Implementation and Analysis of VxLang Virtualization Effectiveness in complicating Reverse Engineering

Seno Pamungkas Rahman* *Department of Electrical Engineering, Faculty of Engineering, Universitas Indonesia, Depok, 16424, Indonesia*
Email: seno.pamungkas@ui.ac.id

***Abstract*—Reverse engineering poses a significant threat to software security, enabling attackers to analyze, understand, and illicitly modify program code. Code obfuscation techniques, particularly code virtualization, offer a promising defense mechanism. This paper presents an implementation and analysis of the effectiveness of code virtualization using the VxLang framework in enhancing software security against reverse engineering. We applied VxLang's virtualization to critical sections of case study applications, including authentication logic. Static analysis using Ghidra and dynamic analysis using x64dbg were performed on both the original and virtualized binaries. The results demonstrate that VxLang significantly increases the complexity of reverse engineering. Static analysis tools struggled to disassemble and interpret the virtualized code, failing to identify instructions, functions, or meaningful data structures. Dynamic analysis was similarly hampered, with obfuscated control flow and the virtual machine's execution model obscuring runtime behavior and hindering debugging attempts. Analysis extended to a Remote Administration Tool (RAT) demonstrated maintained functionality post-virtualization and significantly altered malware detection profiles on VirusTotal, indicating successful evasion of signature-based detection. However, this enhanced security comes at the cost of substantial performance overhead, observed in QuickSort algorithm execution and AES encryption benchmarks, along with a significant increase in executable file size. The findings confirm that VxLang provides robust protection against reverse engineering but necessitates careful consideration of the performance trade-offs for practical deployment.**

***Index Terms*—Code Obfuscation, Code Virtualization, Software Protection, Reverse Engineering, VxLang, Security Analysis, Performance Overhead.**

I. Introduction

THE rapid advancement of software technology has led to increasingly sophisticated applications, yet this progress is paralleled by evolving security threats. Reverse engineering, the process of analyzing software to understand its internal workings without access to source code or original designs [1], represents a critical vulnerability. Attackers leverage reverse engineering to uncover proprietary algorithms, identify security flaws, bypass licensing mechanisms, pirate software, and inject malicious code [2]. Traditional security measures like data encryption or password protection often prove insufficient against determined reverse engineers who can analyze the program's logic once it is running [3].

Manuscript received Month DD, YYYY; revised Month DD, YYYY. This work was supported in part by [Funding Agency, Grant Number, if any]. (Corresponding author: Seno Pamungkas Rahman.)

To counter this threat, code obfuscation techniques aim to transform program code into a functionally equivalent but significantly harder-to-understand form [4]. Among various obfuscation strategies, code virtualization stands out as a particularly potent approach [5], [6]. This technique translates native machine code into custom bytecode instructions executed by a dedicated virtual machine (VM) embedded within the application [7]. The unique Instruction Set Architecture (ISA) of this VM renders conventional reverse engineering tools like disassemblers and debuggers largely ineffective, as they cannot directly interpret the virtualized code [8]. Attackers must first decipher the VM's architecture and bytecode, substantially increasing the effort and complexity required for analysis [9].

VxLang is a code protection framework that incorporates code virtualization capabilities, targeting Windows PE executables [10]. It provides mechanisms to transform native code into its internal bytecode format, executed by its embedded VM. Understanding the practical effectiveness and associated costs of such tools is crucial for developers seeking robust software protection solutions.

This paper investigates the effectiveness of code virtualization using VxLang in mitigating reverse engineering efforts. We aim to answer the following key questions:

1) How effectively does VxLang's code virtualization obscure program logic against static and dynamic reverse engineering techniques?
2) What is the quantifiable impact of VxLang's virtualization on application performance (execution time) and file size?

To address these questions, we implement VxLang's virtualization on selected functions within case study applications (simulating authentication) and performance benchmarks (QuickSort, AES encryption). We then perform comparative static analysis (using Ghidra [11]) and dynamic analysis (using x64dbg [12]) on the original and virtualized binaries. Performance overhead is measured by comparing execution times and executable sizes, and the impact on automated malware detection tools is assessed using VirusTotal analysis on a relevant case study.

The primary contributions of this work are:

- A practical implementation and evaluation of VxLang's code virtualization on representative code segments.

- Qualitative and quantitative analysis of the increased difficulty imposed on static and dynamic reverse engineering by VxLang.
- Measurement and analysis of the performance and file size overhead associated with VxLang's virtualization.
- An empirical assessment of the security-performance trade-off offered by the VxLang framework.

The remainder of this paper is organized as follows: Section II discusses related work in code obfuscation and virtualization. Section III details the methodology employed in our experiments. Section IV briefly outlines the implementation setup. Section V presents and discusses the experimental results for both security and performance analysis. Finally, Section VI concludes the paper and suggests directions for future research.

II. Related Work

Protecting software from unauthorized analysis and tampering is a long-standing challenge. Reverse engineering techniques are constantly evolving, necessitating more sophisticated protection mechanisms. This section reviews relevant work in code obfuscation, focusing on code virtualization.

A. Code Obfuscation Techniques

Obfuscation aims to increase the complexity of understanding code without altering its functionality [4]. Techniques operate at different levels:

1) Source Code Obfuscation: Modifies the human-readable source code.

