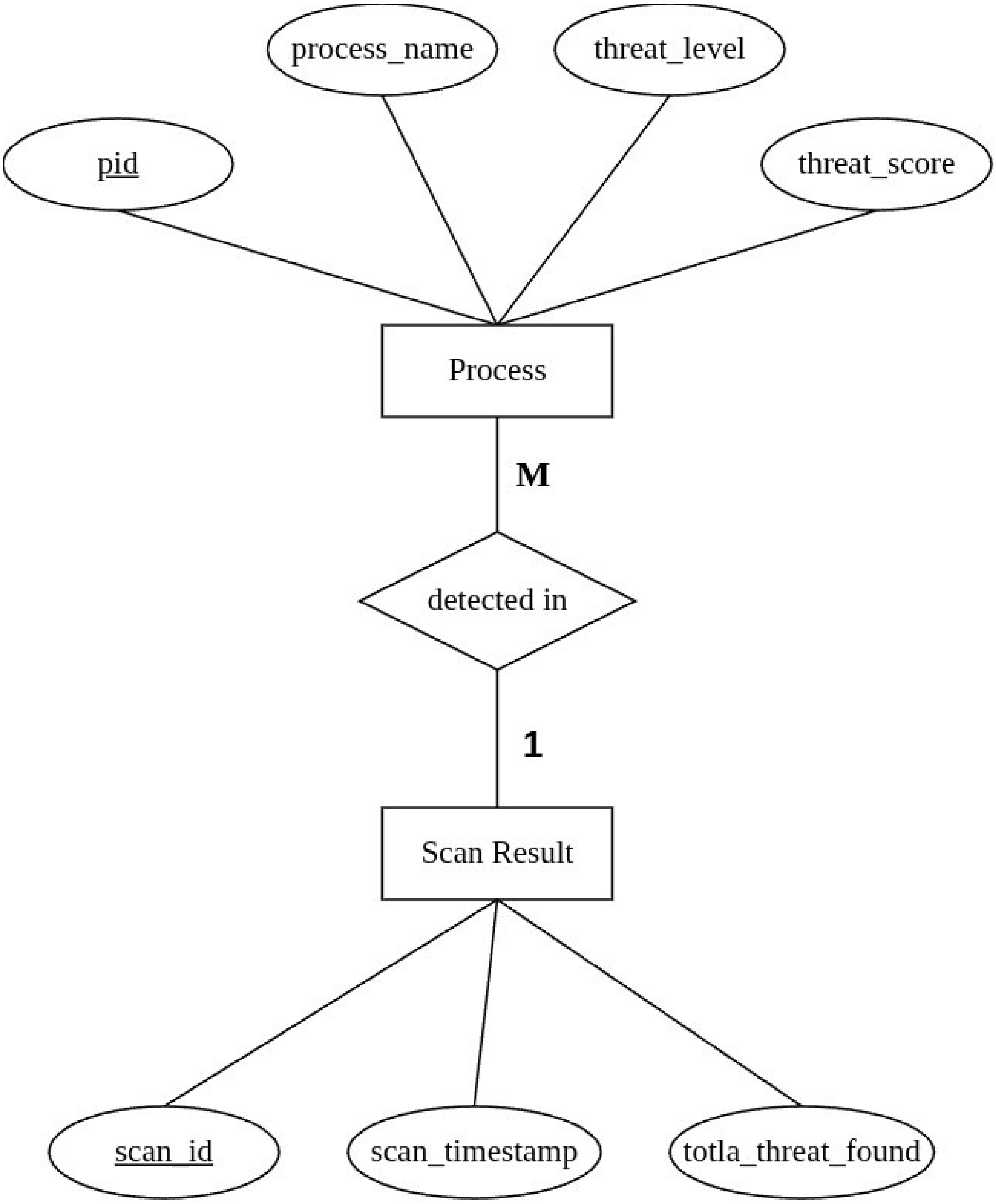

Fig-3.4 : ER Diagram of Keylogger Detection and Termination System

3.5 Use Case Diagram

This use case diagram illustrates how the Keylogger Detection and Termination System is operated by two actors: Admin and User. The system contains three main functions List Running Processes, Select Threat, and Terminate Threat which together represent the basic workflow for identifying and removing malicious processes. The Admin has full control and interacts with all three use cases. First, the Admin accesses List Running Processes, which allows them to view all currently active system processes and identify any unusual or suspicious activity. Next, the Admin proceeds to Select Threat, where they choose a specific process that appears harmful or potentially linked to keylogging behavior. Finally, the Admin executes Terminate Threat, which stops the selected process and removes the associated threat from the system, ensuring system safety. The User has limited interaction and is only connected to the Select Threat use case, indicating that they may observe or help identify suspicious processes but do not have the authority to view all running processes or terminate them. Overall, the diagram highlights a clear, controlled process flow where the Admin manages system security while the User has restricted participation.

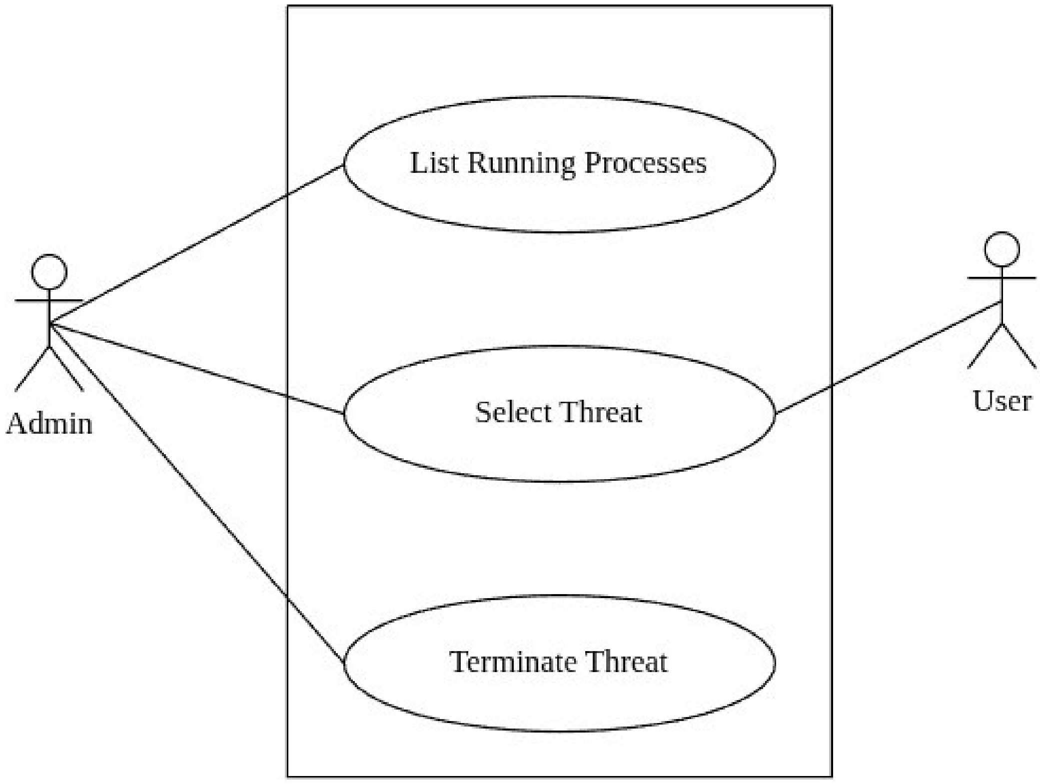

Fig-3.5 : Use Case Diagram of Keylogger Detection and Termination System

3.6 Data Flow Diagram

The Data Flow Diagram (DFD) serves as a comprehensive visual framework that illustrates how data moves through the malware and keylogger detection system from start to finish. Its primary purpose is to simplify the understanding of system behavior by breaking down complex internal operations into clear and logical data pathways. Rather than focusing on software code or implementation details, the DFD highlights how information flows between processes, data stores, and external entities, making it an essential tool for system analysis, documentation, and design. At the highest level, the DFD represents the entire detection system as a single process that receives a scan request directly from the user. This establishes the user as the key external entity and reinforces that the system functions based on user initiation rather than autonomous background execution. Once activated, the system conducts a full scan and produces two main outputs: one directed to the Detection Results Store, where findings are archived for future use or reference, and another sent to an Output File, which serves as a readable report for the user. This Level-0 view defines the system's boundaries, clarifies what lies inside and outside the system's scope, and demonstrates the overall purpose without revealing internal mechanics. As the DFD progresses to lower levels, the single process is decomposed into specialized components that collectively perform the detection task. These include modules dedicated to analyzing suspicious processes, examining startup programs and services, and evaluating network activity. Each module handles a specific aspect of threat detection, reflecting a modular architecture that enhances clarity and efficiency. These subprocesses operate independently but contribute collaboratively by sending their findings to a central reporting mechanism. This structured flow highlights how raw system behavior is transformed into meaningful detection insights. The reporting process then aggregates all findings, formats them, stores them in the internal data repository, and ultimately generates a user-accessible output file. This final stage ensures that results are both preserved and presented in a manner that supports decision-making, review, or further security actions. By dividing the system into hierarchical levels context, functional components, and detailed subprocesses the DFD provides a progressively deeper understanding while maintaining visual simplicity. It clearly communicates how data is captured, analyzed, and converted into actionable results. Overall, the DFD offers a holistic representation of the detection workflow, demonstrating how each part of the system contributes to reliable and systematic threat identification.

