UNIVERSITAS INDONESIA

IMPLEMENTASI DAN ANALISIS EFEKTIVITAS VXLANG *CODE VIRTUALIZATION* DALAM MEMPERSULIT *REVERSE ENGINEERING*

SKRIPSI

SENO PAMUNGKAS RAHMAN
2106731586

FAKULTAS TEKNIK
PROGRAM STUDI TEKNIK KOMPUTER
DEPOK
2025

UNIVERSITAS INDONESIA

IMPLEMENTASI DAN ANALISIS EFEKTIVITAS VXLANG *CODE VIRTUALIZATION* DALAM MEMPERSULIT *REVERSE ENGINEERING*

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar Sarjana Teknik

SENO PAMUNGKAS RAHMAN
2106731586

FAKULTAS TEKNIK
PROGRAM STUDI TEKNIK KOMPUTER
DEPOK
MEI 2025

HALAMAN PERSETUJUAN

Judul	:	Implementasi dan Analisis Efektivitas Vxlang *Code Virtualization* dalam mempersulit *Reverse Engineering*
Penulis	:	Seno Pamungkas Rahman
NPM	:	2106731586

Laporan Skripsi ini telah diperiksa dan disetujui.

Mei 2025

Dr. Ruki Harwahyu, S.T. M.T. MSc.

Pembimbing Skripsi

HALAMAN PERNYATAAN ORISINALITAS

Skripsi ini adalah hasil karya saya sendiri,
dan semua sumber baik yang dikutip maupun dirujuk
telah saya nyatakan dengan benar.

Nama	:	**Seno Pamungkas Rahman**
NPM	:	**2106731586**
Tanda Tangan	:	
Tanggal	:	**Mei 2025**

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh	:	
Nama	:	Seno Pamungkas Rahman
NPM	:	2106731586
Program Studi	:	Teknik Komputer
Judul Skripsi	:	Implementasi dan Analisis Efektivitas Vxlang *Code Virtualization* dalam mempersulit *Reverse Engineering*

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Teknik pada Program Studi Teknik Komputer, Fakultas Teknik, Universitas Indonesia.

DEWAN PENGUJI

Pembimbing	:	Dr. Ruki Harwahyu, S.T. M.T. MSc.	( )
Penguji	:	Yan Maraden, S.T., M.T., M.Sc.	( )
Penguji	:	Muhammad Firdaus Syawaludin Lubis, S.T., MT., Ph.D.	( )

Ditetapkan di	:	Depok
Tanggal	:	Mei 2025

KATA PENGANTAR

Penulis mengucapkan terima kasih kepada :

1. Dr. Ruki Harwahyu, ST, MT, MSc. dosen pembimbing atas segala bimbingan, ilmu, dan arahan baik dalam penulisan skripsi maupun selama masa studi di Teknik Komputer.
2. Orang tua dan keluarga yang telah memberikan bantuan dukungan material dan moral.
3. Teman-teman di program studi Teknik Koputer atas segala dukungan dan kerja samanya.

sehingga penulisan skripsi ini dapat diselesaikan dengan baik dan benar. Akhir kata, penulis berharap Tuhan Yang Maha Esa berkenan membalas segala kebaikan semua pihak yang telah membantu. Penulis berharap kritik dan saran untuk melengkapi kekurangan pada skripsi ini.

Depok, 22 Mei 2025

Seno Pamungkas Rahman

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama	:	Seno Pamungkas Rahman
NPM	:	2106731586
Program Studi	:	Teknik Komputer
Fakultas	:	Teknik
Jenis Karya	:	Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (*Non-exclusive Royalty Free Right*)** atas karya ilmiah saya yang berjudul:

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

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyatan ini saya buat dengan sebenarnya.

Dibuat di : Depok
Pada tanggal : Mei 2025
Yang menyatakan

(Seno Pamungkas Rahman)

ABSTRAK

Nama	:	Seno Pamungkas Rahman
Program Studi	:	Teknik Komputer
Judul	:	Implementasi dan Analisis Efektivitas Vxlang *Code Virtualization* dalam mempersulit *Reverse Engineering*
Pembimbing	:	Dr. Ruki Harwahyu, S.T. M.T. MSc.

