Implementasi dan Analisis Efektivitas Vxlang *Code Virtualization* dalam mempersulit *Reverse Engineering*

Seno Pamungkas Rahman, Dr. Ruki Harwahyu, S.T., M.T., M.Sc.

Departemen Teknik Elektro
Fakultas Teknik, Universitas Indonesia, Kampus UI Depok 16424, Jawa Barat, Indonesia

E-mail: seno.pamungkas@ui.ac.id

Abstrak

Rekayasa balik merupakan ancaman serius terhadap keamanan perangkat lunak, memungkinkan penyerang untuk menganalisis, memahami, dan memodifikasi kode program tanpa izin. Teknik *obfuscation*, terutama virtualisasi kode, menjadi solusi yang menjanjikan untuk melindungi perangkat lunak dari ancaman ini. Penelitian ini bertujuan untuk mengimplementasikan dan menganalisis efektivitas virtualisasi kode dalam meningkatkan keamanan perangkat lunak dengan mempersulit rekayasa balik. Penelitian ini menggunakan VxLang sebagai platform virtualisasi kode. Metode penelitian yang digunakan meliputi implementasi virtualisasi kode pada sebuah aplikasi studi kasus, kemudian dilakukan analisis statis dan dinamis terhadap aplikasi sebelum dan sesudah di-*obfuscate*. Analisis statis dilakukan dengan membandingkan tingkat kesulitan dalam memahami kode *assembly* yang dihasilkan. Analisis dinamis dilakukan dengan mengukur waktu eksekusi dan sumber daya yang digunakan oleh aplikasi. Hasil penelitian menunjukkan bahwa virtualisasi kode dengan VxLang efektif dalam meningkatkan keamanan perangkat lunak. Kode yang telah di-*obfuscate* menjadi lebih sulit dipahami dan dianalisis, terlihat dari meningkatnya kompleksitas kode *assembly*. Penelitian ini diharapkan dapat membuktikan bahwa virtualisasi kode dengan VxLang merupakan teknik yang efektif untuk melindungi perangkat lunak dari rekayasa balik dan dapat dipertimbangkan sebagai solusi untuk meningkatkan keamanan aplikasi.

Implementation and Analysis of the Effectiveness of Vxlang Code Virtualization in Complicating Reverse Engineering

Abstract

Reverse engineering is a serious threat to software security, allowing attackers to analyze, understand, and modify program code without permission. Obfuscation techniques, especially code virtualization, are promising solutions to protect software from this threat. This study aims to implement and analyze the effectiveness of code virtualization in improving software security by complicating reverse engineering. This study uses VxLang as a code virtualization platform. The research methods used include implementing code virtualization on a case study application, then conducting static and dynamic analysis of the application before and after obfuscation. Static analysis is done by comparing the level of difficulty in understanding the resulting assembly code. Dynamic analysis is done by measuring the execution time and resources used by the application. The results of the study show that code virtualization with VxLang is effective in improving software security. Obfuscated code becomes more difficult to understand and analyze, as seen from the increasing complexity of the assembly code. This study is expected to prove that code virtualization with VxLang is an effective technique to protect software from reverse engineering and can be considered as a solution to improve application security.

Keywords: Code Obfuscation, Code Virtualization, Software Protection, Reverse Engineering, Security Analysis, Performance Overhead

1. 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: The next section discusses related work in code obfuscation and virtualization. Then, the methodology employed in our experiments is detailed. Following that, the implementation details are briefly outlined. Subsequently, the experimental results for both security and performance analysis are presented and discussed. Finally, the paper concludes with a summary and suggestions for future research.

2. Literature Review and Theoretical Background

Reverse engineering is a fundamental process in software analysis aimed at understanding the internal mechanisms of a system without access to its original source code [1, 13]. This process poses a significant threat to software security as it can be exploited to uncover proprietary algorithms, identify security vulnerabilities, pirate licenses, and even inject malicious code [2]. This vulnerability has driven the development of various protection techniques, including code obfuscation.

2.1. Code Obfuscation Techniques

Code obfuscation aims to transform program code into a functionally equivalent form that is significantly more difficult for humans to understand and analyze [4]. The goal is not to make reverse engineering impossible, but to increase the complexity and cost required, making it impractical for attackers. Obfuscation techniques can be classified based on the level of abstraction at which they are applied:

2.1.1 Source Code Obfuscation

Modifications are made to human-readable source code.

