Advanced Adaptive Techniques for Securing Applications Against Dynamic Reverse Engineering Attacks

Khair Eddin Sabri

Computer Science Department, King Abdullah II School of Information Technology, The University of Jordan, Amman, Jordan | Cyber Security Department, King Hussein School of Computing Sciences, Princess Sumaya University for Technology, Amman, Jordan
k.sabri@ju.edu.jo (corresponding author)

Received: 13 February 2026 | Revised: 13 March 2026 | Accepted: 24 March 2026

 DOI: https://doi.org/10.48084/etasr.18160

ABSTRACT

Dynamic reverse engineering analyzes a program during execution to understand its behavior and modify its logic to bypass authentication mechanisms or extract sensitive information, with such analysis commonly relying on debugging and runtime instrumentation tools. However, many protection techniques usually use a single anti-debugging or anti-instrumentation mechanism, which can be easily bypassed once attackers identify the detection method. To combat this limitation, this paper proposes an adaptive framework for Windows that detects dynamic program analysis by combining multiple indicators. Specifically, each indicator is assigned a weight that reflects its reliability in identifying active analysis, while a confidence score is calculated from these weights to classify the risk level as low, medium, or high. The program then dynamically adapts its behavior according to the classified risk level. The framework incorporates additional indicators to detect modern instrumentation tools, such as Frida, which can often bypass traditional anti-debugging techniques. The proposed framework was implemented in C and evaluated under multiple scenarios, with experimental results showing that the framework effectively detects dynamic analysis, while performance evaluation indicates low runtime overhead.

Keywords-reverse engineering; anti-debugging; anti-instrumentation; Frida; x86dbg

I. INTRODUCTION

Software reverse engineering poses a significant threat to executable programs, as it seeks to understand program behavior to extract sensitive information or to bypass authentication mechanisms by modifying functionality. Two main approaches are commonly used: static analysis and dynamic analysis. Static analysis examines the program code without execution, with tools such as Interactive Disassembler Professional (IDA) Pro and Ghidra being widely used to translate machine code into assembly language. In contrast, dynamic analysis observes program behavior during execution.

Although static analysis enables inspection of program code, it is often insufficient on its own, as developers can apply techniques such as obfuscation and encryption to conceal internal structure, which leads attackers to rely on dynamic analysis to examine runtime behavior. Therefore, debugging and instrumentation are the primary techniques used in this context. This is because debugging allows attackers to pause, resume, and step through execution at the assembly level, typically using breakpoints to control program flow. Tools such as x64dbg [1] are widely used for this purpose. On the other hand, instrumentation enables interception and modification of function calls through hooking techniques, with a common approach being inline hooking, in which a function prologue is overwritten with a jump instruction to redirect the control flow. Frida [2] is a well-known instrumentation tool that supports monitoring and interception of Application Programming Interface (API) calls, allowing observation of passed values.

Regarding dynamic analysis, most existing techniques for its detection rely on a single indicator, such as checking function prologues or tool signatures. Once attackers understand these techniques, they can bypass them relatively easily. Furthermore, many systems respond to analysis detection by immediately terminating execution, which can expose the protection mechanism and negatively affect program usability due to false positives.

Currently, most existing research focuses on detecting traditional debugging techniques, while modern dynamic instrumentation tools receive less attention. Additionally, prior work primarily studies these indicators on Linux.

In response to the abovementioned limitation, this paper presents an adaptive runtime behavior framework that combines anti-debugging and anti-instrumentation techniques. Instead of relying on a single detection indicator, the framework evaluates multiple indicators at runtime and adapts

program behavior based on the overall confidence score. The main contributions of this work can be summarized as:

- Designing an adaptive framework for Windows applications that integrates multiple runtime indicators to detect dynamic program analysis.
- Introduction of a confidence-based decision model that enables gradual responses instead of binary actions, thereby reducing false positives while maintaining effective protection.
- Implementation of the framework in C and validation through experimental evaluation, demonstrating effectiveness against multiple analysis scenarios, including debugging and Frida-based instrumentation.

II. LITERATURE REVIEW

The analysis of malware and malicious activity has received significant research attention for both detection and analysis purposes. Such analysis can be performed at different levels, including the network level, as in [3, 4], and the host level. At the host level, static and dynamic analysis are commonly used for malware investigation. For example, machine learning techniques combine static and dynamic features to analyze Android malware [5], while Long Short-Term Memory (LSTM) and Neural Network (NN) models have been applied to improve detection accuracy [6]. Similar approaches have also been used to analyze malware capable of launching Denial-of-Service (DoS) attacks [7].