- **Layout Obfuscation:** Alters code appearance (e.g., scrambling identifiers [13], removing comments/whitespace [14]). Provides minimal security against automated tools.
- **Data Obfuscation:** Hides data representation (e.g., encoding strings [15], [16], [17], splitting/merging arrays, using equivalent but complex data types). Can make data analysis harder. Techniques like instruction substitution [18], [19] and mixed boolean-arithmetic [20], [21], [22] fall under this category, obscuring data manipulation logic.
- **Control Flow Obfuscation:** Modifies the program's execution path logic. Examples include inserting bogus control flow [23], using opaque predicates (conditional statements whose outcome is known at obfuscation time but hard to determine statically [24]), and control flow flattening, which transforms structured code into a large switch statement, obscuring the original logic [25].

2) Bytecode Obfuscation: Targets intermediate code (e.g., Java bytecode, .NET CIL, LLVM IR). Techniques include renaming identifiers, control flow obfuscation, string encryption, and inserting dummy code [26], [27]. Effective against decompilation back to high-level source code.

3) Binary Code Obfuscation: Operates on the final machine-executable code.

- **Code Packing/Encryption:** Compresses or encrypts the original code, requiring a runtime stub to unpack/decrypt it before execution [28]. Primarily hinders static analysis but reveals the original code in memory during execution.
- **Control Flow Manipulation:** Uses indirect jumps/calls, modifies call/ret instructions, or chunks code into small blocks with jumps to disrupt linear disassembly and analysis [28].
- **Constant Obfuscation:** Hides constant values through arithmetic/logical operations [28].
- **Code Virtualization:** As discussed below, this is considered one of the strongest binary obfuscation techniques.

4) Disassembly Techniques and Challenges: Understanding the efficacy of binary obfuscation, particularly code virtualization, necessitates a brief overview of disassembly techniques. Disassemblers translate machine code into human-readable assembly language, forming a cornerstone of reverse engineering [29]. Two primary approaches exist: static and dynamic disassembly.

Static disassemblers, such as Ghidra [11] and IDA Pro [30], analyze executable files without running them. They employ techniques like linear sweep or recursive traversal to identify instruction sequences [31], [32]. While comprehensive, static analysis struggles with code that is encrypted, packed, self-modifying, or significantly transformed, as the disassembler may misinterpret data as code or fail to follow the true control flow [29], [33]. Crucially, when faced with custom bytecode from a virtual machine (VM), static disassemblers designed for standard ISAs (e.g., x86-64) cannot correctly interpret these non-native instructions, leading to analysis failure.

Dynamic disassembly, typically a feature of debuggers like x64dbg [12], occurs during program execution. The debugger disassembles instructions on-the-fly as they are about to be executed by the CPU. This approach can overcome some static analysis limitations, such as revealing unpacked or decrypted code in memory [29]. Debuggers can also access runtime symbol information loaded by the operating system for system libraries, providing context for API calls. However, while a debugger can step through the native instructions of an embedded VM's interpreter, it will not directly reveal the original, pre-virtualized logic of the application. Instead, it shows the VM's internal operations executing the custom bytecode, which still obscures the application's core semantics from the analyst. These inherent limitations of standard disassembly tools underscore the challenge posed by advanced obfuscation techniques like code virtualization.

B. Code Virtualization (VM-Based Obfuscation)

Code virtualization translates native code into a custom bytecode format, executed by an embedded virtual machine (VM) [5], [6]. This creates a significant barrier for reverse engineers, as standard tools cannot interpret the custom ISA [8]. The attacker must first understand the VM's architecture, handler implementations, and bytecode mapping, which is a complex and time-consuming task [7], [9].

Key aspects of VM-based obfuscation include:

- **Custom ISA:** Each protected application can potentially have a unique or mutated set of virtual instructions, hindering signature-based detection or analysis reuse.

Oreans highlights the possibility of generating diverse VMs for different protected copies [5].

- **VM Architecture:** Typical VM components include fetch, decode, dispatch, and handler units, mimicking CPU operations but implemented in software [8], [9]. The complexity and implementation details of these handlers directly impact both security and performance.
- **Security vs. Performance Trade-off:** The interpretation layer introduced by the VM inherently adds performance overhead compared to native execution. The level of obfuscation within the VM handlers and the complexity of the virtual instructions influence this trade-off.

Several commercial tools like VMProtect [34] and Themida [35] (which also includes virtualization features beyond basic packing) employ code virtualization. Academic research has also explored techniques like symbolic deobfuscation to analyze virtualized code [8] and methods to enhance virtualization robustness, such as virtual code folding [7].

C. VxLang in Context

VxLang positions itself as a comprehensive framework offering binary protection, code obfuscation (including flattening), and code virtualization [10]. Its approach involves transforming native x86-64 code into an internal bytecode executed by its VM. This study aims to provide an empirical evaluation of the effectiveness of VxLang's virtualization component against standard reverse engineering practices and quantify its associated performance costs, contributing practical insights into its utility as a software protection mechanism. Unlike analyzing established commercial protectors, this work focuses on the specific implementation and impact of the VxLang framework.

III. Methodology

This research employs an experimental approach to evaluate the effectiveness of VxLang's code virtualization. We compare the reverse engineering difficulty and performance characteristics of software binaries before and after applying VxLang's virtualization.

A. Experimental Design

A comparative study design was used, involving a control group (original, non-virtualized binaries) and an experimental group (binaries with critical sections virtualized by VxLang).