- **Layout Obfuscation:** Alters code appearance, such as scrambling variable and function names [14], and removing whitespace and comments [15]. Provides minimal security against automated analysis.

- **Data Obfuscation:** Hides data representation, for example, through string encoding [16, 17, 18], instruction substitution [19, 20], or using mixed boolean-arithmetic expressions [21, 22, 23] to disguise data manipulation logic.
- **Control Flow Obfuscation:** Modifies the program's execution path logic. Examples include inserting bogus control flow [24], using opaque predicates [25], and control flow flattening, which transforms structured code into a large and complex `switch` statement [26].

2.1.2 Bytecode Obfuscation

This technique targets intermediate code such as Java bytecode, .NET CIL, or LLVM IR. Methods include identifier renaming, control flow obfuscation, string encryption, and inserting dummy code [27, 28]. Effective in complicating decompilation back to high-level source code.

2.1.3 Binary Code Obfuscation

Operates directly on machine-executable code.

- **Code Packing/Encryption:** Compresses or encrypts the original code, requiring a runtime stub to unpack/decrypt the code before execution [29]. Primarily hinders static analysis, but the original code is revealed in memory during execution.
- **Control Flow Manipulation:** Uses indirect jumps/calls, modifies `call`/`ret` instructions, or breaks code into small blocks with jumps to disrupt linear disassembly and analysis [29].
- **Constant Obfuscation:** Hides constant values through arithmetic or logical operations [29].
- **Code Virtualization:** Considered one of the strongest binary obfuscation techniques, discussed further below.

2.2. Code Virtualization (VM-Based Obfuscation)

Code virtualization, or Virtual Machine (VM)-based obfuscation, is an advanced technique where segments of native machine code are translated into a custom bytecode format. This bytecode is then executed by a specially designed VM embedded directly into the application [5, 6]. As illustrated in Figure **??** from Oreans' research [5], this process transforms the original code into a series of virtual instructions.

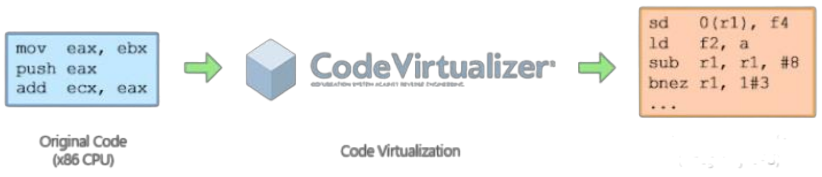

Figure 1: Code Virtualization Process [5]

The abstraction layer introduced by this VM creates a significant barrier for reverse engineers. Standard analysis tools cannot directly interpret the unique bytecode ISA [8]. Reverse engineers must first understand the VM's architecture, virtual instruction handler implementations, and bytecode mapping, 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 reuse of analysis results. Oreans [5] highlights the possibility of generating diverse VMs for different protected copies of an application.

- **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. The execution flow of code virtualization can be seen in Figure 2 (adapted from [7]).

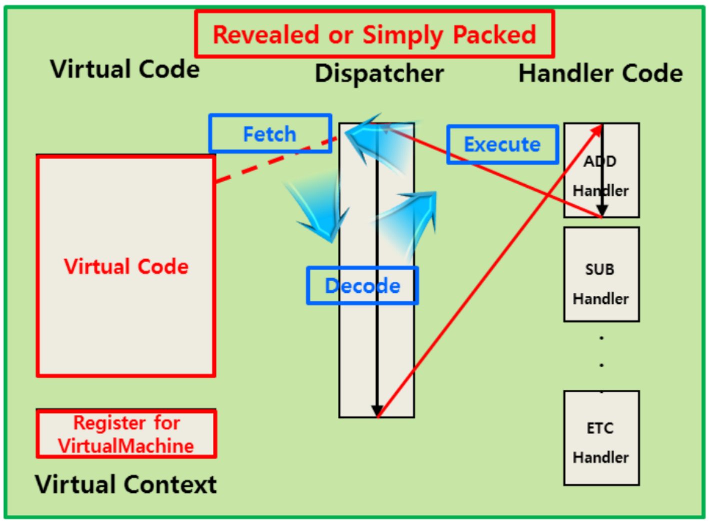

Figure 2: Code Virtualization Execution Flow (adapted from [7])

- **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 [30] and Themida [31] (which also includes virtualization features beyond basic packing) employ code virtualization. Academic research has also explored techniques such as symbolic deobfuscation to analyze virtualized code [8] and methods to enhance virtualization robustness, such as virtual code folding [7].

