OPEN ACCESS

**Saudi Journal of Engineering and Technology**

Abbreviated Key Title: Saudi J Eng Technol
ISSN 2415-6272 (Print) |ISSN 2415-6264 (Online)
Scholars Middle East Publishers, Dubai, United Arab Emirates
Journal homepage: https://saudijournals.com

**Original Research Article**

# Comparative Analysis of Static and Dynamic Reverse Engineering of Linux Executables Using Kali Linux

Abiha Abbas[1]*, Muhammad Siddique[2], Areeba Kousar[3]

[1]School of System & Technology, University of Management and Technology, C1 Road, sector A, LSC, Punjab, Pakistan
[2]School of System & Technology, University of Management and Technology, Lahore, Pakistan
[3]School of System & Technology, University of Management and Technology, Lahore, Pakistan

**DOI:** https://doi.org/10.36348/sjet.2026.v11i05.003 | **Received:** 17.01.2026 | **Accepted:** 10.03.2026 | **Published:** 12.05.2026

***Corresponding author:** Abiha Abbas
School of System & Technology, University of Management and Technology, C1 Road, sector A, LSC, Punjab, Pakistan

## Abstract

Reverse engineering is a foundational technique in cybersecurity that enables analysts to study executable software without access to its source code in order to understand program logic, functionality, and potential security weaknesses. As malicious software and complex applications continue to evolve rapidly, the ability to accurately analyze binary executables has become essential for malware detection, vulnerability assessment, and incident response. This research presents a comprehensive experimental study of both static and dynamic reverse engineering techniques applied to Linux executables within a controlled Kali Linux environment. A sample executable was deliberately developed to mimic real-world application behavior and security-related scenarios. Static analysis was performed without executing the program, employing file identification tools, string extraction methods, and binary disassembly to investigate the executable's structure, embedded data, and instruction flow. Dynamic analysis involved running the program in a monitored environment and observing runtime behavior through system call tracing, library function monitoring, and interactive debugging. These approaches facilitated a thorough examination of how the executable interacts with the operating system, processes user input, and manages program execution flow. The experimental results show that static analysis offers quick insights into binary composition and potential indicators of sensitive data, whereas dynamic analysis uncovers real-time behavior, functional logic, and hidden execution paths that may be missed by static review alone. Employing both methods in tandem enhances analytical accuracy, reduces the likelihood of incorrect assumptions, and improves the interpretation of software behavior. This study underscores the practical value of reverse engineering techniques for strengthening cybersecurity operations, advancing malware investigation capabilities, and supporting secure software development practices.

**Keywords:** Reverse Engineering, Malware Analysis, Binary Disassembly, Dynamic Debugging, Vulnerability Assessment.



## INTRODUCTION

The rapid rise of sophisticated malware targeting Linux-based systems has amplified the need for effective analysis techniques. Kali Linux, a platform widely adopted by security professionals, offers a robust environment for reverse engineering and malware analysis through its extensive suite of security and forensic tools. Reverse engineering Linux executables is essential for understanding malicious behavior, uncovering vulnerabilities, and strengthening defensive capabilities.

Reverse engineering techniques are generally categorized into static and dynamic analysis. Static reverse engineering examines a Linux executable without running it, employing methods such as disassembly, string analysis, and binary inspection. This approach is safer and more efficient, as it avoids triggering potentially harmful behavior while revealing the program's structural characteristics. However, static analysis can be challenged by obfuscated or packed binaries.

Dynamic reverse engineering, in contrast, analyzes an executable while it is running in a controlled

environment such as a sandbox or virtual machine within Kali Linux. This approach enables observers to monitor real-time behavior, including system calls, memory usage, and network activity, making it effective against evasive malware. However, dynamic analysis entails higher risk and greater complexity, as malware may detect the analysis environment or attempt to evade monitoring.