While these studies focus on detecting malicious software, the same analysis techniques can be exploited by attackers to understand and compromise legitimate applications. This potential misuse of analysis tools motivates the development of effective techniques to prevent legitimate programs from being analyzed.

Several approaches are proposed to protect applications against reverse engineering. Some techniques focus on protecting against static analysis, while others target dynamic analysis. For static analysis, a Low Level Virtual Machine (LLVM) optimization sequence was proposed in [8] to produce highly obfuscated versions of the original code. In [9], Loki was introduced, a software obfuscation framework designed to resist automated deobfuscation attacks.

For dynamic analysis, authors in [10] evaluated commonly used anti-debugging techniques on Windows and Linux systems, providing both experimental results and performance analysis. In [11], an empirical study of 2,646 Android and iPhone Operating System (iOS) applications investigated the adoption of software hardening techniques, showing that many applications do not fully implement the proposed best practices. Moreover, an approach to protecting Java bytecode from malicious debugging was presented in [12], with mitigation responses assigned to individual debugging signals.

In [13], several indicators were presented for detecting the Frida instrumentation framework in Linux environments. While similar indicators are considered in this study, the objectives differ. Authors in [13] focused on Linux systems and on how malware can bypass detection mechanisms, whereas the present study targets the protection of legitimate Windows applications. Applying these indicators in Windows environments requires different implementation considerations due to the architectural characteristics of the Windows API. For example, detecting inline hooks cannot rely solely on checking the first bytes of a function for jump instructions, as legitimate Windows API calls may go through multiple layers of libraries that resemble instrumentation behavior.

In addition, authors in [14] improved dynamic analysis efficiency by implementing Frida bindings in Rust, treating Frida as a legitimate analysis tool. In contrast, the proposed framework considers Frida-based instrumentation a potential threat and focuses on detecting its presence during execution.

III. THREAT MODEL

The primary objective of this work is to increase the effort and complexity required to analyze program behavior at runtime, thereby discouraging reverse engineering attempts. Therefore, for the purposes of this study, it is assumed that attackers have full control over the executable files of the program and can perform dynamic analysis using debugging and instrumentation tools such as x64dbg and Frida. Moreover, static analysis is considered out of scope.

IV. ADAPTIVE DETECTION FRAMEWORK

The proposed framework consists of three main components. The first detects debugging activity, the second detects instrumentation, and the third calculates the confidence score from the results of the other two components.

A. Anti-Debugging

Anti-debugging techniques aim to detect whether a program is being executed under a debugger in order to prevent attackers from analyzing or modifying its code or data. The proposed framework uses the following indicators:

- Debugger-related indicators, including API-based detection and monitoring of debug registers.
- Process-related indicators, including identification of debugger processes and verification of the parent process.
- Timing-based indicators, including detection of abnormal execution delays.

1) API-Based Debugger Detection

The Windows APIs IsDebuggerPresent and CheckRemoteDebuggerPresent are standard methods for detecting if a debugger is attached to a process. The output of these APIs is a strong indicator that a program is being debugged.

2) Debug Register Inspection

Hardware breakpoints are implemented using Central Processing Unit (CPU) debug registers (DR0-DR7). The use of these registers is an indicator that a program is being debugged.

3) Process Detection

A practical approach to detecting debugging activity is to enumerate active processes and identify those commonly associated with debuggers, such as x64dbg.exe. Many

debuggers run as user-mode processes and remain active while the program is being analyzed, making them detectable through process enumeration.

The framework also compares the parent process against a predefined whitelist of expected system processes. Under normal conditions, programs are usually launched by standard system processes. For example, the parent process is explorer.exe when launched through the graphical user interface, and cmd.exe when launched from the command line. If the parent process does not match these expected cases, it may indicate that the program was started by a debugger, providing an additional debugging indicator.

4) Timing-Based Debugger Detection

Debugging activities, such as stepping, typically slow down program execution. By measuring the execution time of a specific sensitive function, an unusually long execution time may indicate the presence of debugging. The execution time of each monitored function is measured and averaged over multiple runs, including scenarios under high system load, to establish a baseline for normal behavior.