2.3. 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 [32].

2.3.1 Static Disassemblers

Static disassemblers, such as Ghidra [11] and IDA Pro [33], analyze executable files without running them. They employ techniques like linear sweep or recursive traversal to identify instruction sequences [34, 35]. 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 [32, 36]. Crucially, when faced with custom bytecode from a VM, static disassemblers designed for standard ISAs (e.g., x86-64) cannot correctly interpret these non-native instructions, leading to analysis failure.

2.3.2 Dynamic Disassemblers

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 [32]. 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.

2.4. 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, thereby contributing practical insights into its utility as a software protection mechanism. Unlike analyzing established commercial protectors, this research focuses on the specific implementation and impact of the VxLang framework.

3. Research Design and Methodology

This research employs a quantitative experimental approach to evaluate the effectiveness of code virtualization using VxLang in complicating reverse engineering efforts and to analyze the performance overhead it incurs.

3.1. Experimental Design

The experimental design used is a comparison between a control group and an experimental group.

- **Control Group:** Case study applications and benchmarks compiled normally without the application of VxLang code virtualization.
- **Experimental Group:** The same applications and benchmarks, but with critical code sections processed using VxLang code virtualization.

The independent variable is the application of VxLang code virtualization, while the dependent variables include the difficulty level of reverse engineering (static and dynamic analysis) and performance impact (execution time and file size).

3.2. Study Objects

The study objects in this research include:

1. **Authentication Case Study Applications:** User login simulation applications were developed in several interface variants (Console, Qt, Dear ImGui) and two authentication mechanisms (hardcoded credentials and cloud-based validation). These applications serve as the primary targets for reverse engineering analysis.
2. **Performance Benchmark Applications:** Applications specifically designed to measure the performance impact on specific computational tasks, namely the QuickSort algorithm and AES-CBC-256 encryption. A minimal application with embedded data was also used to assess the baseline file size increase.

3. **Remote Administration Tool (RAT) Case Study:** The client component of Lilith RAT [37], an open-source RAT, was analyzed for post-virtualization functionality and changes in detection profiles on VirusTotal. Nine additional malware/PUA samples were also analyzed for their impact on VirusTotal detection.

3.3. Research Instruments and Materials

This research utilizes the following instruments:

- **Hardware:** Windows 11 (64-bit) based PC.
- **Development Software:** Clang/clang-cl (C++17), CMake, Ninja, VxLang SDK, Qt 6, Dear ImGui, OpenSSL 3.x, libcurl.
- **Analysis Tools:** Ghidra (v11.x) for static analysis, x64dbg (latest release) for dynamic analysis.
- **Performance Measurement:** C++ `std::chrono` library for time, `std::filesystem::file_size` for file size.

3.4. Data Collection Procedure

Data collection was carried out through several main stages:

3.4.1 Artifact Preparation

The case study applications and benchmarks were compiled in two versions: original (without virtualization) and intermediate (with VxLang macros and linked VxLang library). The intermediate version was then processed using the `vxlang.exe` command-line tool to generate the final virtualized executable (`*_vxm.exe`).

3.4.2 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, and attempt static patching to bypass logic. Systematic observations were recorded.
2. **Dynamic Analysis (x64dbg):** Run executable under debugger, search for strings/patterns at runtime, set breakpoints at suspected logic locations, step through execution, observe register/memory values, and attempt runtime manipulation (patching conditional jumps) to bypass authentication. Systematic observations were recorded.

3.4.3 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`.

3.4.4 Lilith RAT and Other Malware Samples Analysis

1. **Functional Integrity Testing (Lilith RAT):** 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.

2. **VirusTotal Analysis:** Both original and virtualized executables of the Lilith RAT client and nine other malware/PUA samples were submitted to VirusTotal to compare detection rates and threat characterizations.

3.5. 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.

- **Performance Data:** Calculation of descriptive statistics, percentage overhead for execution time, throughput calculation (MB/s), and percentage increase in file size. Comparative tables and graphs will be used.