- **Independent Variable:** Application of VxLang code virtualization (Applied vs. Not Applied).
- **Dependent Variables:**
  - *Reverse Engineering Protection Effectiveness:* Evaluated quantitatively and through systematic observation. This includes recording the success/failure rates of authentication bypass attempts, analyzing the behavior of static analysis tools (e.g., Ghidra's ability to disassemble code, identify critical strings, and map control flow), and observing outcomes of dynamic analysis (e.g., x64dbg's effectiveness in tracing execution, identifying runtime data, and success/failure of runtime manipulation). Metrics such as the number of obfuscated critical strings, tool-reported disassembly failures, and binary indicators of bypass success will be used.
  - *Performance Overhead:* Quantitatively measured via execution time for specific computational tasks (QuickSort, AES encryption/decryption).
  - *File Size Overhead:* Quantitatively measured by comparing the size (in bytes) of the final executable files.

B. Study Objects

Two categories of applications were developed and analyzed:

1) Authentication Case Study Applications: Simple applications simulating user login were created to serve as targets for reverse engineering analysis focused on bypassing the authentication mechanism. Variants included:

- **Interface Types:** Console (CLI), Qt Widgets (GUI), Dear ImGui (Immediate Mode GUI).
- **Authentication Mechanisms:** Hardcoded credentials (comparing input against string literals) and Cloud-based validation (sending credentials via HTTP POST to a local backend server).

For each variant, the core authentication logic (comparison function or the call to the cloud request function and subsequent result check) was targeted for virtualization in the experimental group.

2) Performance Benchmark Applications: Applications designed to measure the performance impact of virtualization on specific computational tasks:

- **QuickSort Benchmark:** Implemented a standard recursive QuickSort algorithm. The core recursive function was virtualized. Tested with varying array sizes (100 to 1,000,000 elements).
- **AES Encryption Benchmark:** Implemented AES-256-CBC encryption/decryption using OpenSSL's EVP API. The loop performing batch encryption/decryption operations on 1GB of data was virtualized.
- **File Size Benchmark:** A minimal application with embedded dummy data to assess the baseline size increase due to the inclusion of the VxLang runtime.

3) Case Study: Lilith RAT: To further evaluate VxLang's effectiveness on more complex software potentially exhibiting malicious characteristics and to assess its impact on automated detection tools, a Remote Administration Tool (RAT) named Lilith [36] was included as an additional study object. The client component of this open-source C++ RAT was compiled and analyzed both in its original form and after applying VxLang virtualization to its core functions. Analysis focused on static/dynamic reverse engineering difficulty, functional integrity post-virtualization, and detection rates by antivirus engines via VirusTotal. The Lilith server component remained unmodified and was used for functional testing of the client.

C. Instrumentation and Materials

- **Hardware:** Standard Windows 11 (64-bit) PC.
- **Development Tools:** Clang/clang-cl (C++17), CMake, Ninja, Neovim.
- **Libraries/Frameworks:** VxLang SDK, Qt 6, Dear ImGui (+GLFW/OpenGL3 backend), OpenSSL 3.x, libcurl, nlohmann/json.
- **Analysis Tools:** Ghidra (v11.x) for static analysis, x64dbg (latest snapshot) for dynamic analysis.
- **Performance Measurement:** C++ `std::chrono::high_resolution_clock` for timing, `std::filesystem::file_size` for file size.

D. Data Collection Procedure

1) Security Analysis: For each authentication application (original and virtualized):

1) **Static Analysis (Ghidra):** Load executable, search for relevant strings (e.g., "Failed", "Authorized", potential credentials), analyze disassembly/decompilation around string references or entry points, identify conditional jumps controlling authentication success/failure, attempt static patching to bypass logic. Record systematic observations regarding tool behavior (e.g., string search success, disassembly quality, logic identifiability) and the success/failure of static patching attempts.
2) **Dynamic Analysis (x64dbg):** Run executable under debugger, search for strings/patterns at runtime, set breakpoints at suspected logic locations (identified via static analysis or runtime observation), step through execution, observe register/memory values, attempt runtime manipulation (patching conditional jumps) to bypass authentication. Record systematic observations regarding runtime behavior, data visibility (especially for virtualized strings), the success/failure of runtime manipulation for bypass, and other objective indicators of analysis complexity.

2) Performance Analysis: For each benchmark application (original and virtualized):

1) **Execution Time:** Run QuickSort benchmark 100 times per data size, record individual times. Run AES benchmark on 1GB data, record total encryption/decryption time. Use `std::chrono`. Calculate average, standard deviation (for QuickSort), and throughput (for AES).
2) **File Size:** Measure the size of the final executable file in bytes using `std::filesystem::file_size`.

3) Lilith RAT Analysis: For the Lilith RAT client (original and virtualized):

1) **Functional Integrity Testing:** The virtualized client was tested for core RAT functionalities (connection to server, remote command execution, file system access) against an unmodified server on a local network (client IP: `192.168.1.15`, server IP: `192.168.1.235` on port `1337`) to ensure virtualization did not break essential operations. A test file (`password.txt` containing `"THIS IS A SECRET"`) on the client machine was used to verify remote file access.
2) **VirusTotal Analysis:** Both original and virtualized client executables were submitted to VirusTotal to compare detection rates and threat characterizations by various antivirus engines.