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.

Kata kunci:
Pengaburan Kode, Virtualisasi Kode, Perlindungan Perangkat Lunak, Rekayasa Balik

ABSTRACT

Name	:	Seno Pamungkas Rahman
Study Program	:	Teknik Komputer
Title	:	Implementation and Analysis of the Effectiveness of Vxlang Code Virtualization in complicating Reverse Engineering
Counsellor	:	Dr. Ruki Harwahyu, S.T. M.T. MSc.

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.

Key words:
Code Obfuscation, Code Virtualization, Software Protection, Reverse Engineering

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR KODE

BAB 1
PENDAHULUAN

1.1 Latar Belakang

Perkembangan pesat teknologi perangkat lunak telah mendorong terciptanya aplikasi yang semakin kompleks dan canggih, menawarkan berbagai inovasi dan manfaat di berbagai sektor kehidupan. Namun, kemajuan ini juga diiringi oleh peningkatan ancaman keamanan yang semakin beragam dan canggih. Salah satu ancaman yang signifikan adalah rekayasa balik (reverse engineering). Reverse engineering adalah proses menganalisis suatu sistem, dalam hal ini perangkat lunak, untuk mengidentifikasi komponen-komponennya, interaksi antar komponen, dan memahami cara kerja sistem tersebut tanpa akses ke dokumentasi asli atau kode sumber. Dalam konteks perangkat lunak, reverse engineering memungkinkan pihak yang tidak berwenang untuk membongkar kode program, memahami algoritma yang digunakan, menemukan kerentanan, mencuri rahasia dagang, melanggar hak cipta, dan bahkan menyisipkan kode berbahaya.

Teknik-teknik keamanan konvensional seperti enkripsi data dan proteksi password seringkali tidak cukup ampuh untuk mencegah reverse engineering. Enkripsi hanya melindungi data saat transit atau saat disimpan, tetapi tidak melindungi kode program itu sendiri. Penyerang yang berhasil mendapatkan akses ke program yang berjalan dapat mencoba untuk membongkar dan menganalisis kode meskipun data dienkripsi. Demikian pula, proteksi password hanya membatasi akses awal ke program, tetapi tidak mencegah reverse engineering setelah program dijalankan. Penyerang dapat mencoba untuk melewati mekanisme otentikasi atau membongkar program untuk menemukan password atau kunci enkripsi.

Oleh karena itu, dibutuhkan teknik perlindungan yang lebih robust dan proaktif untuk mengamankan perangkat lunak dari reverse engineering. Salah satu pendekatan yang menjanjikan adalah obfuscation. Obfuscation bertujuan untuk mengubah kode program menjadi bentuk yang lebih sulit dipahami oleh manusia, tanpa mengubah fungsionalitasnya. Obfuscation dapat dilakukan pada berbagai tingkatan, mulai dari mengubah nama variabel dan fungsi menjadi nama yang tidak bermakna, hingga mengubah alur kontrol program menjadi lebih kompleks dan sulit dilacak.

Di antara berbagai teknik obfuscation, code virtualization dianggap sebagai salah satu yang paling efektif. Code virtualization bekerja dengan menerjemahkan kode mesin asli (native code) menjadi instruksi virtual (bytecode) yang dieksekusi oleh mesin virtual (VM) khusus yang tertanam dalam aplikasi [1]. VM ini memiliki Instruction Set Architecture (ISA) yang unik dan berbeda dari ISA prosesor standar. Dengan demikian, tools reverse engineering konvensional seperti disassembler dan debugger tidak dapat langsung digunakan untuk menganalisis kode yang divirtualisasi. Penyerang harus terlebih dahulu memahami ISA dan implementasi VM untuk dapat menganalisis bytecode, yang secara signifikan meningkatkan kompleksitas dan waktu yang dibutuhkan untuk melakukan reverse engineering.