- **Trade-off Analysis:** Synthesis of security findings and performance results to evaluate the balance between protection enhancement and performance/size costs introduced by VxLang.

4. Results and Discussion

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

4.1. 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.

4.1.1 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. In non-virtualized binaries, standard comparison instructions and conditional jumps controlling authentication were readily found, making static patching feasible. For instance, in the non-virtualized *app_imgui* application, disassembly analysis revealed password comparison and a `JNZ` instruction leading to a failure block if the password was incorrect. This instruction could be changed to `JZ` to invert the logic. Furthermore, credential strings "seno" and "rahman" were found in the defined data section, indicating vulnerable hard-coded storage. For the cloud version, although credentials were not hard-coded, the client-side logic checking the server's response could still be identified and manipulated by patching conditional jump instructions.

Conversely, static analysis proved significantly more challenging for binaries processed by VxLang. Table 1 summarizes key metrics from Ghidra analysis for representative applications, illustrating the impact of virtualization.

Table 1: 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%**

The data in Table 1 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 bytecode ISAs introduced by virtualization [32, 34, 35]. 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.

4.1.2 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 2.

Table 2: 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%**	-

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 2) 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 2, "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 [32, 5]. 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].

4.1.3 Analysis of Potentially Malicious Software and VirusTotal Detection

To evaluate VxLang's impact on more complex software and automated detection, ten software samples, including the Lilith RAT client [37] and nine other publicly available malware/PUA samples (*Al-Khaser, donut, DripLoader, FilelessPELoader, JuicyPotato, ParadoxiaClient, PELoader, RunPE-In-Memory, SigLoader*), were analyzed. For the Lilith RAT, after careful, iterative placement of VxLang macros targeting critical functions, functional testing confirmed that the virtualized client remained fully operational, successfully connecting to its server and executing remote commands. This demonstrated VxLang's capability to preserve core functionalities in complex applications if applied precisely.

All ten samples, in their original and VxLang-virtualized forms, were submitted to VirusTotal (72 AV engines). The results (summarized in Table 3) showed a varied impact on detection rates. For five samples, including Lilith RAT (22 to 18 detections) and *donut* (30 to 19 detections), virtualization led to a decrease in detections, often accompanied by a shift from specific threat labels to more generic ones or AI/heuristic-based flags. This suggests successful obfuscation of static signatures.

However, for the other five samples, such as *JuicyPotato* (9 to 20 detections) and *FilelessPELoader* (16 to 21 detections), virtualization resulted in an *increase* in detections. This indicates that the virtualization layer itself or its artifacts can trigger suspicion from some AV engines, potentially being flagged as packed or protected software common in malware. Overall, the average change in detection across all ten samples was minimal (+0.4 detections), highlighting that while VxLang can alter detection profiles, it does not guarantee evasion and its effect is sample-dependent.

Table 3: VirusTotal Detection Count Comparison for Various Samples (Non-VM vs. VM from 72 Engines)

Malware/Application	**Non-VM Detections**	**VM Detections**	**Change in Detections**
Lilith_Client	22	18	**-4**
Al-Khaser	19	15	**-4**
donut	30	19	**-11**
DripLoader	17	16	**-1**
FilelessPELoader	16	21	**+5**
JuicyPotato	9	20	**+11**
ParadoxiaClient	17	16	**-1**
PELoader	14	17	**+3**
RunPE-In-Memory	12	16	**+4**
SigLoader	16	17	**+1**
	Average Change in Detections		**+0.4**

4.2. Performance and Size Overhead Results

4.2.1 Execution Time Overhead

The performance impact was measured using QuickSort and AES benchmarks.

- **QuickSort:** As shown in Table 4 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.