E. Data Analysis Techniques

- **Security Protection Analysis:** Quantitative metrics (e.g., bypass success rates, percentage of critical strings successfully obfuscated, reduction in identifiable functions) and descriptive analysis of systematically recorded observations from static and dynamic analysis (e.g., tool disassembly success, string visibility) will be used to compare the protection effectiveness between control and experimental groups.
- **Performance Data:** Calculation of descriptive statistics (mean, standard deviation), percentage overhead for execution time, throughput calculation (MB/s), and percentage increase in file size. Comparative tables and graphs will be used for presentation.
- **Trade-off Analysis:** Synthesis of security findings and performance results to evaluate the balance between protection enhancement and performance/size costs introduced by VxLang.

IV. Implementation Details

This section briefly outlines the key aspects of the experimental setup and the integration of VxLang.

A. Development Environment

All development and testing were conducted on a Windows 11 (64-bit) system. The Clang compiler (v19.1.3, via `clang-cl` for MSVC ABI compatibility) targeting x86-64 was used with the C++17 standard. CMake (v3.31) and Ninja (v1.12.1) managed the build process. Essential libraries included the VxLang SDK, Qt 6, Dear ImGui, OpenSSL 3.x, and libcurl, linked appropriately via CMake.

B. VxLang Integration

VxLang was applied to the target applications using its Software Development Kit (SDK) and external processing tool.

1) Code Marking: Critical code sections intended for virtualization were demarcated in the C++ source code using the SDK's macros, primarily `VL_VIRTUALIZATION_BEGIN` and `VL_VIRTUALIZATION_END`. For instance, in the authentication logic:

```
// ... Input username/password ...
#ifdef USE_VL_MACRO
VL_VIRTUALIZATION_BEGIN; // Mark start
#endif

if (check_credentials(username, password)) {
    // Authorized path
} else {
    // Unauthorized path
}

#ifdef USE_VL_MACRO
```

```
VL_VIRTUALIZATION_END; // Mark end
#endif
// ...
```

Similar macros were placed around the recursive `quickSort` function body and the main encryption/decryption loop in the AES benchmark.

2) Build Process: The CMake configuration was set up to generate two distinct build types:

1) **Original Build:** Compiled without the `USE_VL_MACRO` preprocessor definition and without linking the VxLang library. Produces the baseline executable (e.g., `app_qt.exe`).
2) **Intermediate Build (VM Marked):** Compiled with `USE_VL_MACRO` defined and linked against `vxlib64.lib`. Produces an intermediate executable containing the VxLang markers (e.g., `app_qt_vm.exe`).

3) Virtualization Processing: The intermediate executables (e.g., `app_qt_vm.exe`) generated by the build process, which contain the VxLang markers and are linked against the VxLang library, were then directly processed using the VxLang command-line tool. This was done by executing the tool with the intermediate executable as an argument, for example:

```
vxlang.exe app_qt_vm.exe
```

This command automatically processes the input file, replacing the native code within the marked sections (`VL_VIRTUALIZATION_BEGIN/END`) with its corresponding virtualized bytecode and embedding the necessary VM runtime. The tool generates the final virtualized executable in the same directory, automatically appending `_vxm` to the original filename (e.g., producing `app_qt_vxm.exe`). This resulting `*_vxm.exe` file was then used for all subsequent testing and analysis. No explicit configuration files (like JSON) were used in this processing step.

C. Lilith RAT Preparation

The Lilith RAT client source code [36] was compiled using the same environment (Clang/clang-cl, CMake, Ninja). For the virtualized version, VxLang SDK macros (`VL_VIRTUALIZATION_BEGIN/END`) were strategically placed around key functional blocks within the client's source code, targeting areas responsible for connection handling, command processing, and core RAT functionalities. An intermediate executable (`Lilith_Client_vm.exe`) was built with the `USE_VL_MACRO` flag and linked against `vxlib64.lib`. This intermediate file was then directly processed using the external VxLang command-line tool (`vxlang.exe Lilith_Client_vm.exe`) to produce the final virtualized Lilith client executable (`Lilith_Client_vxm.exe`) used in the analysis. The Lilith server component remained unmodified.

It is pertinent to note that the strategic placement of VxLang macros was crucial for maintaining the functional integrity of the Lilith RAT client. Initial attempts to virtualize larger, more complex code blocks, particularly those involving intricate control flow (e.g., entire switch-case statements handling packet types) or direct network I/O calls, occasionally resulted in application instability or crashes. Stable functionality was achieved by applying virtualization more granularly to specific, well-contained functions or critical logical segments, underscoring the need for careful, iterative testing when integrating VM-based obfuscation into complex applications.

TABLE I
GHIDRA STATIC ANALYSIS METRICS COMPARISON (NON-VIRTUALIZED VS. VIRTUALIZED)

Application	Version	Instructions	Functions	Defined Data	Symbols
app_qt (GUI)	Non-Virt.	6104	538	1578	2113
	Virtualized	214	25	174	103
	Change %	**-96.49%**	**-95.35%**	**-88.97%**	**-95.13%**
console (CLI)	Non-Virt.	3090	261	726	1018
	Virtualized	174	20	146	88
	Change %	**-94.37%**	**-92.34%**	**-79.89%**	**-91.36%**
encryption (Benchmark)	Non-Virt.	6282	368	849	1920
	Virtualized	159	20	155	77
	Change %	**-97.47%**	**-94.57%**	**-81.74%**	**-95.99%**

V. RESULTS AND DISCUSSION

This section presents the results of the security analysis and performance measurements, followed by a discussion of the findings.

