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. Pendahuluan

Perkembangan pesat teknologi perangkat lunak telah mendorong terciptanya aplikasi yang semakin kompleks, namun diiringi oleh peningkatan ancaman keamanan. Rekayasa balik (*reverse engineering*), proses menganalisis perangkat lunak untuk memahami cara kerjanya tanpa akses ke kode sumber [1], merupakan kerentanan kritis. Penyerang memanfaatkan rekayasa balik untuk mengungkap algoritma, menemukan celah keamanan, membajak perangkat lunak, dan menyisipkan kode berbahaya [2]. Ukuran keamanan tradisional seringkali tidak memadai terhadap analisis logika program saat berjalan [3].

Untuk mengatasi ancaman ini, teknik pengaburan kode (*code obfuscation*) bertujuan mengubah kode program menjadi bentuk yang lebih sulit dipahami tanpa mengubah fungsionalitasnya [4]. Di antara berbagai strategi pengaburan, virtualisasi kode (*code virtualization*) menonjol sebagai pendekatan yang sangat kuat [5, 6]. Teknik ini menerjemahkan kode mesin asli menjadi instruksi *bytecode* khusus yang dieksekusi oleh mesin virtual (VM) yang tertanam dalam aplikasi [7]. Arsitektur Set Instruksi (ISA) unik dari VM ini membuat alat rekayasa balik konvensional seperti *disassembler* dan *debugger* menjadi tidak efektif karena tidak dapat langsung menginterpretasikan kode yang divirtualisasi [8]. Penyerang harus terlebih dahulu menguraikan arsitektur VM dan *bytecode*-nya, yang secara substansial meningkatkan upaya dan kompleksitas analisis [9].

VxLang adalah sebuah kerangka kerja proteksi kode yang menggabungkan kapabilitas virtualisasi kode, menargetkan *executable* Windows PE [10]. Kerangka kerja ini menyediakan mekanisme untuk mentransformasi kode asli menjadi format *bytecode* internalnya, yang dieksekusi oleh VM yang tersemat. Memahami efektivitas praktis dan biaya terkait dari alat semacam itu sangat penting bagi pengembang yang mencari solusi proteksi perangkat lunak yang tangguh.

Makalah ini menyelidiki efektivitas virtualisasi kode menggunakan VxLang dalam mengurangi upaya rekayasa balik. Kami bertujuan untuk menjawab pertanyaan kunci berikut:

1. Seberapa efektif virtualisasi kode VxLang mengaburkan logika program terhadap teknik rekayasa balik statis dan dinamis?

2. Apa dampak kuantitatif dari virtualisasi VxLang terhadap performa aplikasi (waktu eksekusi) dan ukuran berkas?

Untuk menjawab pertanyaan-pertanyaan ini, kami mengimplementasikan virtualisasi VxLang pada fungsi-fungsi terpilih dalam aplikasi studi kasus (mensimulasikan autentikasi) dan *benchmark* performa (QuickSort, enkripsi AES). Kami kemudian melakukan analisis statis komparatif (menggunakan Ghidra [11]) dan analisis dinamis (menggunakan x64dbg [12]) pada biner asli dan yang telah divirtualisasi. *Overhead* performa diukur dengan membandingkan waktu eksekusi dan ukuran *executable*, dan dampak pada alat deteksi *malware* otomatis dinilai menggunakan analisis VirusTotal pada studi kasus yang relevan.

Kontribusi utama dari penelitian ini adalah:

- Implementasi dan evaluasi praktis virtualisasi kode VxLang pada segmen kode representatif.
- Analisis kualitatif dan kuantitatif terhadap peningkatan kesulitan yang ditimbulkan pada rekayasa balik statis dan dinamis oleh VxLang.
- Pengukuran dan analisis terhadap *overhead* performa dan ukuran berkas yang terkait dengan virtualisasi VxLang.
- Penilaian empiris terhadap *trade-off* keamanan-performa yang ditawarkan oleh kerangka kerja VxLang.

Sisa dari makalah ini diorganisasikan sebagai berikut: Bagian selanjutnya membahas penelitian terkait dalam pengaburan dan virtualisasi kode. Kemudian, metodologi yang digunakan dalam eksperimen kami dirinci. Setelah itu, detail implementasi diuraikan secara singkat. Selanjutnya, hasil eksperimental untuk analisis keamanan dan performa disajikan dan dibahas. Akhirnya, makalah ini ditutup dengan kesimpulan dan saran untuk penelitian di masa mendatang.

2. Tinjauan Pustaka dan Landasan Teori

Rekayasa balik (*reverse engineering*) merupakan proses fundamental dalam analisis perangkat lunak yang bertujuan untuk memahami mekanisme internal suatu sistem tanpa akses ke kode sumber aslinya [1, 13]. Proses ini menjadi ancaman signifikan terhadap keamanan perangkat lunak karena dapat dimanfaatkan untuk mengungkap algoritma propietari, mengidentifikasi kerentanan keamanan, membajak lisensi, hingga menyisipkan kode berbahaya [2]. Kelemahan ini mendorong pengembangan berbagai teknik proteksi, di antaranya adalah pengaburan kode (*code obfuscation*).