Table 4: 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

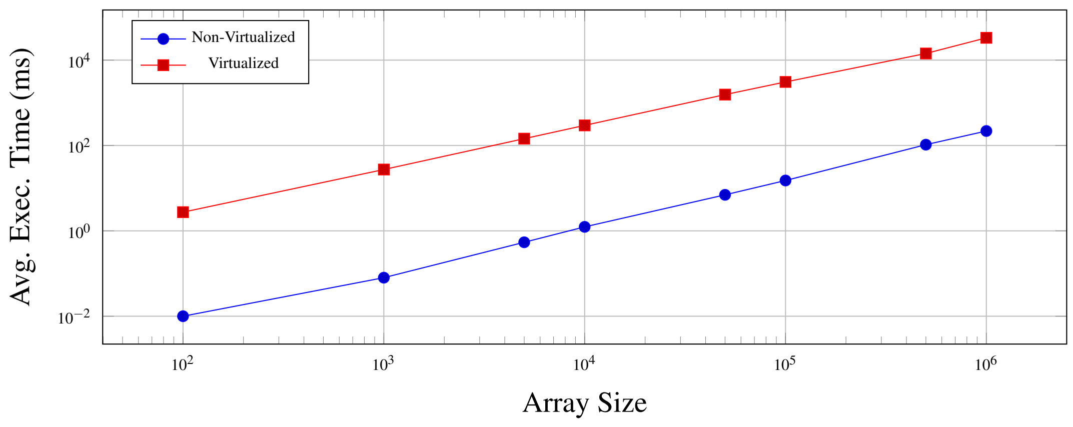

Figure 3: Quick Sort Execution Time Comparison (Log-Log Scale).

- **AES Encryption:** Table 5 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 5: 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

4.2.2 File Size Overhead

Table 6 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 6: 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

4.3. 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 [34, 35], and significantly complicates dynamic analysis with tools like x64dbg, as the underlying logic is executed by an opaque VM [32]. 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 on ten malware/PUA samples revealed a nuanced impact: while VxLang could reduce signature-based detections for some samples, it led to increased detections for others, likely due to the virtualization layer itself being flagged. This highlights a complex interplay between obfuscation and evolving AV detection heuristics [5, 8, 29].

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, **provided that macro placement is done carefully and iteratively to avoid functional disruption, especially in code sections with complex control flows or I/O operations**. However, the severe performance cost necessitates a strategic, selective application, targeting only critical sections. The VirusTotal results also imply that while detection profiles are altered, evasion is not guaranteed, as heuristic/AI methods or flags for protected software can still lead to detection, and in some cases, increase it. 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.

5. Conclusion

This study investigated the effectiveness of code virtualization using the VxLang framework to complicate software reverse engineering. Implementation involved marking source code, compiling intermediate executables, and processing them with the VxLang tool to generate virtualized binaries.

Experimental analysis demonstrated that VxLang's code virtualization significantly increases the difficulty of reverse engineering. Static analysis using Ghidra on virtualized code failed to identify meaningful instructions, functions, or data structures. Similarly, dynamic analysis with x64dbg was hampered by obfuscated control flow and the virtual machine's (VM) execution model, which obscured runtime behavior and debugging attempts. Efforts to bypass authentication logic, trivial in non-virtualized versions, were successfully thwarted in VxLang-protected binaries. However, effective implementation requires careful and iterative placement of virtualization macros, as improper placement, especially in code with I/O or complex control flows, can disrupt application functionality, as observed in the Lilith RAT case study.

Analysis of ten malware/PUA samples on VirusTotal showed varied impacts of VxLang on detection rates: approximately half of the samples exhibited a decrease in detections, often with a shift to generic/heuristic flags, while the remainder showed an increase in detections, indicating the virtualization layer itself can trigger alerts.

This enhanced security comes with substantial performance overhead, observed in QuickSort algorithm and AES encryption benchmarks, along with a significant increase in executable file size. The findings underscore a clear trade-off: strong protection against reverse engineering at the cost of performance degradation and increased file size. Therefore, practical application of VxLang should likely be selective, targeting only the most critical and sensitive code sections.

Future research could focus on exploring more advanced reverse engineering techniques against VM-based protections, investigating VxLang configuration options for security-performance balance, and comparative studies with other virtualization solutions. Deeper analysis of VxLang's interaction with various malware types and antivirus detection techniques would also yield valuable insights.

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.S. Kurukkal, and H.G. Lemu. “Reverse engineering in medical application: literature review, proof of concept and future perspectives.” In: *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. (Visited on 12/18/2024).

[4] H Jin, J Lee, S Yang, K Kim, and D.H. Lee. “A Framework to Quantify the Quality of Source Code Obfuscation”. In: *A Framework to Quantify the Quality of Source Code Obfuscation* 12 (2024), p. 14.