A. Security Analysis Results

The effectiveness of VxLang virtualization was evaluated through static and dynamic analysis attempts to understand and bypass the authentication logic in the case study applications.

1) Static Analysis (Ghidra): Static analysis of non-virtualized binaries using Ghidra was generally straightforward. Relevant strings (e.g., "Authentication Failed") and control flow for authentication logic were typically identifiable. For instance, in non-virtualized binaries, standard comparison instructions and conditional jumps controlling authentication were readily found (see Appendix Listings **??**, **??** in the full thesis for detailed examples), making static patching feasible.

Conversely, static analysis proved significantly more challenging for binaries processed by VxLang. Table I summarizes key metrics from Ghidra analysis for representative applications, illustrating the impact of virtualization.

The data in Table I consistently shows a drastic reduction (typically ¿90%) in the number of recognizable instructions, functions, defined data, and symbols in virtualized binaries. For instance, the `app_qt` application saw its recognizable instructions decrease from 6104 to 214 (a **-96.49%** change) and functions from 538 to 25 (a **-95.35%** change) after virtualization. This starkly contrasts with non-virtualized versions, indicating a fundamental transformation of the code into a format uninterpretable by standard disassembly. This reduction in identifiable program elements severely impedes static analysis, making it nearly impossible to locate and comprehend the relevant control flow logic for authentication or other critical functions. Static bypass attempts on virtualized binaries were consequently unsuccessful. These findings align with the understanding that static disassemblers struggle with custom

TABLE II
X64DBG DYNAMIC ANALYSIS METRICS COMPARISON
(NON-VIRTUALIZED VS. VIRTUALIZED)

Application	Version	Instr. Count (Observed)	Mem. Sections	Def. Symbols	Key Str. Found
app_qt (GUI)	Non-VM	8022	7	209	Yes
	VM	8011	10	15	No
	Change %	-0.14%	+42.86%	**-92.82%**	-
console (CLI)	Non-VM	5797	6	88	Yes
	VM	5843	9	12	No
	Change %	+0.79%	+50.00%	**-86.36%**	-
encryption (Benchmark)	Non-VM	8336	6	94	Yes
	VM	8207	9	13	No
	Change %	-1.55%	+50.00%	**-86.17%**	-

bytecode ISAs introduced by virtualization [29], [31], [32]. The file size also increased substantially (e.g., `app_qt.exe` from 122 KB to 1,578 KB after virtualization, a 13x increase), indicative of the embedded VM and bytecode.

2) Dynamic Analysis (x64dbg): Dynamic analysis of non-virtualized binaries using x64dbg was generally straightforward. Setting breakpoints based on string references or near conditional jumps identified during static analysis proved effective. Stepping through the code clearly revealed comparison logic and conditional jumps, and runtime patching successfully bypassed authentication.

For VxLang-virtualized binaries, dynamic analysis presented a more nuanced challenge, as summarized by key metrics in Table II.

Key observations from dynamic analysis of virtualized binaries include:

- **Visibility of Native VM Instructions:** Once the virtualized application was fully loaded, x64dbg displayed valid native x86-64 instructions. These instructions belong to the VxLang VM interpreter, not the original application's native logic. The observed instruction count (Table II) did not drastically change, reflecting VM activity rather than revealing the original program's complexity to the analyst.
- **Increased Memory Sections:** The number of memory sections consistently increased (by 40-50%), likely due to the VxLang runtime and bytecode.
- **Persistent Obscurity of Critical Data and Application Logic:**
  - **Key String Obfuscation:** Critical strings targeted for virtualization (e.g., "Authentication Failed") were **not found** using standard runtime searches in x64dbg (Table II, "Key Str. Found": No). This demonstrates effective runtime string protection.
  - **Drastic Reduction in Defined Symbols:** Observable defined symbols were significantly reduced (by ¿85%), hampering contextual understanding and navigation.
  - **Abstraction of Application Logic:** The core application logic was transformed into internal bytecode executed by the VxLang VM. Debuggers show the VM's execution, not the direct native execution of the original logic, making it extremely difficult to trace or understand the application's intended behavior.
- **Ineffectiveness of Simple Runtime Patching:** Consequently, attempts to bypass authentication by patching simple conditional jumps at runtime were rendered ineffective. The critical decision-making points were embedded within the VM's opaque execution flow.

These dynamic analysis findings align with the principles of VM-based obfuscation [5], [29]. While the debugger can step through the VM interpreter's native code, the actual application logic is abstracted away, making direct analysis and manipulation exceptionally difficult without deep knowledge of the VM's architecture [8].

3) Analysis of Lilith RAT: After careful and iterative placement of VxLang macros, functional testing confirmed that the virtualized Lilith client (`Lilith_Client_vm.vxm.exe`) remained fully operational. Initial, broader applications of virtualization macros to large code blocks within Lilith, especially around complex control flow constructs (e.g., switch-cases for packet processing) or I/O operations (e.g., `SendString` calls), led to functional issues such as application crashes when specific commands like `remoteControl cmd` were triggered. Stable operation, as demonstrated in a two-machine local network setup (client IP: `192.168.1.15`, server IP: `192.168.1.235`, port `1337`), was achieved by refining macro placement to target more specific, self-contained critical functions. The virtualized client successfully connected to the unmodified Lilith server, allowed remote command prompt access, and facilitated remote file reading (`password.txt` containing `"THIS IS A SECRET"`). This confirmed that, with precise application, VxLang's virtualization can preserve core RAT functionalities (network communication, command execution, file system interaction) despite significant code transformation. This highlights the necessity of meticulous macro placement and thorough functional validation when applying such obfuscation to multi-faceted software.

