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. 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.

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.

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 [29] and Themida [30] (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.

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`) were then processed offline by the external VxLang command-line tool. This tool reads a JSON configuration file specifying input/output paths and virtualization options (e.g., virtualizing the entry point). The tool modifies the intermediate executable in-place or creates a new output file, replacing the native code within the marked sections with its virtualized bytecode and embedding the necessary VM runtime. The resulting file is the final virtualized executable used for testing. A simplified JSON configuration might look like:

```
{
  "Input": "path/to/intermediate_app_vm.exe",
  "Output": "path/to/final_virtualized_app.exe",
  "Virtualizer": {
    "EntryPoint": false
  },
  "Obfuscator": {
    "EntryPoint": false
  },
  "Packer": {
    "Enable": false
  }
  // Other options omitted for brevity
}
```

For this study, default virtualization settings were primarily used after marking the code sections.

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 was straightforward. Relevant strings (e.g., "Authentication Failed", hardcoded "seno", "rahman") were easily located in the '.rdata' or '.text' sections. Cross-references led directly to the authentication functions. Disassembly clearly showed standard comparison instructions (`memcmp`, `strcmp`, or C++ string comparison operators) followed by conditional jumps (`JNZ`, `JE`) controlling the flow based on the comparison result (See example Listing **??** and context in Appendix Listing **??**). For cloud variants, the call to the request function and the subsequent check on its return value were identifiable (Listing **??**, Appendix **??**). Static patching of the conditional jump instruction to force the "authorized" path was trivial in all non-virtualized cases.

- **Virtualized Binaries:** Static analysis proved significantly more challenging.

  - *Instruction Recognition Failure:* Ghidra consistently failed to recognize standard x86-64 instructions within the virtualized sections, reporting large blocks of unknown opcodes or '???' entries. Analysis summaries showed a drastic reduction (often to zero) in the count of recognized instructions and functions compared to the originals (e.g., Fig. 1 vs. Fig. 2 from the thesis).

  - *Data Obfuscation:* Critical strings like "Authentication Failed" or hardcoded credentials were no longer present as plain text in the binary, preventing
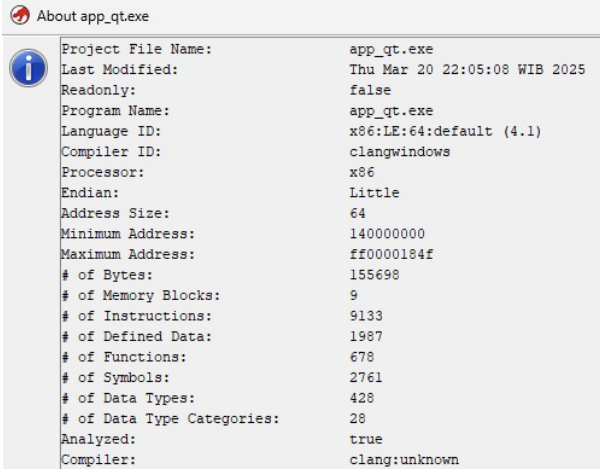
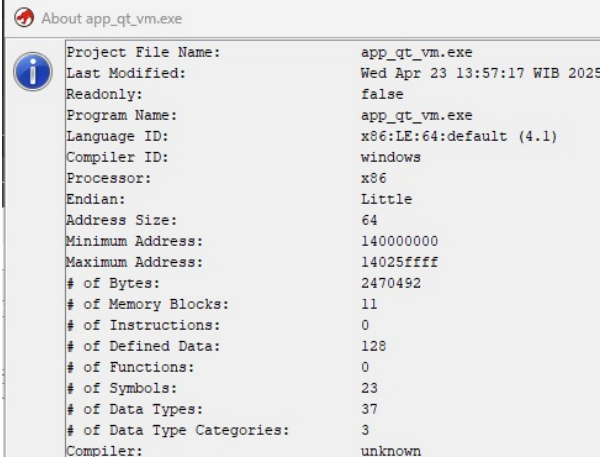Fig. 1. Ghidra Analysis Summary for `app_qt.exe` (Non-Virtualized).

Fig. 2. Ghidra Analysis Summary for `app_qt_vm.exe` (Virtualized).

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	5.15	0.42
1,000	0.09	0.00	53.08	5.52
5,000	0.61	0.09	300.64	26.13
10,000	1.38	0.22	585.70	79.88
50,000	8.45	0.73	3,434.32	592.92
100,000	17.86	1.42	6,771.09	553.09
500,000	106.16	5.61	29,698.73	3,518.63
1,000,000	216.59	9.49	45,186.90	6,198.38

simple string searches from locating the relevant code sections. Symbol information was also heavily stripped.

  – *Control Flow Obscurity:* The clear structure of conditional checks and jumps seen in the original code was replaced by opaque sequences of seemingly unrelated or unknown instructions, making it impossible to identify the core authentication logic or the specific conditional jump to patch statically. The control flow graph became fragmented and uninformative within the virtualized regions.