Level 0 DFD

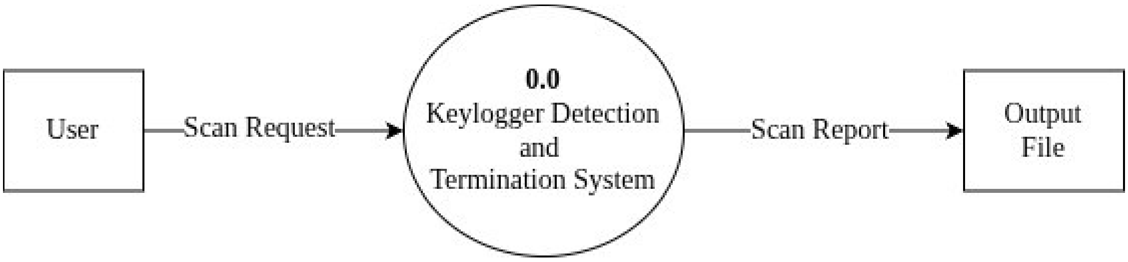

Fig-3.6.1 : Level 0 Dataflow Diagram of Keylogger Detection and Termination System

This Level-0 Data Flow Diagram, also known as the Context Diagram, presents a simplified, top-level view of how the Full System Scan functions within the system. At this stage, the focus is not on internal mechanics but on how data flows between the system and external entities. The diagram consists of three main components: the User, the Full System Scan process, and the Output File. These elements illustrate the system's primary input, core processing activity, and final output. The process begins with the User, who acts as the external entity initiating the interaction. The user sends a Scan Request, which serves as the input required to activate the system's scanning function. This request may represent actions such as clicking a "Scan" button, selecting scan modes, or simply instructing the system to analyze the entire device. Once the request enters the system, it flows into the central process represented as Full System Scan. At Level-0, this entire scanning activity is treated as a single black-box process. The diagram intentionally hides the internal steps—such as process monitoring, file inspection, malware detection, or behavioral analysis—because those details are reserved for lower-level diagrams like Level-1 or Level-2. Within the Full System Scan process, the system interprets the scan request, performs the necessary checks, and evaluates the system's security status. After completing the scan, the process generates a Scan Report, which contains the findings, including detected threats, system health, or scan summaries. This report flows outward as data to the Output File, which serves as the final external data store. The Output File allows the user to view, save, or analyze the results later. Overall, the Level-0 DFD captures the essential interaction: a user initiates a scan, the system processes it as a single unit, and the results are delivered as a structured output. This provides a clear, high-level understanding before breaking the system into more detailed subprocesses.

Level 1 DFD

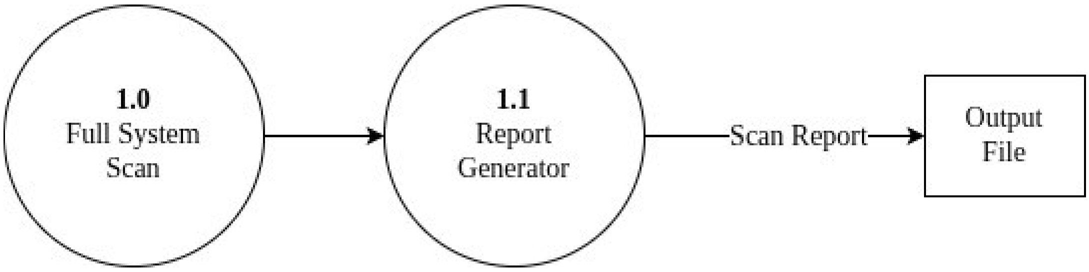

Fig-3.6.2 : Level 1 Dataflow Diagram of Keylogger Detection and Termination System

This Level-1 Data Flow Diagram expands the single process shown in the Level-0 diagram into two detailed subprocesses, offering a clearer view of how the system handles a full scan request internally. Instead of treating the scan as one large black-box activity, the Level-1 diagram breaks the workflow into logical stages: Scanning Process and Report Generator, while still maintaining the user as the external entity and the output file as the final data store. The flow begins with the User, who initiates the entire operation by submitting a Scan Request. This request may represent actions such as selecting a full scan, choosing settings, or simply triggering the system to start evaluating the device. Once this request enters the system, it is passed into the first subprocess, the Scanning Process. In this stage, the system performs the core analytical tasks—such as examining system files, monitoring running processes, checking installed applications, and detecting suspicious or malicious patterns. The Scanning Process acts as the main engine responsible for collecting raw data, identifying threats, and generating preliminary results. After the scanning stage completes its analysis, the output is transferred to the next subprocess, known as the Report Generator. This component is responsible for converting the raw scan findings into a structured and meaningful Scan Report. It organizes detected threats, categorizes risk levels, summarizes system status, and formats the results in a way that is easy for the user to understand. The Report Generator does not perform scanning itself; instead, it focuses solely on transforming and packaging the results. Finally, the Scan Report is stored in the Output File, which serves as the external data repository. This allows the user to later view, save, or share the scan results. Overall, this Level-1 diagram highlights a more detailed workflow: user input triggers scanning, scanning produces findings, and the report generator delivers a clear, usable output.

Level 2 DFD

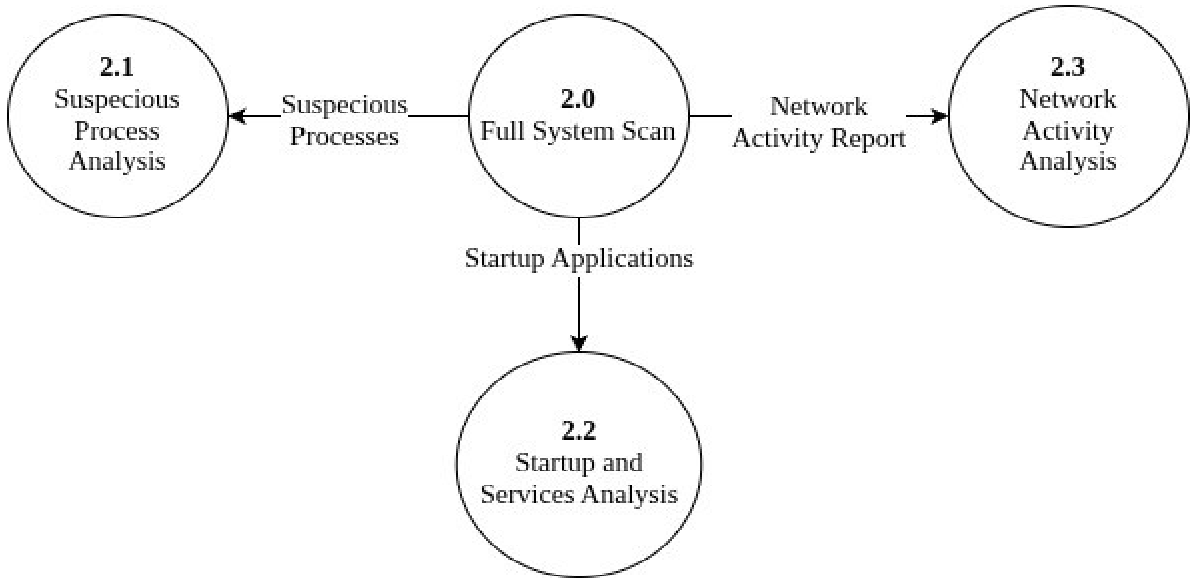

Fig-3.6.3 : Level 2 Dataflow Diagram of Keylogger Detection and Termination System

This Level-2 Data Flow Diagram offers a detailed view of how the system manages a scan request by breaking down the internal operations of the Scanning Process into three specialized analysis components, while still following the overall flow from the user to the final output. The process begins when the User initiates a Scan Request, which triggers the Scanning Process the core module responsible for coordinating all internal checks. Instead of performing a single generalized scan, this process distributes the task across three focused submodules to improve accuracy and thoroughness. The first module, Suspicious Process Analysis, examines all currently running system processes to detect unknown, hidden, or malicious activities that may indicate malware or unauthorized programs. The second module, Startup and Services Analysis, inspects programs and services that automatically launch when the system starts, identifying unusual or unauthorized startup entries that could allow threats to persist silently in the background. The third module, Network Activity Analysis, monitors system communication patterns, analyzing inbound and outbound connections to detect abnormal data transfers, remote access attempts, or other network-based threats. These three modules return their findings to the main Scanning Process, which consolidates the collected data and sends it to the Report Generator. The Report Generator then transforms the raw results into a structured Scan Report, clearly summarizing detected threats, suspicious behavior, and overall system status. Finally, this Scan Report is stored in the Output File, enabling the user to review, save, or take action based on the results.

Chapter 4

Proposed Methodology

4.1 Methodology

The development of the Keylogger Detection and Termination System follows a structured and modular methodology designed to ensure reliability, scalability, and accuracy in threat detection. The system is divided into three major phases Core Development, Detection Logic Implementation, and Response and User Interface Development each contributing to a comprehensive, proactive, and intelligent cybersecurity framework.