B. Anti-Instrumentation

Runtime instrumentation tools such as Frida enable dynamic analysis through API interception, which is a common technique used in dynamic instrumentation. The proposed framework uses the following indicators:

- Instrumentation and hooking indicators, including detection of function control-flow manipulation.
- Process-related indicators, including detection of Frida-related processes and verification of the parent process.
- Library-based indicators, including detection of injected Dynamic Link Libraries (DLLs) associated with Frida.

1) Inline API Hook Detection

Inline hooks work by modifying the first few instructions of a function to redirect execution to another code location, typically using jump or call instructions. To detect such modifications, the first few bytes of the imported functions are checked for unexpected jump or call instructions. This technique is effective because many hooking tools modify the beginning of functions to alter the execution flow.

To illustrate this mechanism, the CreateFileW API is used as a reference case. The framework reads the function's in-memory instructions and verifies whether execution is redirected to an external address. Since instrumentation frameworks commonly rely on such redirection for API interception, this serves as a strong indicator of runtime manipulation.

However, modern Windows systems introduce an important nuance: many APIs exposed via kernel32.dll are internally implemented in kernelbase.dll. Consequently, redirections between these modules are legitimate. To account for this behavior, the framework validates the destination address of detected jumps:

- If the target address lies within trusted system modules (kernel32.dll, kernelbase.dll), the redirection is considered benign.
- If the target lies outside these modules, it is flagged as a potential inline hook.

This approach reduces false positives while maintaining sensitivity to external instrumentation.

The inline hook detection mechanism does not require checking all imported functions of the program. Instead, the framework focuses on preselected critical APIs that can be determined by developers. Thus, selecting a subset of APIs keeps the runtime overhead low while still providing effective detection.

2) Process Enumeration

A commonly used technique for detecting Frida-based dynamic instrumentation is to enumerate active processes to identify Frida-associated names, such as frida-server, frida-helper, or frida-agent. Since Frida typically requires such processes to manage code injection and communication with the target program, their presence may indicate instrumentation. Furthermore, similar to the anti-debugging approach, the framework verifies the parent process of the program.

3) DLL Libraries

On Windows, Frida often operates by injecting a DLL (agent) into the target process. The injected library becomes part of the process and appears in its list of loaded modules. Consequently, Frida can be detected by enumerating the loaded modules and checking module names or file paths for known Frida-related DLLs, such as frida-agent or frida-gadget.

C. Adaptive Behavior

Relying on a single detection mechanism is insufficient against modern dynamic analysis tools, which can selectively bypass individual checks. To address this limitation, the proposed framework employs a multi-indicator adaptive decision model. Each indicator is assigned a weight reflecting its reliability and susceptibility to false positives. A confidence score is then computed by aggregating the weights of all triggered indicators:

$$Confidence\ Score = \sum_{i=1}^{n} w_i \cdot I_i \quad (1)$$

where w_i is the weight of the indicator i, and $I_i \in \{0,1\}$ denotes whether the indicator is triggered. The selection of weight values is based on the strength of evidence provided by each indicator and empirical evaluation across different execution scenarios. The goal of the weighting scheme is not to represent exact probabilities but to reflect the reliability of each indicator in contributing to the final decision. As shown in Table I:

- Indicators that provide direct and explicit evidence of dynamic analysis are assigned a weight of 5.
- Indicators that strongly signal dynamic activity but may also appear in legitimate scenarios are assigned a weight of 4.

- A lower weight of 2 is assigned to the timing-based indicator, as execution timing variations can occur due to normal system workload. Therefore, the timing indicator is treated as supporting evidence.

TABLE I. THE PROPOSED INDICATORS AND THEIR SCORES

Indicator	Description	Score
IsDebuggerPresent	User-mode debugger	5
Debug registers	Hardware breakpoints	4
Debugger process detected	Presence of debugger	4
Parent process	Suspicious launcher	4
Time	Slow execution	2
Inline hook	API interception	5
Frida process detected	External instrumentation	4
Frida module loaded	Injected agent	5

To balance detection sensitivity and usability, three confidence levels are defined:

- When the confidence score is below 5, the framework assumes that no strong analysis indicators are present, and the program is executing normally. This threshold corresponds to cases where only one weak indicator is triggered, which is insufficient to indicate active analysis.
- A confidence score between 5 and 9 indicates possible analysis activity. This range corresponds to the triggering of one strong indicator or two weaker indicators. In this case, the program may limit certain sensitive features or modify execution paths while less critical functions continue to operate normally. This design helps reduce false positives while still responding to potential threats.
- A confidence score of 10 or higher indicates strong evidence of active analysis. This corresponds to the presence of multiple strong indicators or a combination of strong and multiple weak indicators. In this case, the program may apply stronger protection measures, such as terminating execution.