Static bypass attempts were unsuccessful due to the inability to locate and understand the relevant control flow logic.

2) Dynamic Analysis (x64dbg):

- **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.

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

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 51,400% for 100 elements to about 20,860% for 1,000,000 elements. 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 1GB of data increased by approximately
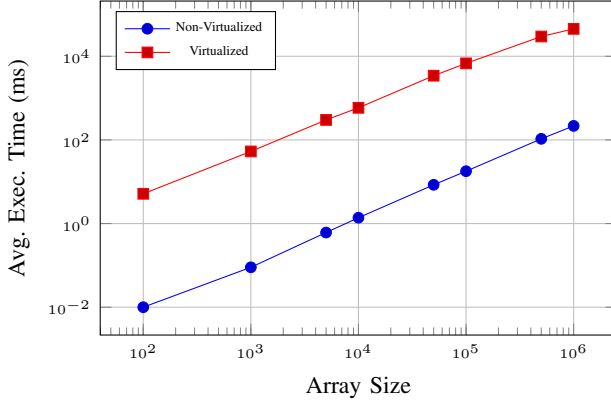
Fig. 3. Quick Sort Execution Time Comparison (Log-Log Scale).

TABLE II
AES-256-CBC PERFORMANCE RESULTS (1GB DATA)

Metric	Non-Virtualized	Virtualized
Total Encryption Time (ms)	2,722.96	13,193.51
Total Decryption Time (ms)	2,055.01	12,529.90
Avg. Encrypt Time/Block (ms)	0.00272	0.01319
Avg. Decrypt Time/Block (ms)	0.00206	0.01253
Encrypt Throughput (MB/s)	358.64	74.02
Decrypt Throughput (MB/s)	475.21	77.94
Combined Throughput (MB/s)	416.92	75.98

TABLE III
EXECUTABLE FILE SIZE COMPARISON (KB)

Program	Non-Virtualized (KB)	Virtualized (KB)
quick_sort	119	1,951
encryption	131	1,834
size	97,802	112,716
console	105	1,942
app_imgui	1,773	2,753
app_qt	145	1,954

384%, and decryption time increased by about 510%. Consequently, the combined throughput dropped dramatically from 416.92 MB/s to 75.98 MB/s (an 82% reduction). This confirms a significant overhead for cryptographic operations.

2) File Size Overhead: Table III shows a consistent increase in executable file size after virtualization. For smaller programs (console, QuickSort, AES), the size increased by over 15-18 times (from ~100-130 KB to ~1.8-1.9 MB). For larger GUI applications (`app_imgui`, `app_qt`) and the benchmark with embedded data (`size`), the relative increase was smaller but still significant (e.g., `app_imgui` increased by about 55%, `size` by 15%). This overhead is primarily attributed to the inclusion of the VxLang VM runtime and the bytecode representation of the original code.

C. Discussion

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

Security Enhancement: VxLang provides a substantial barrier against common reverse engineering techniques. The transformation of native code into an interpreted bytecode format effectively neutralizes standard static analysis tools like Ghidra and significantly complicates dynamic analysis with debuggers like x64dbg. The obfuscation of control flow and data within the virtualized sections makes identifying and manipulating critical logic, such as authentication checks, extremely challenging compared to non-virtualized code. This aligns with the expected security benefits of VM-based obfuscation [5], [8].

Performance Cost: The security benefits come at a steep price in terms of performance. The interpretation overhead introduced by the VM significantly slows down the execution of virtualized code, particularly for computationally intensive tasks like sorting or cryptographic operations. The observed overhead percentages (often exceeding several thousand percent for QuickSort) highlight that virtualizing performance-critical loops or frequently called functions can severely impact application responsiveness.

Size Increase: The considerable increase in file size, especially for smaller applications, is another factor to consider. This is primarily due to the embedded VM runtime. While less critical than performance for many desktop applications, it might be a concern for resource-constrained environments or distribution size limits.

Practical Implications: VxLang appears to be a potent tool for protecting highly sensitive code segments where security is paramount and performance impact on those specific segments is acceptable or manageable. Examples include anti-tamper mechanisms, license validation routines, or core algorithmic intellectual property. However, applying it indiscriminately to large portions of an application, especially performance-sensitive areas, is likely impractical due to the severe execution speed degradation. Developers must strategically select which code sections warrant the high level of protection (and associated cost) offered by virtualization. The choice between hardcoded and cloud-based authentication showed that while cloud validation removes credentials from the client, the client-side logic handling the *result* of the validation remains a viable target for bypass if not protected, reinforcing the need for techniques like virtualization on critical client-side checks.

VI. CONCLUSION

This paper investigated the effectiveness of code virtualization using the VxLang framework as a technique to mitigate software reverse engineering. 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.

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 on the state-of-the-art in VM-based obfuscation. Analyzing the security of the VxLang VM interpreter itself could also reveal potential vulnerabilities.

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

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