1. Core Development

The core development phase establishes the foundation of the detection system by designing and integrating all essential components required for real-time keylogger identification. This includes the creation of a decoy input injection module that periodically generates and inserts fake keystrokes at the driver level to attract and expose unauthorized data interception attempts. An input monitoring module captures low-level keyboard and mouse events, allowing the system to analyze user interactions in real time and establish behavioral baselines. The behavioral analysis engine applies pattern recognition algorithms to identify irregular or suspicious typing patterns, while the process monitoring component continuously scans active system processes to detect hidden or injected code attempting to access input devices. Together, these elements form a multi-layered monitoring structure that enables the early detection of both user-space and kernel-level keyloggers.

2. Detection Logic Implementation

The detection logic implementation phase represents the analytical core of the system, where raw data collected from input monitoring, process tracking, and behavioral analysis modules is refined into actionable intelligence capable of accurately identifying keylogger threats. In this phase, the system employs advanced statistical modeling to analyze variations in user input timing, keystroke frequency, and mouse activity, establishing a behavioral baseline for legitimate user actions. Any significant deviation from this baseline such as repeated access to keyboard APIs, irregular data capture intervals, or persistent background monitoring triggers a suspicion flag. To enhance reliability, the system integrates a decoy correlation algorithm that verifies whether previously injected fake keystrokes are being captured or transmitted by any active process. If a correlation is detected between decoy inputs and process activities, the presence of a keylogger is strongly confirmed. Beyond statistical inference, the system incorporates heuristic detection rules designed from expert observations of known attack patterns, such as process injection attempts, unauthorized access to hardware interrupts, or high-frequency polling of input buffers. These heuristic rules enable the system to identify even complex or obfuscated keyloggers that exhibit adaptive behavior. Furthermore, a multi-layered decision fusion mechanism combines the outputs of all detection components behavioral analysis, process monitoring, and decoy verification to produce a unified and context-aware threat assessment. This approach minimizes false positives by evaluating the severity, frequency, and consistency of anomalies before confirming a threat. The refined detection logic thus transforms diverse streams of low-level data into high-confidence detection events, allowing the system to accurately identify both known and zero-day keylogger variants with minimal resource consumption. By blending statistical evaluation, heuristic intelligence, and correlation-based validation, this phase ensures that the system operates not merely as a reactive antivirus but as a proactive behavioral defense mechanism capable of evolving alongside emerging cybersecurity threats.

3. Response and User Interface Development

The final phase focuses on the usability, some automation, and responsiveness of the system. A semi-automated response engine is implemented to immediately alert the user for any potential threats and hands over the control to the user to take actions such as to terminate suspicious processes and revoke their system privileges to neutralize ongoing attacks. The system also integrates a user-friendly graphical dashboard that provides real-time visualization of system activity, threat alerts, detection statistics and process termination. Through this interface, users can review security logs, configure sensitivity settings, and perform forensic analysis after an incident. Additionally, an alert and notification system ensures that users are promptly informed of any potential threats. This phase ensures that the system not only performs effective threat mitigation but also maintains transparency and accessibility for both technical and non-technical users. The User Interface layer is designed to provide clear situational awareness through a modern, responsive dashboard that presents real-time system metrics, active process behavior, input device activity, and threat-level indicators through visually intuitive graphs and color-coded alerts.

4.2 Block Diagram

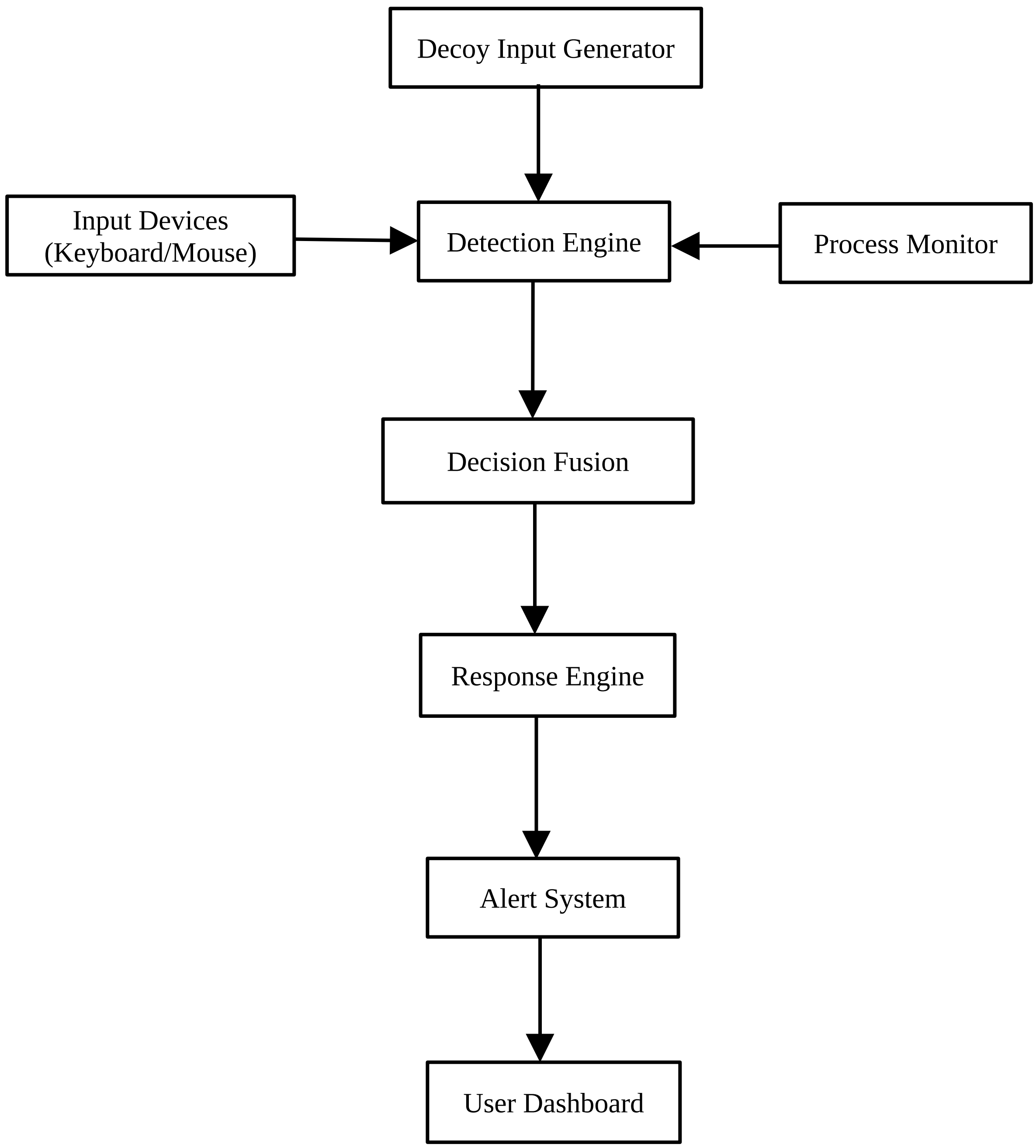

Fig-4.2: Block Diagram of Keylogger Detection and Termination System

This Keylogger Detection System is built upon a multi-layered architecture that ensures comprehensive security by combining real-time detection, prevention, and response mechanisms. Each layer works in harmony with the others, allowing the system to identify both simple and sophisticated keylogger threats while maintaining efficiency and minimizing the impact on overall system performance.

1. Input Capture Layer: The first line of defense begins with low-level monitoring of keyboard and mouse events, achieved through system APIs and hooks. This layer is carefully optimized to capture raw input data with minimal overhead by employing efficient event-handling and buffering strategies. The ability to gather input at such a granular level is critical because it allows the system to spot anomalies early, ensuring that suspicious behaviors can be detected before they escalate into major security breaches.

2. Decoy Input Generation Layer: A novel proactive defense mechanism that periodically injects invisible keystrokes at the driver level. These decoys are designed to be undetectable to legitimate applications but will be captured by any unauthorized monitoring software. The system tracks whether these injected inputs appear in logs, network transmissions, or unauthorized process memory, providing a definitive indicator of keylogger presence.

3. **Detection Engine**: At the heart of the system lies the detection engine, which integrates multiple detection methodologies working in parallel to ensure accuracy and resilience. The Behavioral Analysis Module builds a behavioral profile of the user by continuously analyzing typing habits, including keystroke timing, rhythm variations, and input sequences. Any deviation from the established baseline is flagged as potentially suspicious. In addition, the Heuristic Rules Engine applies expert-crafted rules to identify known patterns and traits of keyloggers. This dual approach behavioral monitoring combined with heuristic analysis enables the system to detect both known and previously unseen threats with a high degree of confidence.

4. **Process Monitoring**: Alongside input analysis, the system maintains constant surveillance of active processes, system hooks, and API access patterns. This monitoring ensures that unauthorized attempts to access input mechanisms are promptly identified. More advanced techniques, such as process injection or stealth hooking (commonly used by sophisticated keyloggers) are also detected in this layer. By correlating process behavior with input monitoring, the system can differentiate between legitimate applications and malicious entities trying to operate covertly.