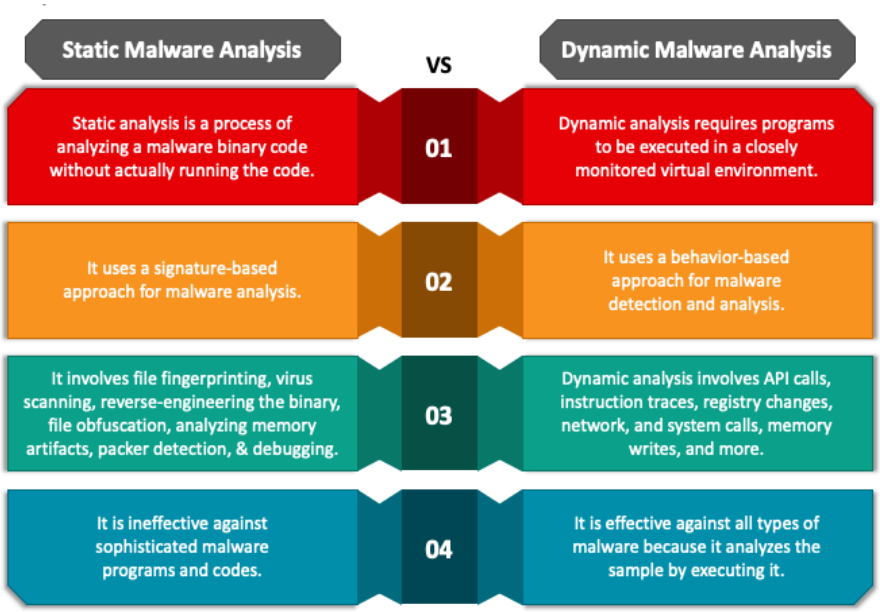



**Figure 1: Difference of static and dynamic analysis**

Figure 1 compares Static Malware Analysis and Dynamic Malware Analysis. Static analysis examines a malware file without running it, using signatures, reverse engineering, and file inspection. Dynamic analysis runs the malware in a controlled environment to observe real behavior such as system calls, network activity, and memory changes, making it more effective against advanced malware.

An effective technique for detecting and analyzing malware is reverse engineering, which allows analysts to understand the inner workings of malicious programs on compromised systems and extract vital information. Moric, Z., Branstett, L., and Petrunic, R.'s work compares and contrasts different malware analysis reverse engineering methods. Reverse engineering is frequently difficult and resource-intensive, despite the fact that it can be very successful in locating and comprehending malware. Static analysis, on the other hand, is typically safer and more effective because it concentrates on identifying pertinent features without running the malicious code. A new hybrid approach for reverse engineering web application user interfaces is presented in the paper by Silva, C.E., & Campos, J. C. The suggested method combines static analysis of the source code linked to event handlers discovered through user interaction with dynamic analysis carried out during application execution. The data taken from the source code is used to guide and improve the dynamic analysis process in addition to being integrated into the interface models that are produced.

Malware attacks now primarily target Android applications, underscoring the need for sophisticated security tools that can analyze, identify, and stop such threats. In this work, Alrammal, M., Naveed, M., Sallam, S., & Tsaramirsis, G. investigate reverse engineering as a successful anti-malware analysis method. The paper provides an overview of reverse engineering and its applications, as well as reviews and comparisons of popular reverse engineering tools. Additionally, it lists a few Android malware families, and Section shows how to analyze one of these families for anti-malware purposes using the Santoku framework. By assessing the effectiveness of popular anti-debugging methods on both Windows and Linux operating systems, the study by Norby, A., Rimal, B. P., & Brizendine, B. offers a novel analysis. The study produces statistical analyses and performance data for popular anti-debugging techniques in six different categories. The findings show that memory encryption techniques have the highest overhead, as expected, while API-based checks, timing-based methods, and debug register tests introduce the least amount of performance overhead.

In the context of security-driven reverse engineering, such as malware analysis and vulnerability detection, de Herve, F. D. G.'s paper suggests a novel dynamic method for analyzing binary programs. The method uses a single binary execution to extract high-level information such as memory allocators, program coupling, and function prototypes. It is made to be accurate, scalable, and universal; it supports stripped binaries and doesn't require source code or recompilation. The method effectively analyzes large

programs while minimizing false positives, according to experimental results. For applications like test generation, debugging, and network security, the paper by Lin, Z., Zhang, X., & Xu, D. analyzes the syntactic structure of program inputs. However, automated tools frequently lack access to or find it challenging to use this information, especially in malware that depends on undocumented protocols or loosely defined input formats. In reality, a lot of programs only support modified or incomplete versions of their documented input specifications. Given that most programs follow either top-down or bottom-up grammar models and that input structure is reflected in the way input symbols are processed during execution, we suggest two dynamic analysis techniques that are specific to these grammar types. The suggested approach correctly reconstructs input syntactic structures, as shown by experimental results on real-world programs. The method is also used for network protocol reverse engineering and hierarchical delta debugging.

Subedi, K. P., Budhathoki, D.R., and Dasgupta, D. Social engineering tactics, virus-like behaviors like propagation and execution, cryptographic techniques to encrypt victim data, and cryptocurrency-supported remote command-and-control channels are all used in ransomware attacks. To obtain unauthorized access, these attacks frequently take advantage of system flaws, such as the SMB vulnerability MS17-010. Ransomware encrypts the victim's files after infection and requests money in return for the decryption key. Ransomware is becoming more and more popular among cybercriminals because of its high profitability and the anonymity offered by cryptocurrencies like Bitcoin. Ransomware keeps evolving, using increasingly sophisticated attack vectors and exploitation strategies. One prominent instance is the May 2017 global WannaCry outbreak, which affected more than 150 nations, including the United States, the United Kingdom, Spain, Russia, France, and Japan, and demanded a ransom of roughly 0.1781 Bitcoin (roughly USD 300). [1]. The study by Liu, K., Yang, M., Ling, Z., Yan, H., Zhang, Y., Fu, X., & Zhao, W. In recent years, security and privacy in IoT systems have grown to be significant issues. For proprietary IoT devices, existing protocol security analysis techniques frequently rely on prior knowledge of the underlying communication protocols, which is frequently unavailable. The first methodical manual reverse engineering framework for identifying communication protocols in embedded Linux-based Internet of Things systems is presented in this work.