2.1. Teknik Pengaburan Kode (*Code Obfuscation*)

Pengaburan kode bertujuan untuk mentransformasi kode program ke dalam bentuk yang secara fungsional ekivalen namun secara signifikan lebih sulit untuk dipahami dan dianalisis oleh manusia [4]. Tujuannya bukan untuk membuat rekayasa balik mustahil, melainkan untuk meningkatkan kompleksitas dan biaya yang dibutuhkan sehingga menjadi tidak praktis bagi penyerang. Teknik pengaburan dapat diklasifikasikan berdasarkan level abstraksi di mana ia diterapkan:

2.1.1 Pengaburan Kode Sumber (*Source Code Obfuscation*)

Modifikasi dilakukan pada kode sumber yang dapat dibaca manusia.

- **Pengaburan Tata Letak (*Layout Obfuscation*):** Mengubah tampilan kode, seperti mengacak nama variabel dan fungsi [14], serta menghapus spasi putih dan komentar [15]. Memberikan tingkat keamanan minimal terhadap analisis otomatis.

- **Pengaburan Data (*Data Obfuscation*):** Menyembunyikan representasi data, misalnya melalui enkoding *string* [16, 17, 18], substitusi instruksi [19, 20], atau penggunaan ekspresi aritmetika-boolean campuran (*mixed boolean-arithmetic*) [21, 22, 23] untuk menyamarkan logika manipulasi data.

- **Pengaburan Alur Kontrol (*Control Flow Obfuscation*):** Memodifikasi logika alur eksekusi program. Contohnya termasuk penyisipan alur kontrol palsu (*bogus control flow*) [24], penggunaan predikat buram (*opaque predicates*) [25], dan perataan alur kontrol (*control flow flattening*) yang mengubah struktur kode menjadi pernyataan `switch` besar dan kompleks [26].

2.1.2 Pengaburan *Bytecode* (*Bytecode Obfuscation*)

Teknik ini menargetkan kode perantara seperti Java *bytecode*, .NET CIL, atau LLVM IR. Metode yang digunakan meliputi penggantian nama pengenal, pengaburan alur kontrol, enkripsi *string*, dan penyisipan kode semu (*dummy code*) [27, 28]. Efektif untuk mempersulit dekompilasi kembali ke kode sumber tingkat tinggi.

2.1.3 Pengaburan Kode Biner (*Binary Code Obfuscation*)

Beroperasi langsung pada kode mesin yang dapat dieksekusi.

- **Pengepakan/Enkripsi Kode (*Code Packing Encryption*):** Mengompresi atau mengenkripsi kode asli, memerlukan *stub* runtime untuk membongkar/mendekripsi kode sebelum eksekusi [29]. Utama menghambat analisis statis, namun kode asli akan terungkap di memori saat eksekusi.

- **Manipulasi Alur Kontrol:** Menggunakan lompatan/panggilan tidak langsung, memodifikasi instruksi `call`/`ret`, atau memecah kode menjadi blok-blok kecil dengan lompatan untuk mengganggu analisis dan *disassembly* linier [29].

- **Pengaburan Konstanta:** Menyembunyikan nilai-nilai konstan melalui operasi aritmetika atau logika [29].

- **Virtualisasi Kode (*Code Virtualization*):** Dianggap sebagai salah satu teknik pengaburan biner terkuat, yang akan dibahas lebih lanjut.

2.2. Virtualisasi Kode (*Code Virtualization*)

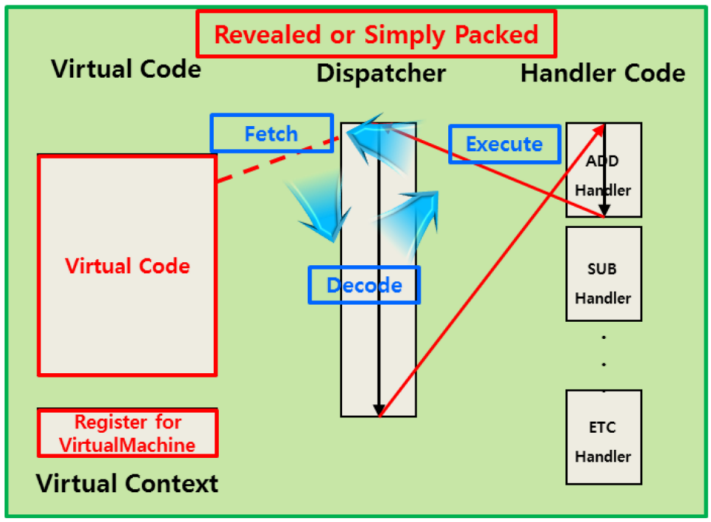
Virtualisasi kode, atau pengaburan berbasis Mesin Virtual (VM), adalah teknik canggih di mana segmen kode mesin asli diterjemahkan menjadi format *bytecode* khusus. *Bytecode* ini kemudian dieksekusi oleh sebuah VM yang dirancang khusus dan disematkan langsung ke dalam aplikasi [5, 6]. Seperti yang diilustrasikan pada Gambar **??** dari penelitian Oreans [5], proses ini mengubah kode asli menjadi serangkaian instruksi virtual.