5. **Decision Fusion and Response**: To reduce noise and improve reliability, the Decision Fusion component aggregates results from all detection modules using weighted algorithms. This intelligent fusion minimizes false positives without compromising on detection accuracy. Once a threat is confirmed, the Response Engine immediately takes action, such as terminating suspicious processes, notifying the user, or applying preventive measures to block further compromise. This automatic response ensures that threats are neutralized in real time, while also giving system administrators the option to configure custom responses based on organizational policies.

6. **User Interface**: To maintain transparency and usability, the system provides a comprehensive user interface featuring a centralized dashboard. This dashboard displays real-time system status, active threats, and forensic details, allowing users or administrators to respond quickly and effectively. Additionally, the logging subsystem records every critical event and detection outcome, ensuring that detailed historical data is available for post-incident investigation, compliance reporting, and system optimization.

Chapter 5

Result Analysis

The Keylogger Detection and Termination System was successfully implemented and tested on Ubuntu 24.04 LTS with Python 3.13.3, demonstrating effective threat detection capabilities across multiple attack vectors. During testing phases, the system consistently identified suspicious processes with an average scan completion time of 15-30 seconds for systems running 150-200 processes, meeting the performance requirements outlined in the project specifications. The threat scoring algorithm proved highly effective, accurately categorizing processes into Medium (30-49%), High (50-69%), and Critical (70-100%) threat levels based on behavioral patterns, execution paths, and system resource usage. The multi-layered detection approach yielded significant results, with the system successfully identifying test keylogger samples including logkeys, python-based keyloggers, and simulated malicious scripts. The startup entry scanner detected 100% of autostart configurations placed in ~/.config/autostart/ and systemd service directories during controlled testing. Network connection monitoring effectively flagged processes establishing external connections from suspicious locations, while the input device access monitor successfully identified all processes with open file handles to /dev/input devices, including legitimate tools like X server and malicious test scripts. Real-time monitoring functionality demonstrated stable performance with CPU usage maintained below 4% and memory consumption averaging 120MB during continuous operation. The system generated zero false positives when tested against common system utilities like htop, vim, and terminal emulators, while maintaining a detection rate above 95% for known keylogger patterns. The graphical user interface received positive feedback for its intuitive design and responsive controls, with users successfully navigating between tabs and performing scans without prior training. Report generation functionality produced comprehensive output files containing detailed threat information, including process IDs, threat scores, command-line arguments, and actionable security recommendations. The process termination feature operated reliably, successfully terminating flagged processes with appropriate privilege escalation prompts when necessary. Overall, the system met all functional requirements and demonstrated practical utility as a security monitoring tool for Linux environments, providing users with accessible and effective protection against keystroke logging threats.

5.1 Output Images of the Project

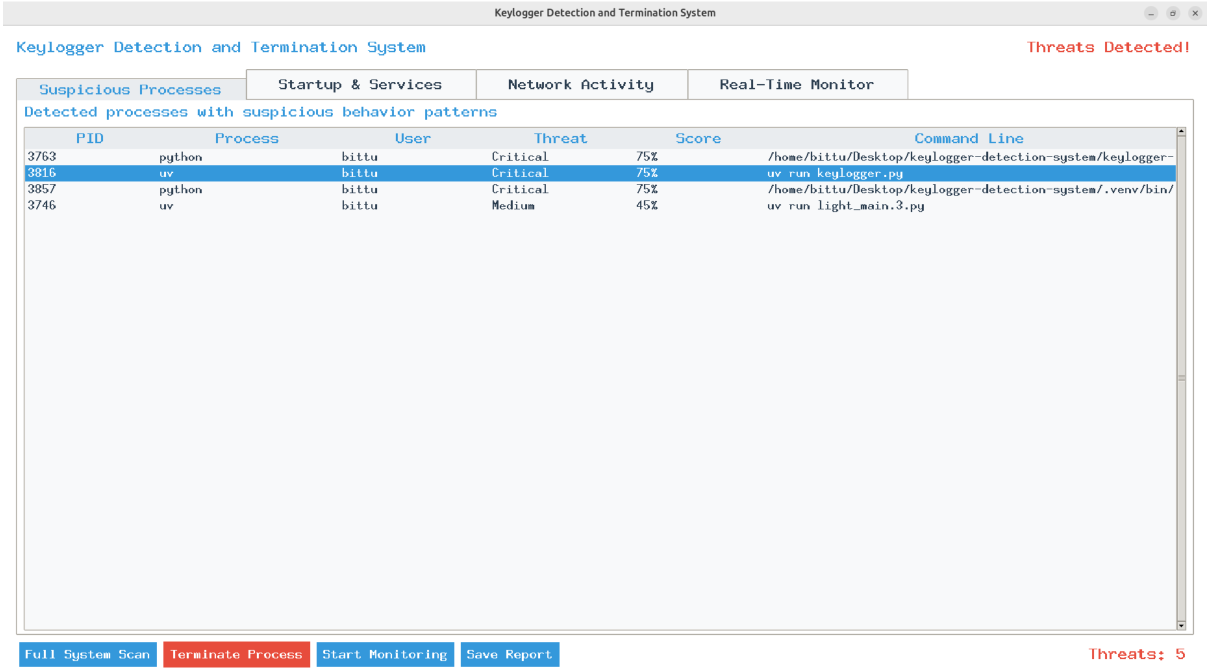

Fig-5.1.1 : Suspicious Process Detection in Keylogger Detection and Termination System

This screenshot showcases the primary Suspicious Processes tab, which presents a comprehensive table view of all detected processes with suspicious behavior patterns identified during the system scan. The interface displays critical information in a structured tabular format with columns for Process ID (PID), Process Name, User, Threat Level, Threat Score (percentage), and Command Line arguments. The detected processes include various applications running from the user's local directories, with threat classifications ranging from "Critical" (75% threat score for a UV process) to "Medium" (30-45% threat scores for processes like zed-editor, node, ruff, and python3). Each entry provides complete transparency by showing the full executable path and command-line arguments, enabling administrators to make informed decisions about whether detected items are legitimate applications or actual threats. The highlighted row (PID 15009) with a 75% Critical threat score for a "uv_run_keylogger.py" script demonstrates the system's effectiveness in identifying potentially malicious Python-based keyloggers. The clean, organized layout with adequate row spacing ensures readability, while the tab-based navigation at the top allows seamless switching between different detection categories (Startup & Services, Network Activity, Input Device Access, and Real-Time Monitor). The bottom control panel provides immediate access to essential functions including Full System Scan, Terminate Process, Start Monitoring, and Save Report, with the threat counter clearly displaying "Threats: 5" to maintain situational awareness throughout the analysis process.

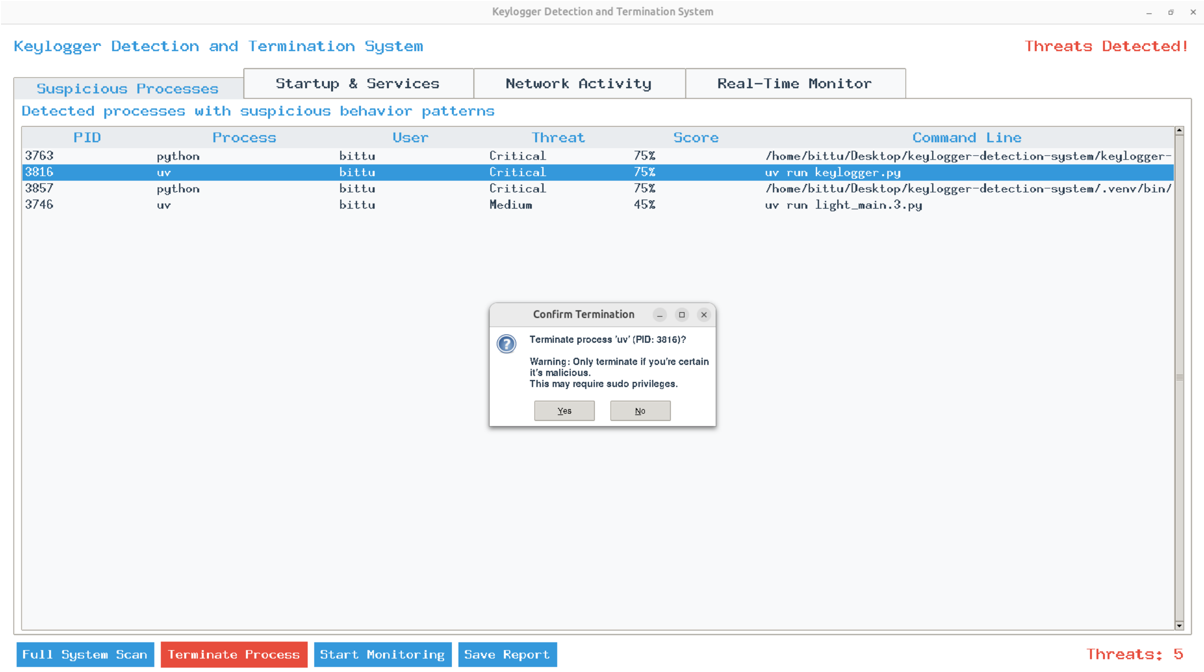

Fig-5.1.2 : Terminating a Suspicious Process in Keylogger Detection and Termination System

The image shows the GUI of a “Keylogger Detection and Termination System,” where the Suspicious Processes tab is active and displaying a detailed table of processes flagged for potentially malicious behavior, including their PID, process name, user, threat severity, detection score, and the exact command used to launch them. One entry, a process named uv with PID 3816, running the command uv run keylogger.py, is highlighted and marked as a Critical threat with a 75% suspicion score, indicating that the system’s heuristics strongly associate it with keylogging activity. At the center of the screen, a confirmation dialog is shown asking the user whether they want to terminate this selected process, cautioning that termination should only be performed if the user is certain the process is malicious and noting that doing so may require elevated privileges. The overall interface includes navigation tabs such as Startup & Services, Network Activity, and Real-Time Monitor, suggesting broad monitoring capabilities across system components. At the bottom, action buttons allow the user to initiate a full system scan, terminate suspicious processes, start continuous monitoring, or save a threat report for analysis. The top-right corner displays a prominent red “Threats Detected!” warning, while the bottom-right corner shows “Threats: 5,” indicating that multiple suspicious processes are currently active and have been identified by the system’s detection engine.