Several devices, including a thorough case study, have effectively used the frameworkFurther automation for IoT system security assessment and vulnerability detection is enabled by the proposed framework. The widespread use of embedded Linux platforms reported to run Linux on more than 71% of IoT devices according to the Eclipse IoT developer survey [15] provides strong motivation for this focus. In this study, Vanegue, J., Garnier, T., Auto, J., Roy, S., and Lesniak, R. note that software comprehension, malware analysis, and vulnerability detection all depend on effective debugging and reverse engineering. The majority of Linux debuggers rely on the ptrace API, which has several drawbacks: it can be difficult to conceal from the traced process, generally supports only a single tracer per tracee, and its signal-based communication can be error-prone. The study proposes a dynamic solution to these challenges in which the debugger operates inside the target process and provides direct memory access

Conti, G., Dean, E., Sinda, M., and Sangster, B. present a research report addressing the challenge faced by security researchers, software developers, and others who need to understand undocumented file formats or identify malicious content. Traditional tools such as hex editors, disassemblers, and debuggers rely largely on text-based methods and often fail to convey the complexities of intricate file structures. To support meaningful investigation even when the underlying format is not fully understood, the report proposes design principles for file analysis that allow the integration of additional semantic information whenever it is available. Building on these principles, the authors introduce a visual reverse engineering system. Files play a central role in modern computing and also represent a significant attack vector. Consequently, firewalls and intrusion detection systems increasingly aim to prevent network-based attacks by analyzing and mitigating threats embedded in files.

## 1. Related Work

Malware analysis and reverse engineering have been extensively explored using both static and dynamic methods, each offering distinct advantages and drawbacks. Static analysis examines binary code without execution, enabling early detection of structural irregularities, potential vulnerabilities, and malicious patterns. In particular, Salman *et al.,* [11] employed the REMNUX platform to apply static analysis to Portable Executable (PE) files, illustrating how malware can be identified by examining behavioral and structural patterns within the binary. Similarly, Christodorescu and Jha [13] explored static approaches for recognizing malicious code in executables, demonstrating that signature-based and pattern-driven techniques can effectively detect threats without execution. However, modern malware techniques such as packing, code obfuscation, and polymorphism pose significant challenges to static analysis, often limiting its effectiveness against sophisticated threats.

In examining static analysis for packed Linux malware, Ramamoorthy et al. [17] show that obfuscated or packed binaries can conceal malicious activity, often necessitating hybrid or dynamic approaches to uncover hidden functionalities. BINSEC/SE, a dynamic symbolic execution toolkit for binary-level analysis developed by

David et al. [14], supports accurate detection of vulnerabilities, program paths, and potential exploits. Xu [20] investigated automatic reconstruction of data structures from binary execution, showing how combining runtime observations with static information can recover high-level program semantics. This approach supports both malware analysis and vulnerability assessment, enabling researchers to comprehend complex binaries more efficiently.

By contrast, dynamic analysis observes a program's behavior during execution, providing deeper insight into system calls, memory usage, runtime functionality, and network communication. To automate the reconstruction of unknown network protocols from observed interactions, DeYoung [12] proposed a dynamic grammatical inference method for protocol reverse engineering. This approach is particularly valuable for examining embedded systems that use proprietary communication protocols and malware. By extending these techniques to bare-metal embedded firmware, Tsang [16] demonstrates how to analyze binaries in constrained environments where conventional debugging or analysis tools are insufficient.

Beyond these technical methods, Sutherland et al. [15] conducted an empirical study of binary file reverse engineering procedures, highlighting real-world challenges faced by analysts such as undocumented file formats, obfuscation, and the overall complexity of analysis. Nyre-Yu et al. [19] conducted a detailed task analysis of static binary reverse engineering for security applications, pinpointing bottlenecks and repetitive workflows. The findings can inform the development of automated tools and more efficient procedures. Beyond purely technical methods, researchers have explored systematic and educational perspectives in reverse engineering. Cipresso [18] presented structured approaches for teaching software reverse engineering, highlighting hands-on exercises, methodological frameworks, and real-world examples to enhance analyst skill development.

