

Implementation and Analysis of Code Virtualization Effectiveness using VxLang for Mitigating 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].

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 Difficulty:* Qualitatively assessed based on the effort required for static analysis (code understanding, logic identification, patching attempts using Ghidra) and dynamic analysis (runtime tracing, memory inspection, manipulation attempts using x64dbg). Success/failure of bypassing authentication logic was recorded.
 - *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.

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 qualitative observations on difficulty.
- 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, altering flags/memory) to bypass authentication. Record qualitative observations and success/failure of bypass attempts.

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`.

E. Data Analysis Techniques

- **Qualitative Security Data:** Descriptive analysis based on observation notes comparing the reverse engineering effort and success rates between control and experimental groups for both static and dynamic analysis phases.
- **Quantitative 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 `*_vnm.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 `vmlib64.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_vnm.exe`) used in the analysis. The Lilith server component remained unmodified.

V. RESULTS AND DISCUSSION

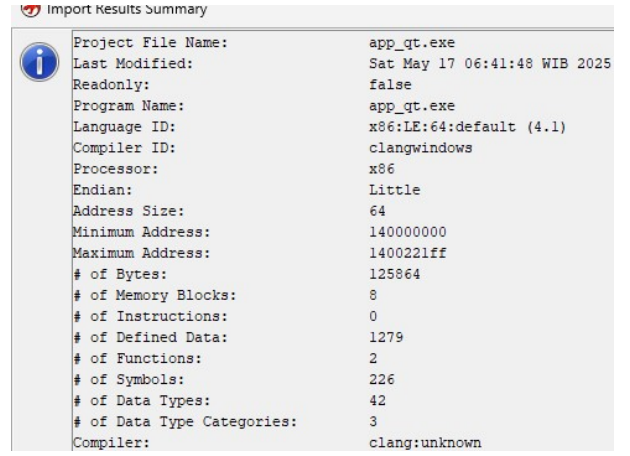
This section presents the results of the security analysis and performance measurements, followed by a discussion of the findings.

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):

- **Non-Virtualized Binaries:** Analysis of non-virtualized binaries was generally straightforward. Relevant strings and control flow for authentication logic were typically identifiable. For instance, in the non-virtualized `app_qt.exe`, while a specific recent Ghidra analysis reported an unusually low count of 0 instructions and only 2 functions (possibly an analytical anomaly for that run, with 1279 defined data entries and 226 symbols, compiler identified as 'clangwindows', size 125,864 bytes), the typical expectation for such binaries is observable logic. Standard comparison instructions and conditional jumps were readily found in other non-virtualized samples (see Appendix Listings ??, ??), making static patching feasible.
- **Virtualized Binaries:** Static analysis proved significantly more challenging for binaries processed by VxLang.
 - *Instruction and Function Obscuration:* For the virtualized `app_qt_vm.vnm.exe`, Ghidra consistently failed to recognize standard x86-64 instructions, reporting 0 instructions and 0 functions. This starkly contrasts even with the anomalous low detection in the original, indicating a fundamental transformation of the code into a format uninterpretable by standard disassembly.



Import Results Summary	
Project File Name:	app_qt.exe
Last Modified:	Sat May 17 06:41:48 WIB 2025
Readonly:	false
Program Name:	app_qt.exe
Language ID:	x86:LE:64:default (4.1)
Compiler ID:	clangwindows
Processor:	x86
Endian:	Little
Address Size:	64
Minimum Address:	140000000
Maximum Address:	1400221ff
# of Bytes:	125864
# of Memory Blocks:	8
# of Instructions:	0
# of Defined Data:	1279
# of Functions:	2
# of Symbols:	226
# of Data Types:	42
# of Data Type Categories:	3
Compiler:	clang:unknown

Fig. 1. Ghidra Analysis Summary for `app_qt.exe` (Non-Virtualized).

- *Data and Symbol Reduction:* Critical data was obscured. The count of defined data entries in `app_qt_vm.vnm.exe` dropped to 174 from 1279 in the original, and symbols decreased from 226 to 33. This impedes the identification of code sections via string or symbol searches. The compiler was also reported as 'unknown'.
- *File Size Increase:* The file size for `app_qt_vm.vnm.exe` increased substantially to 2,008,495 bytes from 125,864 bytes (approximately 15.95 times larger), indicative of the embedded VM and bytecode.
- *Control Flow Obscurity:* The clear structure of conditional checks and jumps seen in original, easily analyzable code was replaced by opaque sequences, rendering static identification and patching of the core authentication logic nearly impossible. The control flow graph became fragmented and uninformative.

Static analysis proved significantly more challenging for binaries processed by VxLang. The observed failure of Ghidra to recognize standard x86-64 instructions (reporting 0 instructions and functions for `app_qt_vm.vnm.exe`) and the drastic reduction in defined data and symbols are direct consequences of code virtualization. As outlined in Section II, static disassemblers are designed for native ISAs [29], [31]. VxLang transforms the original code into a custom bytecode format, rendering it uninterpretable by Ghidra, which attempts to map these bytes to an x86-64 ISA for which they are not valid [32]. The substantial file size increase further indicates the embedding of the VM runtime and the transformed bytecode. Static bypass attempts on virtualized binaries were unsuccessful due to the inability to locate and comprehend the relevant control flow logic. Consequently, static identification and patching of the core authentication logic became practically impossible. The figures referenced (e.g., Fig. 1 and Fig. 2 in the main thesis document) visually demonstrate these differences.