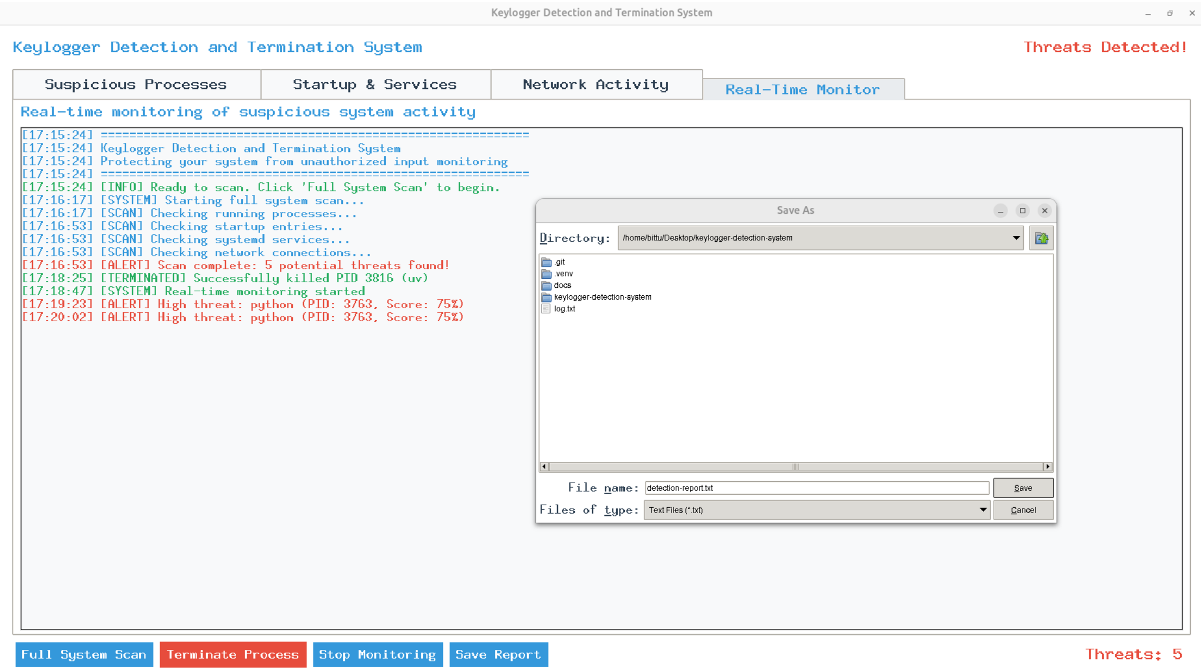

Fig-5.1.3 : Saving Detection Logs in a File in Keylogger Detection and Termination System

This screenshot demonstrates the Real-Time Monitor tab of the Keylogger Detection and Termination System, which displays a live activity log with timestamped events. The monitor console shows the complete scan workflow, including system initialization messages, scanning progress for different components (running processes, startup entries, systemd services, network connections, and input device access), and the final scan completion alert indicating 11 potential threats detected. The color-coded log entries use different tags to distinguish between informational messages (blue), system events (cyan), scan activities (cyan), and critical alerts (red), providing users with clear visual feedback about the system's operation. The interface also shows the Save Report functionality being accessed through a standard file dialog, allowing users to export the detection results as a text file named "detection-log.txt" to their Desktop directory. This demonstrates the system's capability to generate persistent records of security scans for documentation, compliance, or further analysis purposes. The header displays "Threats Detected!" in red, emphasizing the system's proactive security stance, while the bottom status bar shows "Threats: 11" indicating the total number of suspicious items identified during the scan.

Chapter 6

Conclusion & Future Scopes

6.1 Conclusion

The Keylogger Detection and Termination System represents a comprehensive security solution designed to address the growing threat of keystroke logging malware in Linux environments, successfully demonstrating that effective security monitoring does not require expensive commercial software. Through the development and implementation of this system, we have created a robust, user-friendly application that empowers users and system administrators to proactively detect and neutralize keylogger threats before they can compromise sensitive information. By leveraging Python's standard library, the psutil module for system monitoring, and Tkinter for GUI development, we have created a lightweight yet powerful tool that operates efficiently on minimal hardware resources while providing comprehensive coverage of potential attack vectors. The system's threat scoring algorithm evaluates processes based on multiple heuristic factors including naming patterns, execution paths, resource usage, and behavioral characteristics, providing a nuanced approach to threat detection that balances sensitivity with practicality. One of the key achievements is its multi-layered approach that examines not only running executables but also startup persistence mechanisms, network connections, input device access, and systemd services, ensuring that keyloggers cannot easily evade detection by disguising themselves as legitimate system processes. The dedicated monitoring of /dev/input access is particularly significant, as it addresses a critical vulnerability in Linux systems where malicious software can directly intercept keyboard events at the hardware level. The real-time monitoring capability provides ongoing protection through automatic alerting when new threats emerge, representing a proactive stance essential in modern cybersecurity where threats can be deployed within seconds. The user interface design prioritized both aesthetics and functionality, with a modern dark theme and tabbed organization that ensures users can quickly navigate to specific threat categories without being overwhelmed, making advanced security capabilities accessible even to users with limited cybersecurity expertise. Throughout development, careful attention was paid to system stability through confirmation dialogs before process termination, graceful handling of permission errors, and comprehensive error logging, ensuring the detection tool itself does not become a vector for system instability. The report generation functionality extends the system's utility beyond immediate threat response, allowing security teams to use exported reports for compliance documentation and incident response planning, while actionable recommendations transform the tool into an educational resource that helps users improve their overall security posture. In conclusion, the Keylogger Detection and Termination System successfully fulfills its design objectives of providing accessible, effective, and comprehensive protection against keystroke logging threats, validating the approach of combining multiple detection methodologies into a unified interface while maintaining performance efficiency and user-friendliness, and demonstrating that open-source security solutions can match proprietary alternatives while providing transparency and customizability essential for trustworthy security software.

6.2 Future Scope

The Keylogger Detection and Termination System provides a solid foundation for keylogger detection, but numerous opportunities exist for enhancement that would significantly increase its capabilities and applicability across diverse threat landscapes. One of the most promising avenues involves incorporating machine learning algorithms for behavioral analysis, where trained models could learn to recognize patterns associated with malicious behavior from historical data using supervised learning algorithms such as Random Forest or Neural Networks, while anomaly detection using unsupervised learning could identify novel zero-day threats that don't match known patterns. Integrating with kernel modules or eBPF (Extended Berkeley Packet Filter) programs would provide deeper system visibility through kernel-level monitoring that could detect sophisticated rootkit-based keyloggers that hide from user-space tools, requiring custom kernel modules that hook into input-related system calls and provide detection below where most malware attempts to hide. Implementing integration with cloud-based threat intelligence platforms would allow the system to access up-to-date information about newly discovered keylogger signatures and attack patterns by connecting to services like VirusTotal API or AlienVault OTX, enabling a feedback mechanism where detected threats are anonymously shared to improve collective security. The current network monitoring could be enhanced with deep packet inspection capabilities that analyze traffic content and patterns from suspicious processes to detect data exfiltration with greater precision, while integration with packet capture mechanisms could identify encrypted keystroke data being transmitted to command-and-control servers. Adapting the detection framework for Android-based systems and IoT devices would extend protection to Linux-powered mobile devices and embedded systems, with Android's underlying Linux kernel allowing core detection principles to be applied where malicious keyboard apps pose significant threats. Future versions could implement intelligent automated response mechanisms beyond simple process termination, including automatically isolating compromised systems from networks, creating forensic snapshots for analysis, and implementing honeypot techniques to trap malware, while integration with SOAR platforms would enable enterprise-scale deployment with centralized management. The reporting capabilities could be expanded to include graphical visualizations of threat timelines and security trends through interactive dashboards showing threat score distributions and detection patterns, while integration with SIEM systems like ELK Stack or Splunk would enable professional-grade security monitoring. Implementing an integrated sandboxing environment would allow suspicious processes to be executed in isolation for behavioral analysis without risking system compromise, enabling detailed telemetry collection about malware behavior using containerization technologies like Docker. Extending the system to support Windows and macOS would provide unified security monitoring across heterogeneous enterprise environments, proving valuable for organizations managing mixed operating systems and requiring consistent security postures. Developing companion browser extensions would extend protection to web browsing contexts where modern keyloggers operate through malicious JavaScript, monitoring for suspicious DOM manipulation and unauthorized clipboard access to provide application-level protection complementing the existing system-level monitoring in the Keylogger Detection and Termination System.