These threshold values were validated through experimental evaluation across different execution scenarios to ensure balanced detection sensitivity and system usability. The selected thresholds can be adjusted depending on application security requirements and risk tolerance levels, providing flexible and context-aware protection.

V. EXPERIMENTAL EVALUATION AND VALIDATION

A. Experimental Setup

The proposed framework was evaluated using four programs of different sizes and complexity levels. These programs ranged from small proof-of-concept programs performing simple file creation to more complex applications incorporating file operations, network communication, and event interception using Windows hooks with several imported libraries. The properties of the evaluated programs are summarized in Table II.

The detection indicators used in the proposed framework depend mainly on runtime analysis behavior, such as the presence of a debugger, process injection, and API hooking, rather than the functionality of the programs. Therefore, increasing the number of evaluated programs does not significantly affect the detection results, since the same analysis tools produce similar runtime behavior regardless of the program being analyzed. For this reason, a representative set of programs with different levels of complexity was considered sufficient to validate the effectiveness of the proposed thresholds and the overall framework.

TABLE II. PROGRAMS USED IN EXPERIMENTAL EVALUATION

Program	Functionality	Monitored APIs	Imported DLL
P1	File Creation	1	kernel32.dll, msvcrt.dll
P2	File Create, Read, Write	3	kernel32.dll, msvcrt.dll
P3	File + Network	5	kernel32.dll, msvcrt.dll, wininet.dll
P4	File + Network + Windows hook	6	kernel32.dll, msvcrt.dll, wininet.dll, user32.dll

All experiments were conducted on a 64-bit Windows 10 equipped with an Intel Core i5-3230M processor running at 2.60 GHz, with 8 GB of Random Access Memory (RAM). The executable file was compiled using the GNU Compiler Collection (GCC) for Windows.

The implementation of the proposed dynamic analysis detection framework is publicly available at: https://github.com/sabrikm/Dynamic_Analysis_Detection.

B. Test Scenarios

To evaluate the effectiveness of the framework, each program was executed individually under three analysis scenarios, where each scenario was repeated three times:

- Normal execution without analysis tools.
- Debugging using x64dbg.
- Instrumentation using Frida.

The confidence scores for all evaluated programs are presented in Table III. The results show similar scores across different programs, indicating that the confidence score depends mainly on runtime analysis behavior rather than program functionality. Specifically, under normal execution without dynamic analysis tools, the calculated confidence score was zero, as expected. When the programs were executed under debugging with x64dbg, the calculated score increased to 15, representing strong evidence of dynamic analysis. This score resulted from debugger presence (5 points), detection of a debugger process (4 points), detection of a suspicious parent process (4 points), and increased execution time (2 points). In the instrumentation scenario using Frida, the calculated score increased to 18, indicating a high-confidence detection of dynamic analysis. This score resulted from inline hook detection (5 points), Frida process detection (4 points),

detection of injected Frida DLL modules (5 points), and detection of a suspicious parent process (4 points).

Programs P3 and P4, which included network communication, occasionally triggered the timing-based indicator because of network delays, increasing the confidence score by 2 points in some runs. Nevertheless, in all cases, the timing indicator did not alter the final classification outcome.

TABLE III. CONFIDENCE SCORE RESULTS OF THE PROGRAM UNDER DIFFERENT EXECUTION SCENARIOS

Program/Scenario	Scenario 1	Scenario 2	Scenario 3
P1	0	15	18
P2	0	15	18
P3	0-2	15	18-20
P4	0-2	15	18-20

C. Comparison with Single-Indicator Techniques

To further evaluate the effectiveness of the proposed multi-indicator approach, three scenarios were considered in which individual indicators may lead to false positives during normal execution:

- The first scenario occurs when only the timing indicator is used for detection. Programs such as P3 and P4 may have longer execution times due to network communication delays, which can trigger the timing indicator even during normal execution.
- The second scenario arises when detection relies only on the presence of a debugger or instrumentation processes. In practice, such tools may be running on the system for legitimate purposes unrelated to analyzing the evaluated program, which may lead to false positives.
- The third scenario involves using the parent process as an indicator. If the program is executed in a non-standard manner, such as through a script, the parent process may appear unusual and be incorrectly classified as suspicious.