Gambar 1: Proses Virtualisasi Kode [5]

Lapisan abstraksi yang diperkenalkan oleh VM ini menciptakan penghalang signifikan bagi perekayasa balik. Alat analisis standar tidak dapat secara langsung menginterpretasikan ISA *bytecode* yang unik tersebut [8]. Perekayasa balik harus terlebih dahulu memahami arsitektur VM, implementasi *handler* instruksi virtual, dan pemetaan *bytecode*, sebuah tugas yang kompleks dan memakan waktu [7, 9].

Beberapa aspek kunci dari pengaburan berbasis VM meliputi:

- **ISA Kustom:** Setiap aplikasi yang dilindungi berpotensi memiliki set instruksi virtual yang unik atau termutasi, mempersulit deteksi berbasis *signature* atau penggunaan ulang hasil analisis. Oreans [5] menyoroti kemungkinan untuk menghasilkan VM yang beragam untuk salinan aplikasi yang berbeda yang dilindungi.
- **Arsitektur VM:** Komponen VM tipikal mencakup unit *fetch, decode, dispatch*, dan *handler*, meniru operasi CPU namun diimplementasikan dalam perangkat lunak [8, 9]. Kompleksitas dan detail implementasi dari *handler-handler* ini secara langsung memengaruhi keamanan dan performa. Alur eksekusi virtualisasi kode dapat dilihat pada Gambar 2 (diadaptasi dari [7]).



Gambar 2: Alur Eksekusi Virtualisasi Kode (diadaptasi dari [7])

- ***Trade-off* Keamanan vs. Performa:** Lapisan interpretasi yang diperkenalkan oleh VM secara inheren menambah *overhead* performa dibandingkan eksekusi asli. Tingkat pengaburan dalam *handler* VM dan kompleksitas instruksi virtual memengaruhi *trade-off* ini.

Beberapa alat komersial seperti VMProtect [30] dan Themida [31] (yang juga mencakup fitur virtualisasi di luar pengepakan dasar) menggunakan virtualisasi kode. Penelitian akademis juga telah mengeksplorasi teknik seperti deobfuscation simbolik untuk menganalisis kode tervirtualisasi [8] dan metode untuk meningkatkan ketahanan virtualisasi, seperti *virtual code folding* [7].

2.3. Teknik dan Tantangan *Disassembly*

Memahami efektivitas pengaburan biner, khususnya virtualisasi kode, memerlukan tinjauan singkat tentang teknik *disassembly*. *Disassembler* menerjemahkan kode mesin menjadi bahasa *assembly* yang dapat dibaca manusia, membentuk landasan rekayasa balik [32].

2.3.1 *Disassembler* Statis

Disassembler statis, seperti Ghidra [11] dan IDA Pro [33], menganalisis berkas *executable* tanpa menjalankannya. Mereka menggunakan teknik seperti sapuan linier (*linear sweep*) atau penelusuran rekursif (*recursive traversal*) untuk mengidentifikasi urutan instruksi [34, 35]. Meskipun komprehensif, analisis statis kesulitan menghadapi kode yang terenkripsi, dikemas, memodifikasi diri sendiri, atau ditransformasi secara signifikan, karena *disassembler* dapat salah menginterpretasikan data sebagai kode atau gagal mengikuti alur kontrol yang sebenarnya [32, 36]. Yang krusial, ketika dihadapkan dengan *bytecode* kustom dari VM, *disassembler*

statis yang dirancang untuk ISA standar (misalnya, x86-64) tidak dapat menginterpretasikan instruksi non-natif ini dengan benar, yang menyebabkan kegagalan analisis.

2.3.2 *Disassembler* Dinamis

Disassembly dinamis, biasanya merupakan fitur dari *debugger* seperti x64dbg [12], terjadi selama eksekusi program. *Debugger* me-*disassemble* instruksi secara *on-the-fly* saat akan dieksekusi oleh CPU. Pendekatan ini dapat mengatasi beberapa keterbatasan analisis statis, seperti mengungkapkan kode yang dibongkar atau didekripsi di memori [32]. *Debugger* juga dapat mengakses informasi simbol *runtime* yang dimuat oleh sistem operasi untuk pustaka sistem, memberikan konteks untuk panggilan API. Namun, meskipun *debugger* dapat melangkah melalui instruksi natif dari *interpreter* VM yang tersemat, ia tidak akan secara langsung mengungkapkan logika asli aplikasi sebelum virtualisasi. Sebaliknya, ia menunjukkan operasi internal VM yang mengeksekusi *bytecode* kustom, yang masih mengaburkan semantik inti aplikasi dari analis. Keterbatasan inheren dari alat *disassembly* standar ini menggarisbawahi tantangan yang ditimbulkan oleh teknik pengaburan canggih seperti virtualisasi kode.

2.4. VxLang dalam Konteks

VxLang memposisikan dirinya sebagai kerangka kerja komprehensif yang menawarkan proteksi biner, pengaburan kode (termasuk *flattening*), dan virtualisasi kode [10]. Pendekatannya melibatkan transformasi kode x86-64 asli menjadi *bytecode* internal yang dieksekusi oleh VM-nya. Studi ini bertujuan untuk memberikan evaluasi empiris terhadap efektivitas komponen virtualisasi VxLang terhadap praktik rekayasa balik standar dan mengukur biaya performa terkait, sehingga menyumbangkan wawasan praktis tentang kegunaannya sebagai mekanisme proteksi perangkat lunak. Berbeda dengan analisis protektor komersial yang sudah mapan, penelitian ini berfokus pada implementasi spesifik dan dampak dari kerangka kerja VxLang.