[5] Oreans. *Code Virtualizer*. `https://www.oreans.com/CodeVirtualizer.php`. 2006. (Visited on 11/11/2024).

[6] Zhoukai Wang, Zuoyan Xu, Yaling Zhang, Xin Song, and Yichuan Wang. “Research on Code Virtualization Methods for Cloud Applications”. In: 2024.

[7] Dong Hoon Lee. “VCF: Virtual Code Folding to Enhance Virtualization Obfuscation”. In: *VCF: Virtual Code Folding to Enhance Virtualization Obfuscation* (2020).

[8] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. “Symbolic deobfuscation: from virtualized code back to the original”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC ’18)*. Pau, France: ACM, 2018, pp. 1516–1523.

[9] Hackcyom. *Hackcyom*. `https://www.hackcyom.com/2024/09/vm-obfuscation-overview/`. 2024. (Visited on 11/22/2024).

[10] *VxLang Documentation*. `https://vxlang.github.io/`. 2025. (Visited on 03/17/2025).

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

[12] Duncan Ogilvie. *x64dbg*. `https://x64dbg.com`. 2014. (Visited on 11/11/2024).

[13] geeksforgeeks. *Reverse Engineering - Software Engineering*. `https://www.geeksforgeeks.org/software-engineering-reverse-engineering/`. 2024. (Visited on 11/04/2024).

[14] J. T. Chan and W Yang. “Advanced obfuscation techniques for Java bytecode”. In: *Advanced obfuscation techniques for Java bytecode* 71.1-2 (2004), pp. 1–10.

[15] 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.

[16] L. Ertaul and S. Venkatesh. “Novel obfuscation algorithms for software security”. In: *International Conference on Software Engineering Research and Practice*. Las Vegas, 2005.

[17] K. Fukushima, S. Kiyomoto, T. Tanaka, and K Sakurai. “Analysis of program obfuscation schemes with variable encoding technique”. In: *Analysis of program obfuscation schemes with variable encoding technique* 91.1 (2008), pp. 316–329.

[18] A Kovacheva. “Efficient code obfuscation for Android”. In: *Advances in Information Technology*. Bangkok, 2013.

[19] C. LeDoux, M. Sharkey, B. Primeaux, and C Miles. “Instruction embedding for improved obfuscation”. In: *Annual Southeast Regional Conference*. Tuscaloosa, 2012.

[20] S.M. Darwish, S.K. Guirguis, and M.S Zalat. “Stealthy code obfuscation technique for software security”. In: *International Conference on Computer Engineering & Systems*. Cairo, 2010.

[21] 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.

[22] M. Schloegel et al. “Hardening code obfuscation against automated attacks”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, 2022.

[23] Y. Zhou, A. Main, Y.X. Gu, and H Johnson. “Information hiding in software with mixed boolean-arithmetic transforms”. In: *International Workshop on Information Security Applications*. Jeju Island, 2007.

[24] 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.

[25] 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.

[26] T. László and Á Kiss. “Obfuscating C++ programs via control flow flattening”. In: *Obfuscating C++ programs via control flow flattening* 30 (2009), pp. 3–19.

[27] Pierre Parrend. *Bytecode Obfuscation*. `https://owasp.org/www-community/controls/Bytecode_obfuscation`. 2018. (Visited on 12/25/2024).

[28] Yakov. *Using LLVM to Obfuscate Your Code During Compilation*. `https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation`. 2020. (Visited on 12/20/2024).

[29] Roundy, Kevin A, Miller, and Barton P. “Binary-code obfuscations in prevalent packer tools”. In: *ACM Computing Surveys (CSUR)* 46.1 (2013), pp. 1–32.

[30] VMProtect Software. *VMProtect*. `https://vmpsoft.com/vmprotect/overview`. (Visited on 11/23/2024).

[31] Oreans. *Themida*. `https://www.oreans.com/Themida.php`. (Visited on 11/23/2024).

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

[33] Hex-Rays. *IDA Pro*. `https://hex-rays.com/ida-pro`. 1991. (Visited on 11/22/2024).

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

[35] Chung-Kil Ko and Johannes 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.

[36] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten 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.

[37] werkamsus. *Lilith - Free & Native Open Source C++ Remote Administration Tool for Windows*. `https://github.com/werkamsus/Lilith`. Accessed: 8 May 2025. 2017.