These scenarios show that individual indicators may incorrectly classify normal execution as analysis. These cases are outlined in Table IV.

TABLE IV. SCENARIOS OF INCORRECT CLASSIFICATION USING SINGLE INDICATORS

Scenario	Indicator	Score	Expected Classification
Network delay	Time	2	Normal
Background tools running	Process detection	4	Normal
Script execution	Parent process	4	Normal

In addition, certain indicators, such as inline hooking detection, debugger presence, or detection of injected modules, represent strong indicators that are less likely to generate false positives when used independently. However, relying exclusively on a single indicator may still result in missed detections when the corresponding analysis technique is absent.

This is the main reason that the proposed framework combines multiple indicators with different weights, which helps reduce false positives caused by weak indicators while maintaining the ability to detect different analysis techniques.

D. Computational Complexity and Runtime Overhead

The proposed framework is designed to maintain low performance overhead while executing multiple runtime checks. Table V summarizes the execution time of individual indicators across the four evaluated programs (P1-P4), which differed in functionality and number of monitored APIs. The results show that most anti-debugging and anti-instrumentation indicators are lightweight and execute very quickly. For instance, simple checks, such as IsDebuggerPresent, debug register inspection, and timing measurements, operate in constant or near-constant time because they rely primarily on direct API calls.

Moreover, inline hook verification required approximately 4 ms when checking APIs within the same library, such as kernel32.dll, as observed in P1 and P2, but when additional libraries were introduced, as in P3 and P4, inline hook detection time increased to 7.8 ms and 9.82 ms, respectively. This increase reflects the additional overhead associated with resolving API addresses and validating targets across multiple imported DLLs. Furthermore, some indicators, such as detecting processes of analysis tools and identifying the parent process, lasted longer because they require scanning system process information. Overall, experimental results showed that executing all indicators required approximately 25 ms for smaller programs and up to 35 ms for larger programs under heavier system load. Thus, despite variations in program size and system activity, the overall overhead remained low and effectively unnoticeable to end users, supporting the suitability of the framework for real-world deployment.

TABLE V. MEASURED OVERHEAD (MS) OF INDIVIDUAL DETECTION INDICATORS FOR P1-P4 PROGRAMS, AVERAGED ACROSS 10 RUNS

Indicator / program	P1	P2	P3	P4
Time	0	0	0	0
IsDebuggerPresent	0.004	0.003	0.008	0.01
Debug registers	0.009	0.011	0.10	0.12
Debugger process detected	5.293	6.235	5.6	6.86
Parent process	10.15	10.868	10.951	11.322
Frida process detected	5.522	5.6	6.235	6.86
Frida module loaded	0.181	0.208	0.329	0.4
Inline hook	4.022	4.073	7.8	9.82
Total	25.181	26.998	31.023	35.392

E. Discussion and Limitations

The results show that the proposed adaptive framework can effectively detect dynamic analysis of programs. Strong indicators, such as debugger detection or inline hooking, have a greater effect on the final decision, while weaker indicators, such as timing delays, only influence the result when they appear together with other signals. Additionally, the experimental results are consistent across different scenarios. This is expected as the detection approach focuses on identifying debugging and instrumentation techniques, which are usually the same regardless of the complexity of the target program. Therefore, increasing program size or complexity

does not significantly affect the detection result but slightly affects the performance.

The experimental results also show that a high confidence score is obtained when active analysis is performed using either debugging or instrumentation. However, if both techniques are used simultaneously for dynamic analysis at the same time, this can further increase the confidence score. This is because in the framework, each indicator is evaluated independently, and overlapping signals contribute cumulatively to the overall confidence score.

As with other anti-analysis approaches, the proposed framework may still be challenged by skilled attackers. Individual indicators are not intended to provide definitive proof of analysis and may be bypassed when used independently. For example, attackers may rename tools such as Frida or x64dbg to evade simple process detection. The following discussion outlines more advanced attack strategies that may target specific indicators and describes possible mitigation approaches within the framework.

One potential attack strategy involves delayed debugger attachment, in which an attacker allows the program to start normally and attaches a debugger only after the initial checks have completed. To mitigate this risk, the framework can repeatedly evaluate indicators throughout program execution rather than only during startup. This approach increases the likelihood of detecting debugging activity introduced at later stages.