3. Desain Penelitian dan Metodologi

Penelitian ini menggunakan pendekatan eksperimental kuantitatif untuk mengevaluasi efektivitas virtualisasi kode menggunakan VxLang dalam mempersulit upaya rekayasa balik dan menganalisis *overhead* performa yang ditimbulkannya.

3.1. Desain Eksperimen

Desain eksperimen yang digunakan adalah perbandingan antara kelompok kontrol dan kelompok eksperimen.

- **Kelompok Kontrol:** Aplikasi studi kasus dan *benchmark* yang dikompilasi secara normal tanpa penerapan virtualisasi kode VxLang.

- **Kelompok Eksperimen:** Aplikasi dan *benchmark* yang sama, namun bagian kode kritikalnya telah diproses menggunakan virtualisasi kode VxLang.

Variabel independen adalah penerapan virtualisasi kode VxLang, sedangkan variabel dependen meliputi tingkat kesulitan rekayasa balik (analisis statis dan dinamis) dan dampak performa (waktu eksekusi dan ukuran berkas).

3.2. Objek Studi

Objek studi dalam penelitian ini mencakup:

1. **Aplikasi Studi Kasus Autentikasi:** Aplikasi simulasi login pengguna dikembangkan dalam beberapa varian antarmuka (Konsol, Qt, Dear ImGui) dan dua mekanisme autentikasi (kredensial *hardcoded* dan validasi berbasis *cloud*). Aplikasi ini menjadi target utama analisis rekayasa balik.

2. **Aplikasi *Benchmark* Performa:** Aplikasi yang dirancang khusus untuk mengukur dampak performa pada tugas komputasi spesifik, yaitu algoritma QuickSort dan enkripsi AES-CBC-256. Sebuah aplikasi minimal dengan data tersemat juga digunakan untuk menilai peningkatan ukuran berkas dasar.

3. **Studi Kasus *Remote Administration Tool* (RAT):** Klien dari Lilith RAT [37], sebuah RAT *open-source*, dianalisis untuk fungsionalitas pasca-virtualisasi dan perubahan profil deteksi pada VirusTotal. Sembilan sampel *malware*/PUA tambahan juga dianalisis dampaknya pada deteksi VirusTotal.

3.3. Instrumen dan Bahan Penelitian

Penelitian ini menggunakan instrumen berikut:

- **Perangkat Keras:** PC berbasis Windows 11 (64-bit).
- **Perangkat Lunak Pengembangan:** Clang/clang-cl (C++17), CMake, Ninja, VxLang SDK, Qt 6, Dear ImGui, OpenSSL 3.x, libcurl.
- **Alat Analisis:** Ghidra (v11.x) untuk analisis statis, x64dbg (rilis terbaru) untuk analisis dinamis.
- **Pengukuran Performa:** Pustaka C++ `std::chrono` untuk waktu, `std::filesystem::file_size` untuk ukuran berkas.

3.4. Prosedur Pengumpulan Data

Pengumpulan data dilakukan melalui beberapa tahapan utama:

3.4.1 Persiapan Artefak

Aplikasi studi kasus dan *benchmark* dikompilasi dalam dua versi: asli (tanpa virtualisasi) dan *intermediate* (dengan makro VxLang dan tertaut pustaka VxLang). Versi *intermediate* kemudian diproses menggunakan *tool command-line* `vxlang.exe` untuk menghasilkan *executable* tervirtualisasi akhir (`*_vxm.exe`).

3.4.2 Analisis Keamanan

Untuk setiap aplikasi autentikasi (asli dan tervirtualisasi):

1. **Analisis Statis (Ghidra):** Memuat *executable*, mencari *string* relevan (misalnya, "Gagal", "Sah", potensi kredensial), menganalisis *disassembly/decompilation* di sekitar referensi *string* atau titik masuk, mengidentifikasi lompatan kondisional yang mengontrol keberhasilan/kegagalan autentikasi, dan mencoba *patching* statis untuk mem-*bypass* logika. Observasi sistematis dicatat.

2. **Analisis Dinamis (x64dbg):** Menjalankan *executable* di bawah *debugger*, mencari *string*/*pattern* saat *runtime*, mengatur *breakpoint* pada lokasi logika yang dicurigai, melangkah melalui eksekusi, mengamati nilai register/memori, dan mencoba manipulasi *runtime* (mem-*patch* lompatan kondisional) untuk mem-*bypass* autentikasi. Observasi sistematis dicatat.

Diagram alir prosedur pengujian keamanan autentikasi dapat dilihat pada Gambar **??** (diadaptasi dari metodologi skripsi).

3.4.3 Analisis Performa

Untuk setiap aplikasi *benchmark* (asli dan tervirtualisasi):

1. **Waktu Eksekusi:** Menjalankan *benchmark* QuickSort 100 kali per ukuran data, mencatat waktu individual. Menjalankan *benchmark* AES pada data 1GB, mencatat total waktu enkripsi/dekripsi. Menggunakan `std::chrono`. Menghitung rata-rata, standar deviasi (untuk QuickSort), dan *throughput* (untuk AES).