This indicates VxLang's virtualization can be applied to complex software, potentially including those with malicious characteristics, without necessarily breaking its intended functionality, while, as shown next, impacting its detectability.

B. Performance and Size Overhead Results

1) Execution Time Overhead: The performance impact was measured using QuickSort and AES benchmarks.

- **QuickSort:** As shown in Table III and Fig. 1, virtualization introduced substantial execution time overhead. The overhead increased with data size, ranging from approximately 27,300% for 100 elements (0.01 ms to 2.74 ms) to about 15,150% for 1,000,000 elements (218.32 ms to 33,292.91 ms). This indicates a significant constant overhead plus a scaling factor imposed by the VM's interpretation loop for the recursive sorting function.
- **AES Encryption:** Table IV shows that the total time for encrypting 976MB of data increased by approximately 396.7% (1878.52 ms to 9330.73 ms), and decryption time increased by about 562.9% (1304.75 ms to 8649.74 ms). Consequently, the combined throughput dropped dramatically from 634.16 MB/s to 108.78 MB/s (an 82.8% reduction). This confirms a significant overhead for cryptographic operations.

TABLE III
QUICK SORT EXECUTION TIME RESULTS (MS)

Array Size	Non-Virtualized		Virtualized	
	Avg Time	Std Dev	Avg Time	Std Dev
100	0.01	0.00	2.74	0.38
1,000	0.08	0.00	27.35	1.25
5,000	0.54	0.05	144.44	8.25
10,000	1.24	0.08	295.77	13.68
50,000	6.98	0.51	1,556.15	122.81
100,000	15.12	1.26	3,080.30	303.02
500,000	104.44	7.30	14,298.92	374.98
1,000,000	218.32	8.10	33,292.91	4,342.93

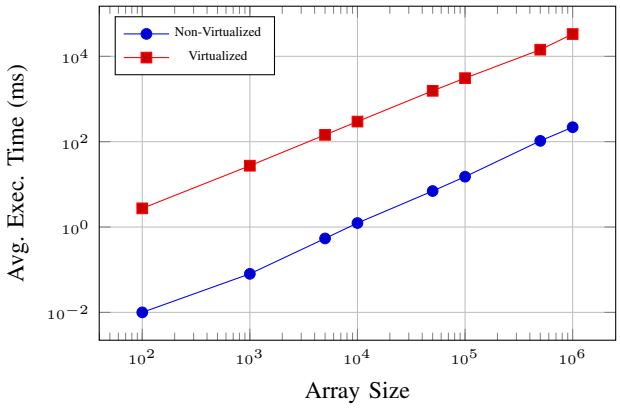

Fig. 1. Quick Sort Execution Time Comparison (Log-Log Scale).

TABLE IV
AES-256-CBC PERFORMANCE RESULTS (976MB DATA)

Metric	Non-Virtualized	Virtualized
Total Encryption Time (ms)	1,878.52	9,330.73
Total Decryption Time (ms)	1,304.75	8,649.74
Avg. Encrypt Time/Block (ms)	0.00188	0.00933
Avg. Decrypt Time/Block (ms)	0.00130	0.00865
Encrypt Throughput (MB/s)	519.86	104.66
Decrypt Throughput (MB/s)	748.46	112.90
Combined Throughput (MB/s)	634.16	108.78

2) File Size Overhead: Table V shows a consistent increase in executable file size after virtualization. For smaller console/benchmark programs (`quick_sort`, `encryption`, `console`, `Lilith_Client`), the size increased by over 15-18 times (from 80-110 KB to 1.5-1.6 MB). For larger GUI applications (`app_imgui` from 1,675 KB to 2,330 KB; `app_qt` from 122 KB to 1,578 KB) and the benchmark with embedded data (`size` from 97,771 KB to 112,324 KB), the relative increase was smaller but still significant. This overhead is primarily attributed to the inclusion of the VxLang VM runtime and the bytecode representation of the original code.

TABLE V
EXECUTABLE FILE SIZE COMPARISON (KB)

Program	Non-Virtualized (KB)	Virtualized (KB)
quick_sort	98	1,537
encryption	110	1,507
size	97,771	112,324
console	92	1,577
console_cloud	281	1,695
app_imgui	1,675	2,330
app_imgui_cloud	1,860	2,418
app_qt	122	1,578
app_qt_cloud	315	1,671
Lilith_Client	84	1,554

C. VirusTotal Detection Analysis

To assess the impact of VxLang virtualization on automated malware detection, both the original and virtualized Lilith RAT client executables were submitted to VirusTotal.