Another possible attack involves temporarily applying inline hooks immediately before API invocation and restoring the original function afterward to evade hook detection. Although this approach may conceal hooks temporarily, it requires repeated modification of executable memory during runtime. The framework can mitigate this behavior by validating function prologues when monitored API functions are invoked, thereby increasing the probability of detecting such transient modifications.

Attackers may also attempt to hook the framework's own indicator-checking functions and force them to return benign results. To reduce the effectiveness of this strategy, some checks can be implemented using raw system calls instead of standard Windows APIs, making interception substantially more difficult.

Another evasion technique involves runtime linking to inject DLLs after the initial DLL enumeration phase, potentially bypassing DLL injection detection. Because these libraries are loaded after the security checks are complete, they may not appear during the initial scan. A possible mitigation strategy is periodic rescanning of loaded modules throughout program execution to identify libraries injected at later stages.

One of the limitations of the proposed framework is its inability to detect kernel-level debugging. API-based debugger checks and debug register inspection can potentially be bypassed by kernel-level debuggers, which operate with elevated privileges and monitor program execution externally to the user-mode environment. Consequently, many user-mode anti-debugging techniques become ineffective under such conditions. Detecting this type of analysis would require kernel-level monitoring or operating-system integrity mechanisms, and since the proposed framework operates entirely in user mode, protection against kernel-level debugging falls outside the scope of this work.

Another limitation involves manually injected instrumentation DLLs. Certain instrumentation frameworks may evade standard DLL injection detection by avoiding conventional injection mechanisms. For example, Frida can perform manual injection without relying on the traditional LoadLibrary API, and in these cases, injected modules may not appear in standard module enumeration lists typically used for DLL detection.

However, to overcome these limitations, the framework relies on multiple independent indicators obtained from different sources. Manipulating all detection functions consistently is therefore more complex than bypassing a single check. An attacker may evade one or two checks, but passing all indicators simultaneously requires greater effort.

Anti-dynamic analysis techniques are often combined with static protections such as obfuscation or packing. However, these protections do not significantly interfere with the proposed framework, as modifications to code structure and hidden imported APIs are typically resolved at runtime. Since the framework primarily relies on runtime indicators, it remains largely independent of the static structure of the import table, and consequently, changes introduced by packing or obfuscation are unlikely to substantially affect its detection capability.

Runtime analysis tools such as Frida are commonly used by developers for legitimate testing and debugging purposes, in addition to reverse engineering activities. To minimize false positives during software development, the proposed framework is intended for integration during later development stages, after completion of core application functionality. This deployment strategy allows runtime monitoring to focus primarily on unauthorized analysis rather than legitimate testing activities.

VI. CONCLUSION

This paper presented an adaptive framework for resisting dynamic analysis through runtime defenses targeting debugging and instrumentation by integrating multiple anti-debugging and anti-hooking techniques into a single framework, which makes dynamic analysis more difficult and time-consuming. In this framework, each indicator contributed to the final confidence score according to its assigned weight, demonstrating the practical effectiveness of the score-based decision model.

The experimental results demonstrated that the framework effectively distinguishes between normal execution and different analysis scenarios. Specifically, during benign execution, no indicators were triggered, and the programs executed normally. In contrast, when debugging or instrumentation tools were applied, multiple indicators were activated, resulting in higher confidence scores and adaptive modifications to program behavior. These findings confirm that

combining multiple runtime indicators provides more reliable detection than relying on a single mechanism.

The proposed framework also demonstrated lightweight performance characteristics. Experimental evaluation showed that execution of all indicators required approximately 25 ms for smaller programs and up to 35 ms for larger programs under heavier system load. Additionally, runtime overhead increased with the number of monitored Application Programming Interfaces (APIs), since each API verification required approximately 4 ms on average.

Overall, the results indicate that the framework can substantially increase the difficulty of dynamic analysis while maintaining low runtime overhead, as execution of all indicators required only a short processing time without noticeably affecting normal program execution.

In addition, the framework is modular and can be extended to support runtime instrumentation frameworks beyond Frida without substantial redesign. This is because the detection model relies primarily on behavior-based runtime indicators rather than framework-specific signatures. Consequently, support for additional instrumentation frameworks can be incorporated by introducing suitable indicators and assigning corresponding confidence weights. Therefore, the adaptive scoring model enables integration of new analysis tools while preserving the existing decision-making architecture.