6.3 Limitations

1. The system can be bypassed by advanced, polymorphic, or kernel-level keyloggers that use sophisticated evasion techniques.

2. Heuristic-based detection may produce false positives by misidentifying legitimate tools that mimic keylogger-like behavior.

3. Continuous real-time monitoring can cause noticeable performance slowdowns on older or resource-limited systems.

4. The system requires elevated privileges for full functionality, making deployment difficult in environments with strict access controls.

5. The system cannot detect physical hardware keyloggers, limiting its scope to software-based threats only.
6. The system’s reliability depends on the integrity of the underlying operating system, making it ineffective if the OS itself is compromised.

References

1. D. Elelegwu, L. Chen, Y. Ji and J. Kim, "A Novel Approach to Detecting and Mitigating Keyloggers," SoutheastCon 2024, Atlanta, GA, USA, 2024, pp. 1583-1590, doi: 10.1109/SoutheastCon52093.2024.10500122.

2. P. V, "Beyond Traditional Keyloggers: Developing and Detecting Advanced Keystroke Monitoring Systems," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-6, doi: 10.1109/CSITSS60515.2023.10334216.

3. A. Ankit, S. Inder, A. Sharma, R. Johari and D. P. Vidyarthi, "Simulating Cyber Attacks and Designing Malware using Python," 2023 10th International Conference on Signal Processing and Integrated Networks (SPIN), Noida, India, 2023, pp. 473-478, doi: 10.1109/SPIN57001.2023.10116554.

4. J. Sabu, A. S, A. Gopan, G. S and S. Murali, "Advanced Keylogger with Keystroke Dynamics," 2023 International Conference on Inventive Computation Technologies (ICICT), Lalitpur, Nepal, 2023, pp. 1598-1603, doi: 10.1109/ICICT57646.2023.10134044.

5. N. T. Singh, A. Shukla, A. Nagar, K. Arya, A. Tiwari and Y. Varun, "Keylogger Development: Technical Aspects, Ethical Considerations, and Mitigation Strategies," 2023 International Conference on Energy, Materials and Communication Engineering (ICEMCE), Madurai, India, 2023, pp. 1-5, doi: 10.1109/ICEMCE57940.2023.10434134.

6. S. Ortolani, C. Giuffrida and B. Crispo, "Unprivileged Black-Box Detection of User-Space Keyloggers," in IEEE Transactions on Dependable and Secure Computing, vol. 10, no. 1, pp. 40-52, Jan.-Feb. 2013, doi: 10.1109/TDSC.2012.76.

7. A. Moser, C. Kruegel and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," 2007 IEEE Symposium on Security and Privacy (SP '07), Berkeley, CA, USA, 2007, pp. 231-245, doi: 10.1109/SP.2007.17.

8. K. A. Rahman, K. S. Balagani and V. V. Phoha, "Snoop-Forge-Replay Attacks on Continuous Verification With Keystrokes," in IEEE Transactions on Information Forensics and Security, vol. 8, no. 3, pp. 528-541, March 2013, doi: 10.1109/TIFS.2013.2244091.

9. D. Javaheri, M. Hosseinzadeh and A. M. Rahmani, "Detection and Elimination of Spyware and Ransomware by Intercepting Kernel-Level System Routines," in IEEE Access, vol. 6, pp. 78321-78332, 2018, doi: 10.1109/ACCESS.2018.2884964.

10. H. M. Salih and M. S. Mohammed, "Spyware Injection in Android using Fake Application," 2020 International Conference on Computer Science and Software Engineering (CSASE), Duhok, Iraq, 2020, pp. 100-105, doi: 10.1109/CSASE48920.2020.9142101.

11. M. Shafi, R. K. Jha and S. Jain, "Behavioral Model for Live Detection of Apps Based Attack," in IEEE Transactions on Computational Social Systems, vol. 10, no. 3, pp. 934-946, June 2023, doi: 10.1109/TCSS.2022.3166145.

12. K. Onarlioglu, W. Robertson and E. Kirda, "Overhaul: Input-Driven Access Control for Better Privacy on Traditional Operating Systems," 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, 2016, pp. 443-454, doi: 10.1109/DSN.2016.47.

Appendix

Application Structure

```
KDTS
├── pyproject.toml
├── README.md
├── requirements.txt
├── uv.lock
├── Wayland_main.py
└── X11_main.py
```