- **Non-Virtualized Lilith:** Detected by **22 out of 72** engines. Analysis revealed specific threat labels like ”trojan.lilithrat/keylogger” and family labels including ”lilithrat” and ”keylogger”. Detections often included specific names like ”Backdoor:Win64/LilithRat.GA!MTB” or ”Trojan[Backdoor]/Win64.LilithRAT”.
- **Virtualized Lilith:** Detected by **18 out of 72** engines, showing a decrease in detection rate. The popular threat label became a generic ”trojan”, and specific family labels disappeared. Detection signatures shifted towards generic malware, heuristic-based flags, AI/ML detections, or packed/protected software warnings (e.g., ”Trojan:Win32/Wacatac.C!ml”, ”Static AI - Suspicious PE”, ”ML.Attribute.HighConfidence”, ”RiskWare[Packed]/Win32.VMProtect.a”).

These results suggest that VxLang virtualization effectively obfuscates static signatures used by many traditional antivirus engines, forcing reliance on less specific heuristic or AI-based methods, and potentially evading detection by some vendors altogether.

D. Discussion

The experimental results clearly demonstrate the core trade-off inherent in using VxLang’s code virtualization.

Security Enhancement and Detection Evasion: VxLang provides a substantial barrier against common reverse engineering techniques. The transformation into interpreted bytecode neutralizes standard static analysis tools like Ghidra, which rely on recognizable native instruction patterns [31], [32], and significantly complicates dynamic analysis with tools like x64dbg, as the underlying logic is executed by an opaque VM [29]. This aligns with the established understanding that VM-based obfuscation fundamentally alters the code structure beyond the interpretation capabilities of standard disassemblers [5], [8]. Furthermore, the VirusTotal analysis indicates that this obfuscation extends to automated malware detection; VxLang effectively hinders signature-based detection, reduces

overall detection rates, and forces AV engines towards more generic or heuristic approaches. This capability to evade specific detection signatures adds another layer to its protective potential, aligning with the expected benefits of advanced obfuscation [5], [8], [28].

Performance Cost: The security and evasion benefits come at a steep price in terms of performance. The interpretation overhead significantly slows down virtualized code, especially for computationally intensive tasks (QuickSort overhead of 15,000% for 1M elements; AES throughput reduction of 83%), potentially rendering indiscriminate application impractical due to severe speed degradation.

Size Increase: The considerable increase in file size (e.g., 15-18x for small applications), mainly due to the embedded VM runtime, is another factor, particularly relevant for smaller applications or distribution constraints.

Practical Implications: VxLang appears potent for protecting highly sensitive code where security and potentially detection evasion are paramount, and the performance impact on those specific segments is acceptable (e.g., anti-tamper, licensing, core IP). The Lilith case shows it can protect complex logic without breaking it. However, the severe performance cost necessitates a strategic, selective application, targeting only critical sections. The VirusTotal results also imply that while detection is hindered, it's not eliminated, especially by heuristic/AI methods or tools flagging the protection layer itself. The choice between hardcoded and cloud-based authentication showed that protecting client-side logic handling the result of validation remains crucial, reinforcing the need for techniques like virtualization on critical checks, regardless of where primary authentication occurs.

VI. Conclusion

This paper investigated the effectiveness of code virtualization using the VxLang framework as a technique to mitigate software reverse engineering. The process involved marking code with SDK macros, compiling intermediate executables (`*_vm.exe`), and processing them directly via the `vxlang.exe` command-line tool to generate final virtualized binaries (`*_vxm.exe`). Through experimental analysis involving static (Ghidra) and dynamic (x64dbg) examination of authentication applications and performance benchmarking (QuickSort, AES), we draw the following conclusions:

VxLang's code virtualization significantly enhances software security by substantially increasing the difficulty of reverse engineering. The transformation into custom bytecode rendered standard static analysis tools ineffective at interpreting program logic and control flow within virtualized sections. Dynamically, while x64dbg could observe the native instructions of the VxLang Virtual Machine (VM) itself during runtime, the core application logic remained effectively obscured. Critical strings targeted by virtualization were not discoverable via standard debugger searches, and the abstraction of the application's decision-making processes into the VM's bytecode execution made direct runtime manipulation (e.g., patching conditional jumps for authentication bypass) unfeasible. Attempts to bypass authentication logic, which were trivial in non-virtualized versions, were successfully thwarted in the virtualized binaries using the employed static and dynamic analysis techniques. **However, the application of these protective measures required careful, iterative placement of virtualization macros, as improper application, particularly in complex code sections involving I/O or intricate control flows, was found to potentially disrupt software functionality, as observed with the Lilith RAT case study.**

Furthermore, analysis of a virtualized RAT (Lilith) showed maintained functionality alongside reduced detection rates and a shift from specific signatures to generic flags on VirusTotal, demonstrating VxLang's capability to also evade traditional antivirus detection mechanisms.

However, this robust security comes with significant drawbacks. We observed substantial performance overhead, with execution times for computational tasks increasing dramatically (by factors ranging from hundreds to tens of thousands) after virtualization. Furthermore, the inclusion of the VxLang VM runtime and bytecode resulted in a considerable increase in executable file size, particularly impactful for smaller applications.

The findings highlight a clear trade-off: VxLang provides strong protection against reverse engineering at the cost of significant performance degradation and increased file size. Therefore, its practical application likely requires a selective approach, targeting only the most critical and sensitive code sections where the security benefits outweigh the performance impact.

Future work could involve exploring more advanced reverse engineering techniques specifically targeting VM-based protections to further assess VxLang's resilience. Investigating the impact of different VxLang configuration options on the security-performance balance would also be valuable. **Further research into the specific code constructs or patterns that interact poorly with VxLang's virtualization process could yield guidelines for more robust and reliable application of such protection mechanisms.** Comparative studies with other commercial or open-source virtualization solutions could provide a broader perspective. Investigating the interaction between VxLang and various antivirus detection techniques (signature-based, heuristic, AI/ML, behavioral) would also yield valuable insights into its detection evasion capabilities and limitations.