2) Dynamic Analysis (x64dbg):

Import Results Summary	
Project File Name:	app_qt_vm.vxm.exe
Last Modified:	Sat May 17 06:43:58 WIB 2025
Readonly:	false
Program Name:	app_qt_vm.vxm.exe
Language ID:	x86:LE:64:default (4.1)
Compiler ID:	windows
Processor:	x86
Endian:	Little
Address Size:	64
Minimum Address:	140000000
Maximum Address:	1401effff
# of Bytes:	2008495
# of Memory Blocks:	11
# of Instructions:	0
# of Defined Data:	174
# of Functions:	0
# of Symbols:	33
# of Data Types:	39
# of Data Type Categories:	3
Compiler:	unknown

Fig. 2. Ghidra Analysis Summary for app_qt_vm.exe (Virtualized, showing data for app_qt_vxm.exe structure).

- **Non-Virtualized Binaries:** Dynamic analysis corroborated static findings. Setting breakpoints based on string references or near conditional jumps identified statically was effective. Stepping through the code clearly showed the comparison logic and the conditional jump execution. Runtime patching of the jump instruction in x64dbg successfully bypassed authentication (See example Listing ?? and context in Appendix Listing ??).
- **Virtualized Binaries:** Dynamic analysis faced significant hurdles.
 - *String Searching Failure:* Searching for relevant strings in memory during runtime often failed, similar to static analysis.
 - *Execution Flow Tracking Difficulty:* Stepping through the virtualized code sections was extremely difficult. The instruction pointer (RIP) often appeared to loop within small blocks or jump to seemingly random locations, consistent with execution being handled by the VM interpreter rather than direct native execution (See Listings ?? vs. ??, Appendix ??). Standard debugging techniques like setting breakpoints based on expected native instructions became unreliable.
 - *State Obfuscation:* Understanding the program's state (relevant variable values, comparison results) was hindered because the actual logic was executed within the VM's context, which was not directly visible or interpretable through the debugger's view of native registers and memory.

Dynamic bypass attempts by patching suspected native jump instructions (if any could be identified near the VM entry/exit) were unsuccessful, as the core logic resided within the VM's execution loop. The inability to trace execution flow coherently and the apparent looping or non-sequential jumps of the instruction pointer are characteristic of code execution being managed by an internal VM interpreter [29]. While x64dbg, as a dynamic tool, can disassemble the native instructions of the VxLang VM itself, it does not directly reveal the original

application logic, which has been translated into custom bytecode. The obfuscated state and hidden strings further complicated runtime understanding. Although debuggers can often resolve symbols for system DLLs loaded at runtime, internal application symbols within the virtualized segments remain obscured by VxLang, severely limiting the utility of dynamic analysis for understanding the protected code's semantics.

These results strongly indicate that VxLang's code virtualization effectively hinders both static and dynamic reverse engineering attempts using standard tools and techniques.

3) *Analysis of Lilith RAT:* Static and dynamic analysis were also performed on the Lilith RAT client (original vs. virtualized). Findings mirrored those from the authentication case studies:

- **Non-Virtualized:** Analysis was feasible. Strings related to commands and functionality were identifiable. Control flow for network communication and command handling could be traced using Ghidra and x64dbg, allowing potential understanding of its mechanisms (e.g., keylogging, remote execution).
- **Virtualized:** Analysis difficulty increased significantly. Ghidra failed to properly disassemble virtualized sections, showing numerous '???' entries and obscuring the logic. Dynamic tracing in x64dbg was severely hampered by the VM execution, making it hard to follow command processing or data flow.
- **Functional Integrity:** Importantly, functional testing confirmed that the virtualized Lilith client remained fully operational, successfully connecting to the server and executing core RAT commands, despite the code transformation.

This indicates VxLang's virtualization hinders analysis even for complex, potentially malicious software, without necessarily breaking its intended functionality.

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 I and Fig. 3, 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 II 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 I
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

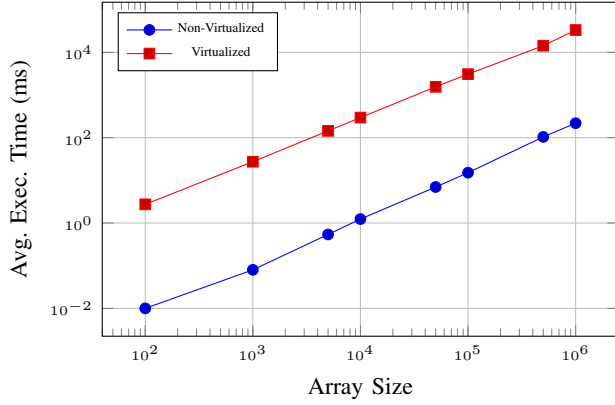


Fig. 3. Quick Sort Execution Time Comparison (Log-Log Scale).

TABLE II
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 III 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 III
EXECUTABLE FILE SIZE COMPARISON (KB)

Program	Non-Virtualized (KB)	Virtualized (KB)
<code>quick_sort</code>	98	1,537
<code>encryption</code>	110	1,507
<code>size</code>	97,771	112,324
<code>console</code>	92	1,577
<code>console_cloud</code>	281	1,695
<code>app_imgui</code>	1,675	2,330
<code>app_imgui_cloud</code>	1,860	2,418
<code>app_qt</code>	122	1,578
<code>app_qt_cloud</code>	315	1,671
<code>Lilith_Client</code>	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. Dynamic analysis was similarly obstructed by the VM execution model, making runtime tracing and manipulation arduous. Attempts to bypass authentication logic, which were trivial in non-virtualized versions, were successfully thwarted in the virtualized binaries using the employed techniques.

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. 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.