2. **Ukuran Berkas:** Mengukur ukuran *executable* akhir dalam *byte* menggunakan `std::filesystem::fi`

3.4.4 Analisis Lilith RAT dan Sampel Malware Lainnya

1. **Pengujian Integritas Fungsional (Lilith RAT):** Klien tervirtualisasi diuji untuk fungsionalitas inti RAT (koneksi ke server, eksekusi perintah jarak jauh, akses sistem berkas) terhadap server yang tidak dimodifikasi di jaringan lokal.

2. **Analisis VirusTotal:** Kedua versi (*executable* asli dan tervirtualisasi) dari klien Lilith RAT dan sembilan sampel *malware*/PUA lainnya diunggah ke VirusTotal untuk membandingkan tingkat deteksi dan karakterisasi ancaman.

3.5. Teknik Analisis Data

- **Analisis Proteksi Keamanan:** Metrik kuantitatif (misalnya, tingkat keberhasilan *bypass*, persentase *string* kritis yang berhasil diobfuskasi, pengurangan fungsi yang dapat diidentifikasi) dan analisis deskriptif dari observasi sistematis pada analisis statis dan dinamis.
- **Data Performa:** Perhitungan statistik deskriptif, *overhead* persentase untuk waktu eksekusi, perhitungan *throughput* (MB/s), dan persentase peningkatan ukuran berkas. Tabel dan grafik komparatif akan digunakan.
- **Analisis *Trade-off*:** Sintesis temuan keamanan dan hasil performa untuk mengevaluasi keseimbangan antara peningkatan proteksi dan biaya performa/ukuran yang ditimbulkan oleh VxLang.

4. Hasil dan Pembahasan

Bagian ini menyajikan hasil analisis keamanan dan pengukuran performa, diikuti dengan diskusi mengenai temuan tersebut.

4.1. Hasil Analisis Keamanan

Efektivitas virtualisasi VxLang dievaluasi melalui upaya analisis statis dan dinamis untuk memahami dan mem-*bypass* logika autentikasi pada aplikasi studi kasus.

4.1.1 Analisis Statis (Ghidra)

Analisis statis terhadap biner non-virtualisasi menggunakan Ghidra umumnya mudah. *String* relevan (misalnya, "Authentication Failed") dan alur kontrol untuk logika autentikasi biasanya dapat diidentifikasi. Pada biner non-virtualisasi, instruksi perbandingan standar dan lompatan kondisional yang mengontrol autentikasi mudah ditemukan, memungkinkan *patching* statis. Sebagai contoh, pada aplikasi *app_imgui* non-virtualisasi, analisis *disassembly* mengungkap perbandingan *password* dan instruksi `JNZ` yang mengarah ke blok kegagalan jika *password* salah. Instruksi ini dapat diubah menjadi `JZ` untuk membalik logika. Lebih lanjut, *string* kredensial "seno" dan "rahman" ditemukan pada bagian *defined data*, menunjukkan penyimpanan *hard-coded* yang rentan. Untuk versi *cloud*, meskipun kredensial tidak *hard-coded*, logika pemeriksaan hasil dari server pada klien tetap dapat diidentifikasi dan dimanipulasi melalui *patching* instruksi lompatan kondisional.

Sebaliknya, analisis statis terbukti secara signifikan lebih menantang untuk biner yang diproses oleh VxLang. Tabel **??** dan **??** merangkum metrik kunci dari analisis Ghidra, mengilustrasikan dampak virtualisasi.

Tabel 1: Perbandingan Metrik Analisis Statis Ghidra (Non-Virtualisasi vs. Virtualisasi) - Versi Ringkas

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

Data pada Tabel 1 secara konsisten menunjukkan penurunan drastis (umumnya >90%) jumlah instruksi, fungsi, data terdefinisi, dan simbol yang dapat dikenali pada biner tervirtualisasi. Misalnya, aplikasi *app_qt* mengalami penurunan instruksi dikenali dari 6104 menjadi 214 (**-96.49%**) dan fungsi dari 538 menjadi 25 (**-95.35%**) setelah virtualisasi. Hal ini mengindikasikan transformasi fundamental kode ke format yang tidak dapat diinterpretasi oleh *disassembler* standar. Pengurangan elemen program yang dapat diidentifikasi ini sangat menghambat analisis statis, membuatnya hampir mustahil untuk menemukan dan memahami alur kontrol logika yang relevan. Upaya *bypass* statis pada biner tervirtualisasi akibatnya tidak berhasil. Temuan ini sejalan dengan pemahaman bahwa *disassembler* statis kesulitan dengan ISA *bytecode* kustom yang diperkenalkan oleh virtualisasi [32, 34, 35]. Ukuran berkas juga meningkat secara substansial (misalnya, *app_qt.exe* dari 122 KB menjadi 1.578 KB setelah virtualisasi, peningkatan 13x), yang mengindikasikan adanya VM dan *bytecode* yang disematkan.