Future work will focus on incorporating additional indicators and evaluating the framework using real-world applications and analysis environments to further assess its effectiveness and robustness.

DECLARATION OF COMPETING INTERESTS

The author declares no known competing financial interests or personal relationships that could have influenced the work reported in this paper.

ACKNOWLEDGMENT

This research work was conducted during the sabbatical leave from the University of Jordan for the academic year 2024/2025, and was carried out at the Cyber Security Department, King Hussein School of Computing Sciences, Princess Sumaya University for Technology, Amman, Jordan.

DATA AVAILABILITY

Not applicable to this work.

REFERENCES

[1] *x64dbg: An open-source x64/x32 debugger for Windows*. (2025). x64dbg. [Online]. Available: https://x64dbg.com.

[2] *Frida: A world-class dynamic instrumentation toolkit*. (2026). O. A. V. Ravnås. [Online]. Available: https://frida.re.

[3] M. A. Haq and M. Khan, "DNNBoT: Deep neural network-based botnet detection and classification," *Computers, Materials & Continua*, vol. 71, no. 1, Oct. 2021, https://doi.org/10.32604/cmc.2022.020938.

[4] M. A. Haq, M. A. R. Khan, and T. AL-Harbi, "Development of PCCNN-based network intrusion detection system for EDGE computing," *Computers, Materials and Continua*, vol. 71, no. 1, pp. 1769–1788, 2021, https://doi.org/10.32604/cmc.2022.018708.

[5] M. A. Haq and M. Khuthaylah, "Leveraging Machine Learning for Android Malware Analysis: Insights from Static and Dynamic Techniques," *Engineering, Technology & Applied Science Research*, vol. 14, no. 4, pp. 15027–15032, Aug. 2024, https://doi.org/10.48084/etasr.7632.

[6] A. Alhussen, "Advanced Android Malware Detection through Deep Learning Optimization," *Engineering, Technology & Applied Science Research*, vol. 14, no. 3, pp. 14552–14557, June 2024, https://doi.org/10.48084/etasr.7443.

[7] M. Abu-Jazoh, I. Almomani, and K. E. Sabri, "DCmal-2025: A Novel Routing-Based DisConnectivity Malware—Development, Impact, and Countermeasures," *Applied Sciences*, vol. 15, no. 18, Sept. 2025, Art. no. 10219, https://doi.org/10.3390/app151810219.

[8] J. C. De La Torre, J. Jareño, J. M. Aragón-Jurado, S. Varrette, and B. Dorronsoro, "Source code obfuscation with genetic algorithms using LLVM code optimizations," *Logic Journal of the IGPL*, vol. 33, no. 5, Aug. 2025, https://doi.org/10.1093/jigpal/jzae069.

[9] M. Schloegel *et al.*, "Loki: Hardening code obfuscation against automated attacks," in *31st USENIX security symposium (USENIX security 22)*, Boston, MA, USA, Aug. 2022, pp. 3055–3073.

[10] A. Norby, B. P. Rimal, and B. Brizendine, "Measurement of Anti-Debugging Techniques on the Windows and Linux Operating Systems for the Intel x86_64 Architecture," *IEEE Access*, vol. 13, pp. 46568–46583, 2025, https://doi.org/10.1109/ACCESS.2025.3550009.

[11] M. Steinböck *et al.*, "SoK: Hardening Techniques in the Mobile Ecosystem — Are We There Yet?," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, June 2025, pp. 789–806, https://doi.org/10.1109/EuroSP63326.2025.00050.

[12] D. Pizzolotto, S. Berlato, and M. Ceccato, "Mitigating Debugger-based Attacks to Java Applications with Self-debugging," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–38, May 2024, https://doi.org/10.1145/3631971.

[13] E. Soriano-Salvador and G. Guardiola-Múzquiz, "Detecting and bypassing frida dynamic function call tracing: exploitation and mitigation," *Journal of Computer Virology and Hacking Techniques*, vol. 19, no. 4, pp. 503–513, Dec. 2022, https://doi.org/10.1007/s11416-022-00458-7.

[14] I.-A. Császár and R. R. Slavescu, "Building fast and reliable reverse engineering tools with Frida and Rust," in *2022 IEEE 18th International Conference on Intelligent Computer Communication and Processing (ICCP)*, Cluj-Napoca, Romania, Sept. 2022, pp. 289–294, https://doi.org/10.1109/ICCP56966.2022.10053941.