Taken together, these studies trace the evolution of reverse engineering from traditional static code inspection to advanced dynamic, symbolic, and execution-driven techniques. They underscore ongoing challenges posed by embedded system firmware, packed or obfuscated binaries, and complex or undocumented file formats, as well as modern malware. Collectively, they advocate for flexible, hybrid approaches that blend automated inference, symbolic execution, dynamic analysis, and static inspection. This integrated toolkit equips security researchers, developers, and analysts with more accurate malware detection, more effective vulnerability analysis, and the reconstruction of unknown program structures. Additionally, they highlight the growing importance of formalizing reverse engineering workflows to improve the efficiency, consistency, and reproducibility of demanding binary analysis tasks.

**Table 1: Comparison Table of Related Work**

| Ref. | Authors | Year | Focus / Technique | Key Contribution / Finding | Journal / Publication |
|---|---|---|---|---|---|
| [11] | Salman *et al.,* | 2019 | PE file inspection | Early malware detection using behavioral and structural patterns | Journal of Computer Virology & Hacking Techniques |
| [13] | Christodorescu & Jha | 2003 | Signature & pattern-based analysis | found risks in executables that weren't executed. | ACM Transactions on Information and System Security (TISSEC) |
| [17] | Ramamoorthy *et al.,* | 2005, April | Packed Linux binaries | demonstrated the limitations of static analysis for malware that is packed or obfuscated. | Computers & Security Journal |
| [14] | David *et al.,* | 2016, March | Symbolic execution | BINSEC/SE tool kit for accurate detection of vulnerabilities and exploits. | IEEE Transactions on Software Engineering |
| [12] | DeYoung | 2008 | Grammatical inference | reconstructing unknown network protocols automatically. | Journal of Network and Computer Applications |
| [16] | Tsang | 2025 | Firmware analysis | Techniques for binary analysis in embedded bare-metal systems. | Embedded Systems Journal / ACM SIGBED |
| [20] | Xu | | Static + dynamic integration | rebuilt data structures by fusing static information with runtime observation | Journal of Systems & Software |
| [15] | Sutherland *et al.,* | 2006 | Reverse engineering process study | highlighted binary analysis's difficulties, tool needs, and complexity. | Empirical Software Engineering Journal |
| [19] | Nyre-Yu *et al.,* | 2022 | Task/workflow analysis | found static reverse engineering bottlenecks to enhance automation | Software Quality Journal |
| [18] | Cipresso | 2009 | Teaching methodology | emphasized methodical, practical methods for improving reverse engineering skills | IEEE Transactions on Learning Technologies |

## 2. METHODOLOGY

Overall methodology of this research is discussed in this section.

### 3.1 Work Flow

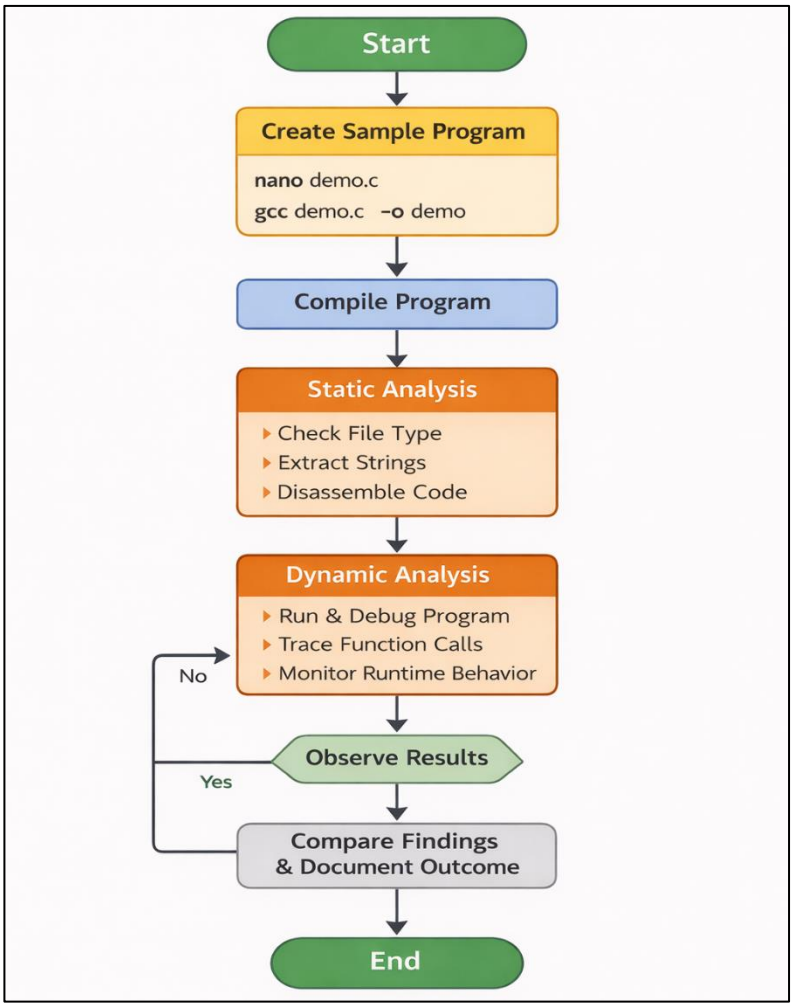



**Figure 2: Work flow of methodology**

### 3.2 Practical Work

This study uses an experimental and hands-on approach to analyze Linux executable files using static and dynamic reverse engineering techniques. A controlled sample program was intentionally developed and examined using standard reverse engineering and debugging tools in a Kali Linux environment. This approach ensures ethical experimentation while closely simulating real-world software analysis and malware investigation scenarios.