4.1.2 Analisis Dinamis (x64dbg)

Analisis dinamis pada biner non-virtualisasi menggunakan x64dbg relatif mudah. Pengaturan *breakpoint* berdasarkan referensi *string* atau dekat lompatan kondisional yang diidentifikasi selama analisis statis terbukti efektif. Melangkah melalui kode dengan jelas mengungkapkan logika perbandingan dan lompatan kondisional, dan *patching runtime* berhasil mem-*bypass* autentikasi.

Untuk biner tervirtualisasi VxLang, analisis dinamis menyajikan tantangan yang lebih bernuansa, sebagaimana dirangkum oleh metrik kunci pada Tabel **??** dan **??**.

Tabel 2: Perbandingan Metrik Analisis Dinamis x64dbg (Non-VM vs. VM) - Versi Ringkas

Aplikasi	Versi	Instr. Count (Observed)	Mem. Sections	Def. Symbols	Key Str. Found
app_qt (GUI)	Non-VM	8022	7	209	Ya
	VM	8011	10	15	Tidak
	Perubahan %	-0.14%	+42.86%	**-92.82%**	-
console (CLI)	Non-VM	5797	6	88	Ya
	VM	5843	9	12	Tidak
	Perubahan %	+0.79%	+50.00%	**-86.36%**	-
encryption (Benchmark)	Non-VM	8336	6	94	Ya
	VM	8207	9	13	Tidak
	Perubahan %	-1.55%	+50.00%	**-86.17%**	-

Observasi kunci dari analisis dinamis biner tervirtualisasi meliputi:

- **Visibilitas Instruksi VM Natif:** Setelah aplikasi tervirtualisasi dimuat sepenuhnya, x64dbg menampilkan instruksi x86-64 natif yang valid. Instruksi ini milik *interpreter* VM VxLang, bukan logika natif asli aplikasi. Jumlah instruksi yang teramati (Tabel **??** dan **??**) tidak berubah secara drastis, yang mencerminkan aktivitas VM.
- **Peningkatan Segmen Memori:** Jumlah segmen memori secara konsisten meningkat (sekitar 40-50%), kemungkinan karena *runtime* VxLang dan *bytecode*.
- **Obfuskasi Data Kritis dan Logika Aplikasi yang Persisten:**
  - **Obfuskasi *String* Kunci:** *String* kritis yang ditargetkan untuk virtualisasi (misalnya, "Authentication Failed") **tidak ditemukan** menggunakan pencarian *runtime* standar di x64dbg.
  - **Pengurangan Drastis Simbol Terdefinisi:** Simbol terdefinisi yang dapat diamati berkurang secara signifikan (>85%), menghambat pemahaman kontekstual.
  - **Abstraksi Logika Aplikasi:** Logika inti aplikasi ditransformasikan menjadi *bytecode* internal yang dieksekusi oleh VM VxLang. *Debugger* menunjukkan eksekusi VM, bukan eksekusi natif langsung dari logika asli, sehingga sangat sulit untuk melacak atau memahami perilaku yang dimaksudkan aplikasi.
- **Ketidakefektifan *Patching Runtime* Sederhana:** Akibatnya, upaya untuk mem-*bypass* autentikasi dengan mem-*patch* lompatan kondisional sederhana saat *runtime* menjadi tidak efektif. Titik pengambilan keputusan kritis tertanam dalam alur eksekusi buram VM.

Temuan analisis dinamis ini sejalan dengan prinsip-prinsip pengaburan berbasis VM [32, 5]. Meskipun *debugger* dapat melangkah melalui kode natif *interpreter* VM, logika aplikasi aktual diabstraksi, membuat analisis dan manipulasi langsung sangat sulit tanpa pengetahuan mendalam tentang arsitektur VM [8].

4.1.3 Analisis Perangkat Lunak Potensial Berbahaya dan Deteksi VirusTotal

Untuk mengevaluasi dampak VxLang pada perangkat lunak yang lebih kompleks dan deteksi otomatis, sepuluh sampel perangkat lunak, termasuk klien Lilith RAT [37] dan sembilan sampel *malware*/PUA lainnya (*Al-Khaser, donut, DripLoader, FilelessPELoader, JuicyPotato, ParadoxiaClient, PELoader, RunPE-In-Memory, SigLoader*), dianalisis. Untuk Lilith RAT, setelah penempatan makro VxLang yang hati-hati dan iteratif pada fungsi-fungsi kritis, pengujian fungsional mengkonfirmasi bahwa klien tervirtualisasi tetap beroperasi penuh.

Kesepuluh sampel, dalam bentuk asli dan tervirtualisasi VxLang, diunggah ke VirusTotal (72 *engine* AV). Hasilnya (dirangkum dalam Tabel 3) menunjukkan dampak yang bervariasi pada tingkat deteksi. Untuk lima sampel, termasuk Lilith RAT (22 menjadi 18 deteksi) dan *donut* (30 menjadi 19 deteksi), virtualisasi menyebabkan penurunan deteksi. Namun, untuk lima sampel lainnya, seperti *JuicyPotato* (9 menjadi 20 deteksi), virtualisasi justru meningkatkan deteksi. Hal ini menunjukkan bahwa lapisan virtualisasi itu sendiri dapat memicu kecurigaan dari beberapa *engine* AV. Secara keseluruhan, perubahan rata-rata deteksi minimal (+0.4 deteksi).