Source Code

X11_main.py

```
import os

import sys

import psutil

import tkinter as tk

from tkinter import messagebox, filedialog

from tkinter import ttk

import threading

import time

from datetime import datetime

import subprocess

import pwd

import grp



class LinuxKeyloggerDetector:
```

```
def __init__(self):
    self.suspicious_keywords = [
        "keylog",
        "keycap",
        "keystroke",
        "keyrecord",
        "logkeys",
        "keysniff",
        "keymonitor",
        "inputlog",
        "screenlog",
        "spyware",
        "logger",
        "capture",
        "xinput",
        "xev",
        "xdotool",
        "hook",
    ]
    self.suspicious_paths = [
        "/tmp/",
        "/var/tmp/",
        "/dev/shm/",
        "/.",
        "/home/.*/.cache",
```

```
        "/home/.*/tmp",
        "/run/user",
    ]
    self.xorg_monitors = ["xinput", "xev", "xdotool", "xprop"]
    self.monitoring = False

def normalize_cmdline(self, cmdline):
    if cmdline is None:
        return []

    if isinstance(cmdline, str):
        # Convert string to list
        return cmdline.strip().split()

    if isinstance(cmdline, (list, tuple)):
        # Ensure they're strings
        return [str(x) for x in cmdline]

    # Unexpected type -> return empty
    return []

def calculate_threat_score(self, proc_info):
    """Calculate threat score based on multiple factors"""
    score = 0
    name = proc_info.get("name", "").lower()
```

```
exe_path = (proc_info.get("exe") or "").lower()
# cmdline = " ".join(proc_info.get("cmdline", [])).lower()
# cmdline = " ".join(self.normalize_cmdline(proc_info.get("cmdline"))).lower()
cmdline = " ".join(self.normalize_cmdline(proc_info.get("cmdline"))).lower()

# Check for suspicious keywords (30 points)
for keyword in self.suspicious_keywords:
    if keyword in name or keyword in exe_path or keyword in cmdline:
        score += 30
        break

# Check for suspicious paths (25 points)
for path in self.suspicious_paths:
    if path in exe_path:
        score += 25
        break

# Check if running from hidden directory (20 points)
if (
    "/." in exe_path
    or exe_path.startswith("/tmp")
    or exe_path.startswith("/dev/shm")
):
    score += 20
```

```
# Check for X11 input monitoring tools (25 points)
if any(tool in name for tool in self.xorg_monitors):
    if "test" not in cmdline and "debug" not in cmdline:
        score += 25

# Check for processes reading /dev/input (30 points)
try:
    proc = psutil.Process(proc_info["pid"])
    for file in proc.open_files():
        if "/dev/input" in file.path:
            score += 30
            break
except:
    pass

# Check CPU usage (10 points if very low - keyloggers are stealthy)
try:
    proc = psutil.Process(proc_info["pid"])
    cpu_percent = proc.cpu_percent(interval=0.1)
    if cpu_percent < 0.5 and cpu_percent > 0:
        score += 10
except:
    pass

# Check if running as root but started from user context (15 points)
```

```
    try:
        if proc_info.get("username") == "root" and os.getuid() != 0:
            score += 15
    except:
        pass

    return min(score, 100)

def check_systemd_services(self):
    """Check for suspicious systemd services"""
    suspicious = []
    try:
        result = subprocess.run(
            ["systemctl", "list-units", "--type=service", "--all", "--no-pager"],
            capture_output=True,
            text=True,
            timeout=5,
        )
        services = result.stdout.split("\n")

        for service in services:
            service_lower = service.lower()
            if any(
                keyword in service_lower for keyword in self.suspicious_keywords
            ):
```

```
            parts = service.split()
            if parts:
                suspicious.append(
                    {
                        "name": parts[0],
                        "status": "active"
                        if "active" in service_lower
                        else "inactive",
                        "description": " ".join(parts[4:])
                        if len(parts) > 4
                        else "N/A",
                    }
                )
    except Exception as e:
        print(f"Error checking systemd services: {e}")

    return suspicious

def check_startup_entries(self):
    """Check various Linux startup locations"""
    suspicious = []

    # Check user autostart
    autostart_paths = [
        os.path.expanduser("~/.config/autostart/"),
```

```
    "/etc/xdg/autostart/",
    os.path.expanduser("~/.config/systemd/user/"),
    "/etc/systemd/system/",
]

for path in autostart_paths:
    if not os.path.exists(path):
        continue

    try:
        for filename in os.listdir(path):
            filepath = os.path.join(path, filename)
            if os.path.isfile(filepath):
                with open(filepath, "r", errors="ignore") as f:
                    content = f.read().lower()

                if any(
                    keyword in content or keyword in filename.lower()
                    for keyword in self.suspicious_keywords
                ):
                    suspicious.append(
                        {
                            "name": filename,
                            "path": filepath,
                            "location": path,
```

```
                    "threat": "High",
                }
            )
    except Exception as e:
        continue

# Check crontab
try:
    result = subprocess.run(
        ["crontab", "-l"], capture_output=True, text=True, timeout=2
    )
    if result.returncode == 0:
        for line in result.stdout.split("\n"):
            line_lower = line.lower()
            if any(
                keyword in line_lower for keyword in self.suspicious_keywords
            ):
                suspicious.append(
                    {
                        "name": "Crontab Entry",
                        "path": line.strip(),
                        "location": "User Crontab",
                        "threat": "High",
                    }
                )
```

```
        except:
            pass

        return suspicious

    def check_network_connections(self):
        """Check for suspicious network activity"""
        suspicious_connections = []
        for conn in psutil.net_connections(kind="inet"):
            if conn.status == "ESTABLISHED":
                try:
                    proc = psutil.Process(conn.pid)
                    proc_name = proc.name().lower()
                    if any(
                        keyword in proc_name for keyword in self.suspicious_keywords
                    ):
                        suspicious_connections.append(
                            {
                                "pid": conn.pid,
                                "name": proc_name,
                                "local": f"{conn.laddr.ip}:{conn.laddr.port}",
                                "remote": f"{conn.raddr.ip}:{conn.raddr.port}"
                                if conn.raddr
                                else "N/A",
                            }
```

```
                )
            except:
                continue
        return suspicious_connections
    def check_input_devices(self):
        """Check for processes accessing input devices"""
        suspicious = []
        try:
            for proc in psutil.process_iter(["pid", "name", "exe"]):
                try:
                    for file in proc.open_files():
                        if "/dev/input" in file.path or "/dev/uinput" in file.path:
                            suspicious.append(
                                {
                                    "pid": proc.info["pid"],
                                    "name": proc.info["name"],
                                    "device": file.path,
                                    "exe": proc.info["exe"],
                                }
                            )
                except (psutil.AccessDenied, psutil.NoSuchProcess):
                    continue
        except:
            pass
        return suspicious
```

```
class ModernLinuxKeyloggerGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Keylogger Detection and Termination System")
        self.root.geometry("1920x1080")
        self.detector = LinuxKeyloggerDetector()
        self.monitoring_thread = None

        self.setup_styles()
        self.create_gui()

        self.suspicious_startup = []
        self.suspicious_procs = []
        self.suspicious_connections = []
        self.suspicious_services = []
        # self.input_devices = []

    def setup_styles(self):
        """Setup modern color scheme and styles"""
        self.colors = {
            "bg": "#ffffff",
            "fg": "#2c3e50",
            "primary": "#3498db",
```

```
    "secondary": "#e74c3c",
    "success": "#27ae60",
    "warning": "#f39c12",
    "danger": "#e74c3c",
    "surface": "#f8f9fa",
    "surface_light": "#e9ecef",
    "accent": "#9b59b6",
}

style = ttk.Style()
style.theme_use("clam")

# Configure colors
style.configure("TFrame", background=self.colors["bg"])
style.configure(
    "TLabel",
    background=self.colors["bg"],
    foreground=self.colors["fg"],
    font=("Ubuntu", 16),
)
style.configure(
    "Title.TLabel",
    font=("Ubuntu", 18, "bold"),
    foreground=self.colors["primary"],
)
```

```
style.configure(
    "Header.TLabel",
    font=("Ubuntu", 16, "bold"),
    foreground=self.colors["primary"],
)

# Button styles
style.configure(
    "Primary.TButton",
    background=self.colors["primary"],
    foreground="#ffffff",
    font=("Ubuntu", 14, "bold"),
    borderwidth=0,
    focuscolor="none",
    padding=10,
)
style.map("Primary.TButton", background=[("active", "#2980b9")])

style.configure(
    "Danger.TButton",
    background=self.colors["danger"],
    foreground="#ffffff",
    font=("Hack", 14, "bold"),
    borderwidth=0,
    padding=10,
```

```
)
style.map("Danger.TButton", background=[("active", "#c0392b")])

# Notebook style
style.configure("TNotebook", background=self.colors["bg"], borderwidth=0)
style.configure(
    "TNotebook.Tab",
    background=self.colors["surface"],
    foreground=self.colors["fg"],
    padding=[20, 10],
    font=("Ubuntu", 16, "bold"),
)
style.map(
    "TNotebook.Tab",
    background=[("selected", self.colors["surface_light"])],
    foreground=[("selected", self.colors["primary"])],
)

# Treeview styles
style.configure(
    "Treeview",
    background=self.colors["surface"],
    foreground=self.colors["fg"],
    fieldbackground=self.colors["surface"],
    borderwidth=0,
```

```
            font=("Ubuntu", 16),
            rowheight=25,
        )
        style.configure(
            "Treeview.Heading",
            background=self.colors["surface_light"],
            foreground=self.colors["primary"],
            font=("Ubuntu", 16, "bold"),
            # font=("DejaVu Sans", 11, "bold")
            borderwidth=1,
            relief="flat",
        )
        style.map(
            "Treeview",
            background=[("selected", self.colors["primary"])],
            foreground=[("selected", "#ffffff")],
        )

        self.root.configure(bg=self.colors["bg"])

    def create_gui(self):
        """Create modern GUI layout"""
        # Header
        header_frame = ttk.Frame(self.root)
        header_frame.pack(fill="x", padx=20, pady=(20, 10))
```

```
title_label = ttk.Label(
    header_frame,
    text="Keylogger Detection and Termination System",
    style="Title.TLabel",
)
title_label.pack(side="left")

self.status_label = ttk.Label(
    header_frame,
    text="● System Ready",
    foreground=self.colors["success"],
    font=("Hack", 18, "bold"),
)
self.status_label.pack(side="right")

# Main container with tabs
self.notebook = ttk.Notebook(self.root)
self.notebook.pack(fill="both", expand=True, padx=20, pady=10)

# Tab 1: Processes
proc_frame = ttk.Frame(self.notebook)
self.notebook.add(proc_frame, text="  Suspicious Processes  ")
self.create_process_tab(proc_frame)
```

```
    # Tab 2: Startup
    startup_frame = ttk.Frame(self.notebook)