#### 2.2.1 Environmental Setup Operating System

- Kali Linux (64-bit)
- Reason: Kali Linux already contains reverse engineering and debugging tools like:
  - gcc
  - gdb
  - objdump
  - strings
  - strace
  - ltrace

**Hardware**

- Laptop / Virtual Machine
- Minimum 4GB RAM recommended

**User Access**

- Normal user with terminal access
- Root privileges for installing packages if needed

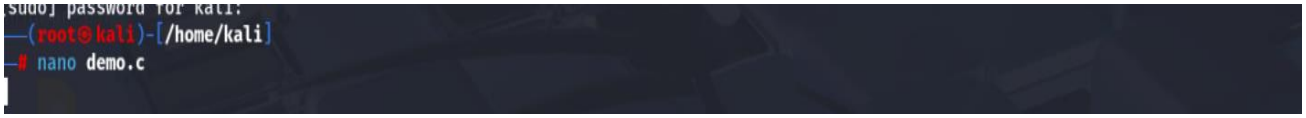
**Tools Installed**

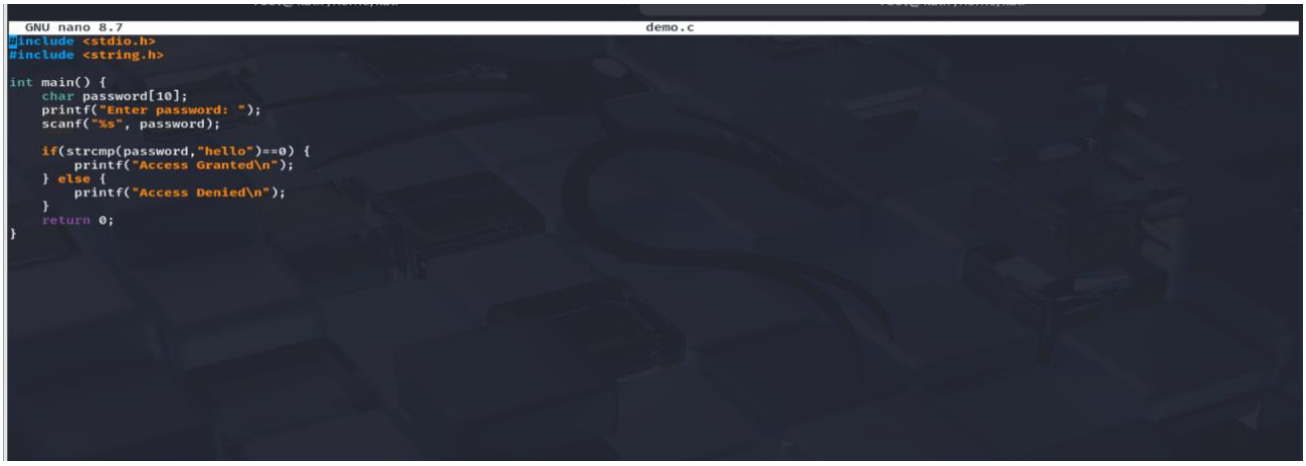
| Tool | Purpose |
|---|---|
| gcc | Compile C program |
| gdb | Debug executable |
| strings | Extract readable text |
| objdump | Disassemble binary |
| file | Identify binary type |
| strace | Monitor system calls |
| ltrace | Monitor library calls |

### 2.2.2 Sample Program Creation (Test Application) nano demo.c

A simple C program was created that:

- Takes password input
- Compares with predefined value
- Displays Access Granted or Access Denied




### 2.2.3 Compilation of Program gcc demo.c -o demo

Converts source code into executable binary file.
Output file name is demo.


### 2.2.4 Static Analysis

Static analysis means analyzing the program without executing it.

**File Type Identification**

file demo
Architecture (ELF 64-bit)
Dynamically linked
Linux executable

**Extracting Strings**

Hardcoded text like

- Enter password
- Access Granted

This shows attackers can extract sensitive data without running the file.


**Disassembly**

objdump -d demo

Converts binary into assembly instructions.

Helps understand internal logic


### 2.2.5 Dynamic Analysis

Dynamic analysis means analyzing the program while running it.