Tabel 3: Perbandingan Jumlah Deteksi VirusTotal untuk Berbagai Sampel (Non-VM vs. VM dari 72 Engine)

Malware/Aplikasi	**Deteksi Non-VM**	**Deteksi VM**	**Perubahan Jumlah Deteksi**
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**
Rata-rata Perubahan Deteksi			**+0.4**

4.2. Hasil *Overhead* Performa dan Ukuran

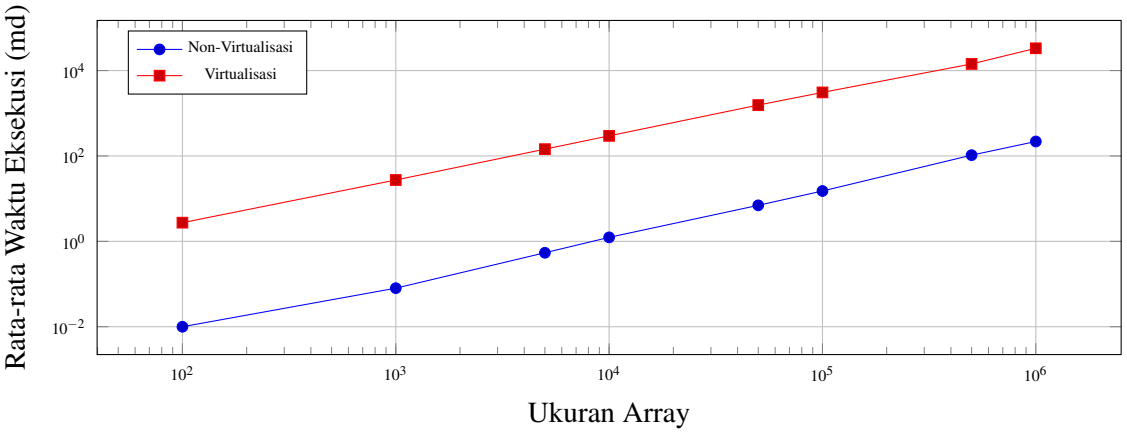
4.2.1 *Overhead* Waktu Eksekusi

Dampak performa diukur menggunakan *benchmark* QuickSort dan AES.

- **QuickSort:** Seperti yang ditunjukkan pada Tabel 4 dan Gambar 3, virtualisasi memperkenalkan *overhead* waktu eksekusi yang substansial. *Overhead* meningkat seiring ukuran data, berkisar dari sekitar 27.300% untuk 100 elemen (0.01 md menjadi 2.74 md) hingga sekitar 15.150% untuk 1.000.000 elemen (218.32 md menjadi 33.292.91 md).

Tabel 4: Hasil Waktu Eksekusi Quick Sort (ms)

Ukuran Array	Non-Virtualisasi		Virtualisasi	
	Rata-rata Waktu	Std Dev	Rata-rata Waktu	Std Dev
100	0.01	0.00	2.74	0.38
1.000	0.08	0.00	27.35	1.25
1.000.000	218.32	8.10	33292.91	4342.93



Gambar 3: Perbandingan Waktu Eksekusi Quick Sort (Skala Log-Log).

- **Enkripsi AES:** Tabel 5 menunjukkan bahwa total waktu untuk mengenkripsi 976MB data meningkat sekitar 396.7% (1878.52 md menjadi 9330.73 md), dan waktu dekripsi meningkat sekitar 562.9% (1304.75 md menjadi 8649.74 md). Akibatnya, *throughput* gabungan turun drastis dari 634.16 MB/s menjadi 108.78 MB/s (penurunan 82.8%).

Tabel 5: Hasil Performa AES-256-CBC (Data 976MB)

Metrik	Non-Virtualisasi	Virtualisasi
Total Waktu Enkripsi (md)	1.878.52	9.330.73
Total Waktu Dekripsi (md)	1.304.75	8.649.74
Throughput Gabungan (MB/s)	634.16	108.78

4.2.2 *Overhead* Ukuran Berkas

Tabel 6 menunjukkan peningkatan konsisten ukuran *executable* setelah virtualisasi. Untuk program konsol/benchmark yang lebih kecil, ukurannya meningkat lebih dari 15-18 kali lipat.

Untuk aplikasi GUI yang lebih besar, peningkatan relatif lebih kecil namun tetap signifikan.

Tabel 6: Perbandingan Ukuran Berkas Executable (KB)

Program	**Non-Virtualisasi (KB)**	**Virtualisasi (KB)**
quick_sort	98	1.537
Lilith_Client	84	1.554

4.3. Diskusi

Hasil eksperimen dengan jelas menunjukkan *trade-off* inti dalam penggunaan virtualisasi kode VxLang.

Peningkatan Keamanan dan Penghindaran Deteksi: VxLang memberikan penghalang substansial terhadap teknik rekayasa balik umum. Transformasi menjadi *bytecode* yang diinterpretasi menetralisir alat analisis statis standar seperti Ghidra [34, 35] dan secara signifikan mempersulit analisis dinamis dengan alat seperti x64dbg, karena logika yang mendasarinya dieksekusi oleh VM buram [32]. Hal ini sejalan dengan pemahaman mapan bahwa pengaburan berbasis VM secara fundamental mengubah struktur kode di luar kemampuan interpretasi *disassembler* standar [5, 8]. Lebih lanjut, analisis VirusTotal terhadap sepuluh sampel *malware*/PUA mengungkapkan dampak yang bernuansa: sementara VxLang dapat mengurangi deteksi berbasis *signature* untuk beberapa sampel, ia menyebabkan peningkatan deteksi untuk sampel lain, kemungkinan karena lapisan virtualisasi itu sendiri ditandai. Ini menyoroti interaksi kompleks antara pengaburan dan heuristik deteksi AV yang berkembang [5, 8, 29].