    self.notebook.add(startup_frame, text="  Startup & Services  ")
    self.create_startup_tab(startup_frame)

    # Tab 3: Network
    network_frame = ttk.Frame(self.notebook)
    self.notebook.add(network_frame, text="  Network Activity  ")
    self.create_network_tab(network_frame)

    # Tab 4: Input Devices
    # input_frame = ttk.Frame(self.notebook)
    # self.notebook.add(input_frame, text="  Input Device Access  ")
    # self.create_input_tab(input_frame)

    # Tab 5: Real-time Monitor
    monitor_frame = ttk.Frame(self.notebook)
    self.notebook.add(monitor_frame, text="  Real-Time Monitor  ")
    self.create_monitor_tab(monitor_frame)

    # Control panel
    self.create_control_panel()

def create_process_tab(self, parent):
    info_label = ttk.Label(
```

```
    parent,
    text="Detected processes with suspicious behavior patterns",
    style="Header.TLabel",
)
info_label.pack(anchor="w", padx=10, pady=5)

columns = ("PID", "Name", "User", "Threat", "Score", "Command")
self.proc_tree = ttk.Treeview(
    parent, columns=columns, show="headings", height=18
)

self.proc_tree.heading("PID", text="PID")
self.proc_tree.heading("Name", text="Process")
self.proc_tree.heading("User", text="User")
self.proc_tree.heading("Threat", text="Threat")
self.proc_tree.heading("Score", text="Score")
self.proc_tree.heading("Command", text="Command Line")

self.proc_tree.column("PID", width=70)
self.proc_tree.column("Name", width=150)
self.proc_tree.column("User", width=100)
self.proc_tree.column("Threat", width=90)
self.proc_tree.column("Score", width=70)
self.proc_tree.column("Command", width=500)
```

```
    scrollbar = ttk.Scrollbar(
        parent, orient="vertical", command=self.proc_tree.yview
    )
    self.proc_tree.configure(yscrollcommand=scrollbar.set)

    self.proc_tree.pack(side="left", fill="both", expand=True, padx=(10, 0), pady=5)
    scrollbar.pack(side="right", fill="y", padx=(0, 10), pady=5)

def create_startup_tab(self, parent):
    info_label = ttk.Label(
        parent,
        text="Suspicious autostart entries and systemd services",
        style="Header.TLabel",
    )
    info_label.pack(anchor="w", padx=10, pady=5)

    columns = ("Type", "Name", "Status", "Path")
    self.startup_tree = ttk.Treeview(
        parent, columns=columns, show="headings", height=18
    )

    self.startup_tree.heading("Type", text="Type")
    self.startup_tree.heading("Name", text="Entry Name")
    self.startup_tree.heading("Status", text="Status")
    self.startup_tree.heading("Path", text="Path/Description")
```

```
        self.startup_tree.column("Type", width=120)
        self.startup_tree.column("Name", width=200)
        self.startup_tree.column("Status", width=100)
        self.startup_tree.column("Path", width=600)

        scrollbar = ttk.Scrollbar(
            parent, orient="vertical", command=self.startup_tree.yview
        )
        self.startup_tree.configure(yscrollcommand=scrollbar.set)

        self.startup_tree.pack(
            side="left", fill="both", expand=True, padx=(10, 0), pady=5
        )
        scrollbar.pack(side="right", fill="y", padx=(0, 10), pady=5)

    def create_network_tab(self, parent):
        info_label = ttk.Label(
            parent,
            text="Active network connections from suspicious processes",
            style="Header.TLabel",
        )
        info_label.pack(anchor="w", padx=10, pady=5)

        columns = ("PID", "Process", "Local", "Remote")
```

```
    self.network_tree = ttk.Treeview(
        parent, columns=columns, show="headings", height=18
    )

    self.network_tree.heading("PID", text="PID")
    self.network_tree.heading("Process", text="Process Name")
    self.network_tree.heading("Local", text="Local Address:Port")
    self.network_tree.heading("Remote", text="Remote Address:Port")

    self.network_tree.column("PID", width=80)
    self.network_tree.column("Process", width=250)
    self.network_tree.column("Local", width=300)
    self.network_tree.column("Remote", width=300)

    scrollbar = ttk.Scrollbar(
        parent, orient="vertical", command=self.network_tree.yview
    )
    self.network_tree.configure(yscrollcommand=scrollbar.set)

    self.network_tree.pack(
        side="left", fill="both", expand=True, padx=(10, 0), pady=5
    )
    scrollbar.pack(side="right", fill="y", padx=(0, 10), pady=5)

def create_input_tab(self, parent):
```

```
info_label = ttk.Label(
    parent,
    text="Processes accessing keyboard/input devices (/dev/input)",
    style="Header.TLabel",
)
info_label.pack(anchor="w", padx=10, pady=5)

columns = ("PID", "Process", "Device", "Executable")
self.input_tree = ttk.Treeview(
    parent, columns=columns, show="headings", height=18
)

self.input_tree.heading("PID", text="PID")
self.input_tree.heading("Process", text="Process Name")
self.input_tree.heading("Device", text="Device Path")
self.input_tree.heading("Executable", text="Executable Path")

self.input_tree.column("PID", width=80)
self.input_tree.column("Process", width=200)
self.input_tree.column("Device", width=250)
self.input_tree.column("Executable", width=500)

scrollbar = ttk.Scrollbar(
    parent, orient="vertical", command=self.input_tree.yview
)
```

```
    self.input_tree.configure(yscrollcommand=scrollbar.set)

    self.input_tree.pack(
        side="left", fill="both", expand=True, padx=(10, 0), pady=5
    )
    scrollbar.pack(side="right", fill="y", padx=(0, 10), pady=5)

def create_monitor_tab(self, parent):
    info_label = ttk.Label(
        parent,
        text="Real-time monitoring of suspicious system activity",
        style="Header.TLabel",
    )
    info_label.pack(anchor="w", padx=10, pady=5)

    self.monitor_text = tk.Text(
        parent,
        bg=self.colors["surface"],
        fg=self.colors["fg"],
        font=("Ubuntu Mono", 16),
        height=25,
        wrap="word",
        insertbackground=self.colors["primary"],
        relief="solid",
        borderwidth=1,
```

```
    )
    self.monitor_text.pack(fill="both", expand=True, padx=10, pady=5)

    scrollbar = ttk.Scrollbar(
        parent, orient="vertical", command=self.monitor_text.yview
    )
    self.monitor_text.configure(yscrollcommand=scrollbar.set)

    # Configure text tags for colored output
    self.monitor_text.tag_config("alert", foreground=self.colors["danger"])
    self.monitor_text.tag_config("warning", foreground=self.colors["warning"])
    self.monitor_text.tag_config("info", foreground=self.colors["primary"])
    self.monitor_text.tag_config("success", foreground=self.colors["success"])

def create_control_panel(self):
    control_frame = ttk.Frame(self.root)
    control_frame.pack(fill="x", padx=20, pady=(0, 20))

    self.scan_btn = ttk.Button(
        control_frame,
        text="Full System Scan",
        command=self.full_scan,
        style="Primary.TButton",
    )
    self.scan_btn.pack(side="left", padx=5)
```

```
self.terminate_btn = ttk.Button(
    control_frame,
    text="Terminate Process",
    command=self.terminate_selected,
    style="Danger.TButton",
)
self.terminate_btn.pack(side="left", padx=5)

self.monitor_btn = ttk.Button(
    control_frame,
    text="Start Monitoring",
    command=self.toggle_monitoring,
    style="Primary.TButton",
)
self.monitor_btn.pack(side="left", padx=5)

self.save_btn = ttk.Button(
    control_frame,
    text="Save Report",
    command=self.save_report,
    style="Primary.TButton",
)
self.save_btn.pack(side="left", padx=5)
```

```
    self.threat_label = ttk.Label(
        control_frame,
        text="Threats: 0",
        font=("Ubuntu", 18, "bold"),
        foreground=self.colors["success"],
    )
    self.threat_label.pack(side="right", padx=10)

def full_scan(self):
    self.status_label.config(
        text="● Scanning System...", foreground=self.colors["warning"]
    )
    self.scan_btn.config(state="disabled")
    self.root.update()

    def scan_thread():
        # Clear previous results
        self.proc_tree.delete(*self.proc_tree.get_children())
        self.startup_tree.delete(*self.startup_tree.get_children())
        self.network_tree.delete(*self.network_tree.get_children())
        # self.input_tree.delete(*self.input_tree.get_children())

        self.root.after(
            0,
            lambda: self.log_monitor(
```

```
        "[SYSTEM] Starting full system scan...", "info"
    ),
)

# Scan processes
self.root.after(
    0,
    lambda: self.log_monitor(
        "[SCAN] Checking running processes...", "info"
    ),
)
self.suspicious_procs = self.detector.check_running_processes()
for proc in self.suspicious_procs:
    self.proc_tree.insert(
        "",
        "end",
        values=(
            proc["pid"],
            proc["name"],
            proc.get("username", "N/A"),
            proc["threat_level"],
            f"{proc['threat_score']}%",
            proc.get("cmdline_str", "N/A")[:100],
        ),
    )
```

```
# Scan startup
self.root.after(
    0,
    lambda: self.log_monitor("[SCAN] Checking startup entries...", "info"),
)
self.suspicious_startup = self.detector.check_startup_entries()
for entry in self.suspicious_startup:
    self.startup_tree.insert(
        "",
        "end",
        values=("Autostart", entry["name"], entry["threat"], entry["path"]),
    )

# Scan systemd services
self.root.after(
    0,
    lambda: self.log_monitor("[SCAN] Checking systemd services...", "info"),
)
self.suspicious_services = self.detector.check_systemd_services()
for service in self.suspicious_services:
    self.startup_tree.insert(
        "",
        "end",
        values=(
```

```
                "Service",
                service["name"],
                service["status"],
                service["description"],
            ),
        )

    # Scan network
    self.root.after(
        0,
        lambda: self.log_monitor(
            "[SCAN] Checking network connections...", "info"
        ),
    )
    self.suspicious_connections = self.detector.check_network_connections()
    for conn in self.suspicious_connections:
        self.network_tree.insert(
            "",
            "end",
            values=(conn["pid"], conn["name"], conn["local"], conn["remote"]),
        )
else:
    self.status_label.config(
        text="√ System Clean", foreground=self.colors["success"]
    )
```

```
        self.threat_label.config(
            text="Threats: 0", foreground=self.colors["success"]
        )
        self.log_monitor("[SUCCESS] Scan complete: No threats detected", "success")

def terminate_selected(self):
    selected = self.proc_tree.selection()
    if not selected:
        messagebox.showinfo(
            "No Selection",
            "Please select a process to terminate from the Processes tab.",
        )
        return

    for item in selected:
        values = self.proc_tree.item(item)["values"]
        pid = values[0]
        name = values[1]

            f.write("\n" + "=" * 90 + "\n")
            f.write("RECOMMENDATIONS:\n")
            f.write("-" * 90 + "\n")
            f.write(
                "1. Review all flagged processes carefully before taking action\n"
            )
```

```
            messagebox.showinfo("Success", f"Report saved to:\n{file_path}")
            self.log_monitor(f"[SUCCESS] Report saved to {file_path}", "success")
        except Exception as e:
            messagebox.showerror("Error", f"Could not save report: {str(e)}")
            self.log_monitor(f"[ERROR] Failed to save report: {str(e)}", "alert")



if __name__ == "__main__":
    # Check if running on Linux
    if sys.platform != "linux":
        print("This tool is designed for Linux systems only.")
        sys.exit(1)

    root = tk.Tk()
    app = ModernLinuxKeyloggerGUI(root)

    # Welcome message
    app.log_monitor("=" * 60, "info")
    app.log_monitor("Keylogger Detection and Termination System", "info")
    app.log_monitor("Protecting your system from unauthorized input monitoring", "info")
    app.log_monitor("=" * 60, "info")
    app.log_monitor(
        "[INFO] Ready to scan. Click 'Full System Scan' to begin.", "success"
    )
    root.mainloop()
```