References

[1] M. Hasbi, E. K. Budiardjo, and W. C. Wibowo, "Reverse engineering in software product line - a systematic literature review," in *2018 2nd International Conference on Computer Science and Artificial Intelligence*, Shenzhen, 2018, pp. 174–179.

[2] Y. Wakjira, N. Kurukkal, and H. Lemu, "Reverse engineering in medical application: Literature review, proof of concept and future perspectives.," *Reverse engineering in medical application: literature review, proof of concept and future perspectives.*, 2024.

[3] Sec-Dudes, *Hands on: Dynamic and static reverse engineering*, https://secdude.de/index.php/2019/08/01/about-dynamic-and-static-reverse-engineering/, 2019.

[4] H. Jin, J. Lee, S. Yang, K. Kim, and D. Lee, "A framework to quantify the quality of source code obfuscation," *A Framework to Quantify the Quality of Source Code Obfuscation*, vol. 12, p. 14, 2024.

[5] Oreans, *Code virtualizer*, https://www.oreans.com/CodeVirtualizer.php, 2006.

[6] Z. Wang, Z. Xu, Y. Zhang, X. Song, and Y. Wang, "Research on code virtualization methods for cloud applications," 2024.

[7] D. H. Lee, "Vcf: Virtual code folding to enhance virtualization obfuscation," *VCF: Virtual Code Folding to Enhance Virtualization Obfuscation*, 2020.

[8] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *International Conference, DIMVA*, 2018.

[9] Hackcyom, *Hackcyom*, https://www.hackcyom.com/2024/09/vm-obfuscation-overview/, 2024.

[10] *Vxlang documentation*, https://vxlang.github.io/, 2025.

[11] National Security Agency, *Ghidra*, https://ghidra-sre.org, 2019.

[12] D. Ogilvie, *X64dbg*, https://x64dbg.com, 2014.

[13] J. T. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode," *Advanced obfuscation techniques for Java bytecode*, vol. 71, no. 1-2, pp. 1–10, 2004.

[14] V. Balachandran and S. Emmanuel, "Software code obfuscation by hiding control flow information in stack," in *IEEE International Workshop on Information Forensics and Security*, Iguacu Falls, 2011.

[15] L. Ertaul and S. Venkatesh, "Novel obfuscation algorithms for software security," in *International Conference on Software Engineering Research and Practice*, Las Vegas, 2005.

[16] K. Fukushima, S. Kiyomoto, T. Tanaka, and K. Sakurai, "Analysis of program obfuscation schemes with variable encoding technique," *Analysis of program obfuscation schemes with variable encoding technique*, vol. 91, no. 1, pp. 316–329, 2008.

[17] A. Kovacheva, "Efficient code obfuscation for android," in *Advances in Information Technology*, Bangkok, 2013.

[18] C. LeDoux, M. Sharkey, B. Primeaux, and C. Miles, "Instruction embedding for improved obfuscation," in *Annual Southeast Regional Conference*, Tuscaloosa, 2012.

[19] S. Darwish, S. Guirguis, and M. Zalat, "Stealthy code obfuscation technique for software security," in *International Conference on Computer Engineering & Systems*, Cairo, 2010.

[20] B. Liu, W. Feng, Q. Zheng, J. Li, and D. Xu, "Software obfuscation with non-linear mixed boolean-arithmetic expressions," in *Information and Communications Security*, Chongqing, 2021.

[21] M. Schloegel et al., "Hardening code obfuscation against automated attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, 2022.

[22] Y. Zhou, A. Main, Y. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *International Workshop on Information Security Applications*, Jeju Island, 2007.

[23] Y. Li, Z. Sha, X. Xiong, and Y. Zhao, "Code obfuscation based on inline split of control flow graph," in *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, Dalian, 2021.

[24] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Information Security: 19th International Conference, ISC 2016*, Honolulu, 2016.

[25] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Obfuscating C++ programs via control flow flattening*, vol. 30, pp. 3–19, 2009.

[26] P. Parrend, *Bytecode obfuscation*, https://owasp.org/www-community/controls/Bytecode_obfuscation, 2018.

[27] Yakov, *Using llvm to obfuscate your code during compilation*, https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation, 2020.

[28] Roundy, K. A, Miller, and B. P, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–32, 2013.

[29] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA: No Starch Press, 2012.

[30] Hex-Rays, *Ida pro*, https://hex-rays.com/ida-pro, 1991.

[31] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley, 2011.

[32] C.-K. Ko and J. Kinder, "Static disassembly of obfuscated binaries," in *Proceedings of the 2nd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '07)*, San Diego, California, USA: ACM, 2007, pp. 31–38.

[33] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*, Vancouver, BC, Canada: USENIX Association, 2017, pp. 843–860.

[34] VMProtect Software, *Vmprotect*, https://vmpsoft.com/vmprotect/overview.

[35] Oreans, *Themida*, https://www.oreans.com/Themida.php.

[36] werkamsus, *Lilith - free & native open source c++ remote administration tool for windows*, https://github.com/werkamsus/Lilith, Accessed: 8 May 2025, 2017.