Biaya Performa: Manfaat keamanan dan penghindaran deteksi datang dengan harga performa yang mahal. *Overhead* interpretasi secara signifikan memperlambat kode tervirtualisasi, terutama untuk tugas-tugas komputasi intensif (*overhead* QuickSort 15.000% untuk 1 juta elemen; pengurangan *throughput* AES 83%), yang berpotensi membuat aplikasi yang tidak pandang bulu menjadi tidak praktis karena degradasi kecepatan yang parah.

Peningkatan Ukuran: Peningkatan ukuran berkas yang cukup besar (misalnya, 15-18x untuk aplikasi kecil), terutama karena *runtime* VM yang disematkan, merupakan faktor lain, khususnya relevan untuk aplikasi kecil atau kendala distribusi.

Implikasi Praktis: VxLang tampak ampuh untuk melindungi kode yang sangat sensitif di mana keamanan dan potensi penghindaran deteksi adalah yang terpenting, dan dampak performa pada segmen spesifik tersebut dapat diterima (misalnya, anti-*tamper*, lisensi, IP inti). Kasus Lilith menunjukkan bahwa VxLang dapat melindungi logika kompleks tanpa merusaknya, **asalkan penempatan makro dilakukan dengan hati-hati dan iteratif untuk menghindari gangguan fungsionalitas, terutama pada bagian kode dengan alur kontrol kompleks atau operasi I/O**. Namun, biaya performa yang parah mengharuskan aplikasi strategis dan selektif, hanya menargetkan bagian kritis. Hasil VirusTotal juga menyiratkan bahwa meskipun profil deteksi diubah, penghindaran tidak dijamin. Pilihan antara autentikasi *hardcoded* dan berbasis

cloud menunjukkan bahwa melindungi logika sisi klien yang menangani hasil validasi tetap krusial.

5. Kesimpulan

Penelitian ini menginvestigasi efektivitas virtualisasi kode menggunakan kerangka kerja VxLang untuk mempersulit rekayasa balik perangkat lunak. Implementasi melibatkan penandaan kode sumber, kompilasi menjadi *executable intermediate*, dan pemrosesan akhir dengan *tool* VxLang untuk menghasilkan biner tervirtualisasi.

Hasil analisis eksperimental menunjukkan bahwa virtualisasi kode VxLang secara signifikan meningkatkan kesulitan rekayasa balik. Analisis statis menggunakan Ghidra terhadap kode tervirtualisasi gagal mengidentifikasi instruksi, fungsi, atau struktur data yang bermakna. Demikian pula, analisis dinamis dengan x64dbg terhambat oleh alur kontrol yang diobfuskasi dan model eksekusi mesin virtual (VM) yang mengaburkan perilaku *runtime* dan upaya *debugging*. Upaya untuk mem-*bypass* logika autentikasi, yang mudah dilakukan pada versi non-virtualisasi, berhasil digagalkan pada biner yang dilindungi VxLang. Namun, implementasi yang efektif memerlukan penempatan makro virtualisasi yang cermat dan iteratif, karena penempatan yang kurang tepat, terutama pada kode dengan I/O atau alur kontrol kompleks, dapat mengganggu fungsionalitas aplikasi, sebagaimana teramati pada studi kasus Lilith RAT.

Analisis terhadap sepuluh sampel *malware*/PUA di VirusTotal menunjukkan dampak VxLang yang bervariasi terhadap tingkat deteksi: sekitar separuh sampel mengalami penurunan deteksi, seringkali dengan pergeseran ke *flag* generik/heuristik, sementara sisanya menunjukkan peningkatan deteksi, mengindikasikan bahwa lapisan virtualisasi itu sendiri dapat memicu peringatan.

Peningkatan keamanan ini dibarengi dengan *overhead* performa yang substansial, teramati pada *benchmark* algoritma QuickSort dan enkripsi AES, serta peningkatan signifikan ukuran berkas *executable*. Temuan ini menggarisbawahi adanya *trade-off* yang jelas antara proteksi yang kuat terhadap rekayasa balik dengan degradasi performa dan penambahan ukuran berkas. Oleh karena itu, aplikasi praktis VxLang sebaiknya dilakukan secara selektif, menargetkan hanya bagian kode yang paling kritis dan sensitif.

Penelitian selanjutnya dapat difokuskan pada eksplorasi teknik rekayasa balik yang lebih canggih terhadap proteksi berbasis VM, investigasi opsi konfigurasi VxLang untuk keseimbangan keamanan-performa, dan studi komparatif dengan solusi virtualisasi lainnya. Analisis lebih mendalam mengenai interaksi VxLang dengan berbagai jenis *malware* dan teknik deteksi antivirus juga akan memberikan wawasan yang berharga.

REFERENSI

[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.