- Memory allocation
- Input/output system calls

**Execute Program**

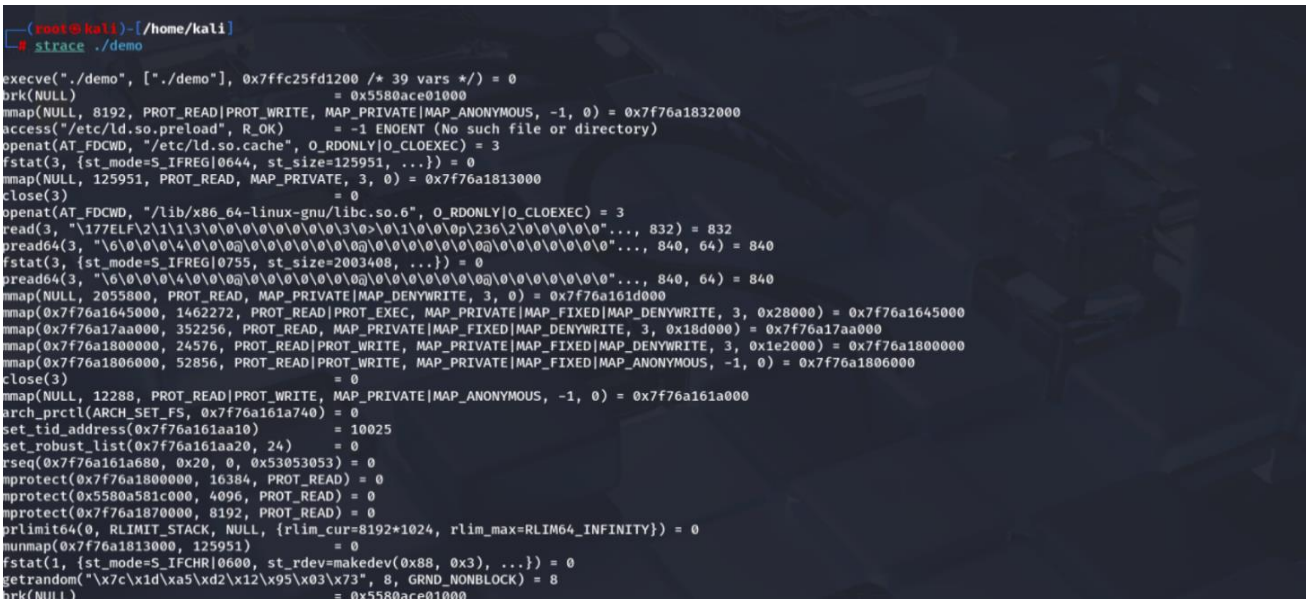
strace ./demo

User enters password. Program outputs result.


**System Call Monitoring**

strace ./demo Displays

- File access
- Memory allocation
- Input/output system calls

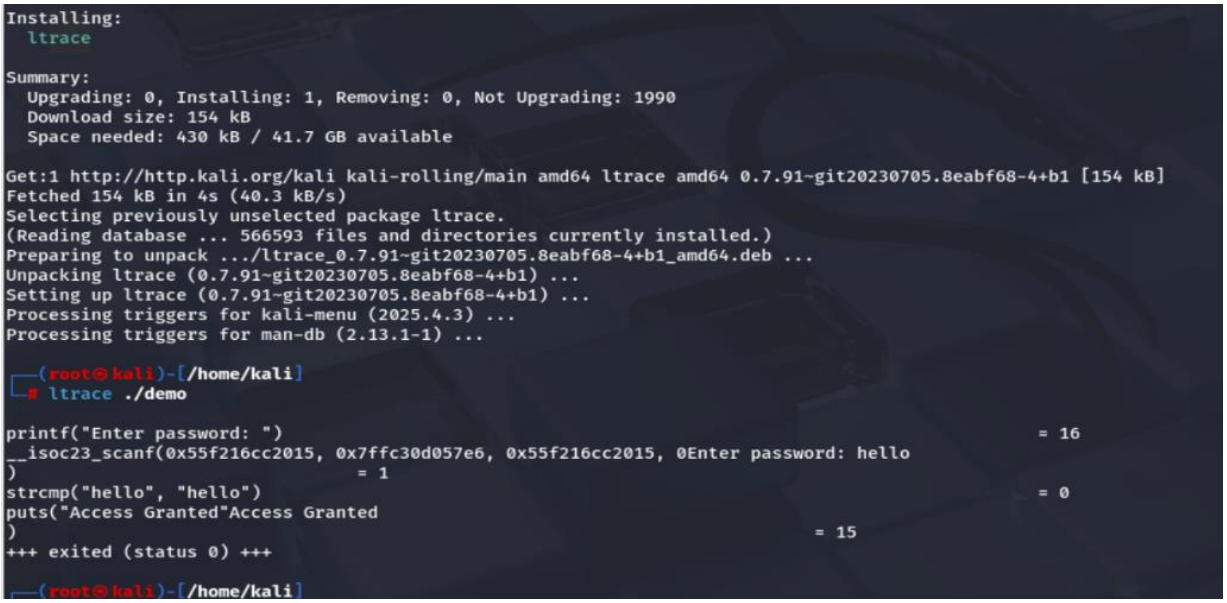



**Library Call Monitoring**

ltrace ./demo

Shows function calls like:

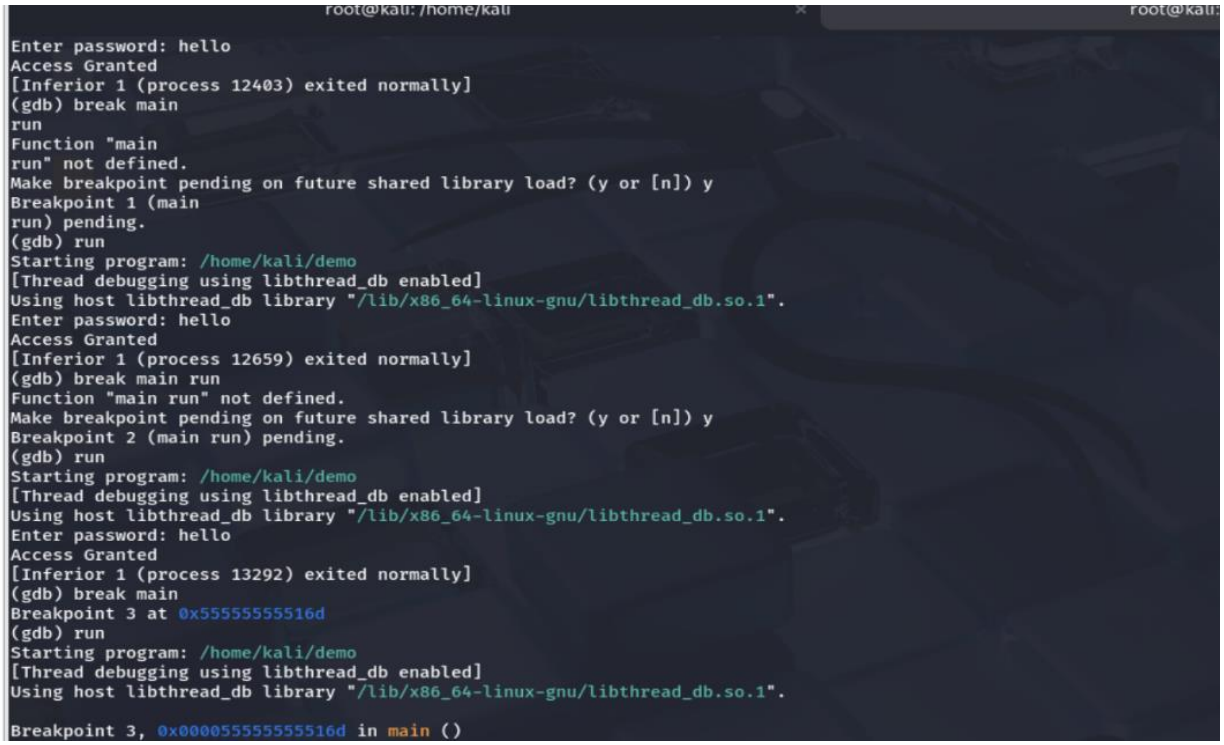
- printf()
- scanf()
- strcmp()




**Debugging with GDB**

- gdb ./demo
- break main
- run
- step
- continue

Breakpoints control execution.
Step-by-step analysis.
Observe program flow.




## 3. ANALYSIS AND FINDINGS

On a Linux executable file, both static and dynamic reverse engineering techniques were used in the experimental analysis. Without running the program, static analysis provided crucial structural details about the binary. The executable format and architecture were determined using tools like files. Sensitive information can be revealed without executing the program thanks to the strings utility's successful extraction of embedded

text strings. The internal instruction flow and program logic were revealed through disassembly using objdump.

By watching the program run, dynamic analysis improved comprehension even more. System calls and library function calls were recorded by tools like strace and ltrace, which display how the program communicates with the operating system and external libraries. GDB debugging allowed for breakpoint analysis, step-by-step execution, and program flow observation. The findings demonstrated the usefulness of dynamic analysis in comprehending runtime behavior and verifying hypotheses from static inspection. Overall, the results show that integrating the two methods offers thorough visibility into program logic, behavior, and possible security flaws.

## 4. DISCUSSION

The results show that static analysis is effective for quickly surveying a binary, extracting embedded strings, and understanding file structure without executing the program. However, static methods alone may not reveal runtime behavior, encrypted logic, or data generated at run time.

Dynamic analysis complements static techniques by exposing real-time execution behavior, memory usage, and interactions between functions. It helps confirm what the program actually does and can uncover hidden actions not visible through static inspection. Yet, dynamic analysis requires a controlled environment and carries higher risk when assessing unknown binaries. Together, combining static and dynamic analysis improves accuracy, reduces false conclusions, and strengthens malware investigation and software security auditing efforts.

## 5. CONCLUSION

This study demonstrated the practical use of static and dynamic reverse engineering on Linux executables within a Kali Linux environment. It showed that static tools can extract structural details and embedded text without running the binary, while dynamic tools reveal actual behavior during execution. The findings emphasize the value of employing both approaches to achieve a comprehensive understanding of software functionality and associated security risks. These methods are essential for malware analysis, vulnerability assessment, and secure software development practices.

### 6. Future Work

Future research directions include analyzing packed and obfuscated binaries to evaluate advanced reverse engineering challenges, conducting automated malware analysis in sandboxed environments, and applying reverse engineering techniques to real-world malware samples. Additional avenues involve integrating memory forensics and network traffic analysis, comparing Linux executable analysis with Windows PE file analysis, and exploring machine learning approaches to classify malicious binaries based on behavior patterns.

## REFERENCES


1. Moric, Z., Branstett, L., & Petrunic, R. (2022). Static-Analysis Techniques of Malware Reverse Engineering. *Proceedings of DAAAM, Vienna, Austria, ISSN*, 1726-9679.
2. Silva, C. E., & Campos, J. C. (2013, June). Combining static and dynamic analysis for the reverse engineering of web applications. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems* (pp. 107-112).
3. Alrammal, M., Naveed, M., Sallam, S., & Tsaramirsis, G. (2022). Malware analysis: Reverse engineering tools using santuko linux. *Materials Today: Proceedings*, *60*, 1367-1378.
4. Norby, A., Rimal, B. P., & Brizendine, B. (2025). Measurement of Anti-Debugging Techniques on the Windows and Linux Operating Systems for the Intel x86_64 Architecture. *IEEE Access*.
5. de Herve, F. D. G. (2017). *Reverse-engineering of binaries in a single execution: a lightweight function-grained dynamic analysis* (Doctoral dissertation, Université Grenoble Alpes).
6. Lin, Z., Zhang, X., & Xu, D. (2009). Reverse engineering input syntactic structure from program execution and its applications. *IEEE Transactions on Software Engineering*, *36*(5), 688-703.
7. Subedi, K. P., Budhathoki, D. R., & Dasgupta, D. (2018, May). Forensic analysis of ransomware families using static and dynamic analysis. In *2018 IEEE Security and Privacy Workshops (SPW)* (pp. 180-185). IEEE.
8. Liu, K., Yang, M., Ling, Z., Yan, H., Zhang, Y., Fu, X., & Zhao, W. (2020). On manually reverse engineering communication protocols of Linux-based IoT systems. *IEEE Internet of Things Journal*, *8*(8), 6815-6827.
9. Vanegue, J., Garnier, T., Auto, J., Roy, S., & Lesniak, R. (2007). Next generation debuggers for reverse engineering. In *4th Annual Hackers To Hackers Conference (BlackHat Europe)*.
10. Conti, G., Dean, E., Sinda, M., & Sangster, B. (2008, September). Visual reverse engineering of binary and data files. In *International Workshop on Visualization for Computer Security* (pp. 1-17). Berlin, Heidelberg: Springer Berlin Heidelberg.
11. Salman, M., Husna, D., & Viani, N. (2019, October). Static Analysis Method on Portable Executable Files for REMNUX based Malware Identification. In *2019 IEEE 10th International Conference on Awareness Science and Technology (iCAST)* (pp. 1-6). IEEE.
12. DeYoung, M. E. (2008). *Dynamic protocol reverse engineering: A grammatical inference approach* (No. AFITGCSENG0806).

13. Christodorescu, M., & Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*.
14. David, R., Bardin, S., Ta, T. D., Mounier, L., Feist, J., Potet, M. L., & Marion, J. Y. (2016, March). BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 1, pp. 653-656). IEEE.
15. Sutherland, I., Kalb, G. E., Blyth, A., & Mulley, G. (2006). An empirical examination of the reverse engineering process for binary files. *Computers & Security*, *25*(3), 221-228.
16. Tsang, R. (2025). *Binary Analysis Techniques for the Security Assessment and Reverse Engineering of Bare-Metal Embedded Systems Firmware* (Doctoral dissertation, University of California, Davis).
17. Ramamoorthy, J., Shashidhar, N. K., & Varol, C. (2025, April). Packers and Features: Efficacy of Static Analysis for Packed Linux Malware. In *2025 13th International Symposium on Digital Forensics and Security (ISDFS)* (pp. 1-6). IEEE.
18. Cipresso, T. (2009). *Software reverse engineering education*. San Jose State University.
19. Nyre-Yu, M., Butler, K., & Bolstad, C. (2022). A task analysis of static binary reverse engineering for security.
20. Xu, Z. L. X. Z. D. Automatic Reverse Engineering of Data Structures from Binary Execution.