VxLang merupakan salah satu platform code virtualization yang menarik untuk dikaji. VxLang menyediakan framework untuk melakukan code virtualization pada berbagai platform dan arsitektur prosesor [2]. Penelitian ini akan mengkaji implementasi dan efektivitas VxLang dalam melindungi perangkat lunak dari rekayasa balik. Dengan menganalisis tingkat kesulitan reverse engineering pada kode yang dilindungi oleh VxLang, ditinjau dari segi analisis statis dan dinamis, penelitian ini diharapkan dapat memberikan kontribusi dalam pengembangan teknik perlindungan perangkat lunak yang lebih aman, handal, dan efektif dalam menghadapi ancaman reverse engineering. Hasil penelitian ini juga diharapkan dapat memberikan informasi berharga bagi para pengembang perangkat lunak dalam memilih dan mengimplementasikan teknik perlindungan yang tepat untuk aplikasi mereka.

1.2 Rumusan Masalah

Berdasarkan latar belakang di atas, rumusan masalah dalam penelitian ini dirumuskan sebagai berikut:

1. Bagaimana implementasi code virtualization menggunakan VxLang pada perangkat lunak, termasuk tahapan-tahapan yang terlibat dan konfigurasi yang diperlukan?
2. Seberapa efektifkah code virtualization menggunakan VxLang dalam meningkatkan keamanan perangkat lunak terhadap rekayasa balik, diukur dari segi kompleksitas analisis kode menggunakan teknik analisis statis dan dinamis?
3. Bagaimana pengaruh code virtualization menggunakan VxLang terhadap performa perangkat lunak, ditinjau dari waktu eksekusi, ukuran file program dan apa trade-off antara keamanan dan performa?

1.3 Tujuan Penelitian

Tujuan dari penelitian ini adalah:

1. Mengimplementasikan code virtualization menggunakan VxLang pada sebuah aplikasi studi kasus, mencakup seluruh tahapan implementasi dan konfigurasi.
2. Menganalisis efektivitas code virtualization menggunakan VxLang dalam meningkatkan keamanan perangkat lunak terhadap reverse engineering melalui analisis statis dan dinamis, membandingkan tingkat kesulitan analisis kode sebelum dan sesudah di-obfuscate.
3. Mengevaluasi pengaruh code virtualization menggunakan VxLang terhadap performa perangkat lunak dengan mengukur waktu eksekusi, ukuran file program, dan menganalisis trade-off antara keamanan dan performa.

1.4 Batasan Masalah

Untuk menjaga fokus dan kedalaman penelitian, batasan masalah dalam penelitian ini adalah:

1. Platform code virtualization yang digunakan hanya VxLang
2. Analisis reverse engineering dibatasi pada analisis statis menggunakan disassembler dan decompiler (Ghidra), serta analisis dinamis menggunakan debugger (x64dbg).
3. Pengujian performa perangkat lunak dibatasi pada pengukuran waktu eksekusi, dan ukuran file program.

1.5 Metodologi Penelitian

Metode penelitian yang digunakan dalam penelitian ini adalah metode eksperimental kuantitatif. Langkah-langkah penelitian meliputi:

1. Studi Literatur Mencakup peninjauan sumber-sumber seperti jurnal, artikel, buku, dan dokumentasi terkait perangkat lunak, rekayasa balik, obfuscation, code virtualization, dan VxLang. Studi literatur ini bertujuan untuk membangun landasan teori yang kuat dan memahami penelitian terdahulu yang relevan.
2. Konsultasi Melibatkan diskusi berkala dengan dosen pembimbing untuk mendapatkan bimbingan, arahan, dan masukan terkait perkembangan penelitian.

3. Pengujian Perangkat Lunak Tahap ini meliputi implementasi code virtualization menggunakan VxLang pada aplikasi serta pengujian perangkat lunak untuk mengevaluasi efektivitas dan dampaknya. Pengujian ini dilakukan dengan nmenguji tingkat kesulitan reverse engineering pada aplikasi sebelum dan sesudah di-obfuscate, baik melalui analisis statis (disassembler, decompiler) maupun analisis dinamis (debugger).

4. Analisis Menganalisis data yang diperoleh dari tahap pengujian perangkat lunak untuk mengevaluasi efektivitas VxLang dalam mempersulit rekayasa balik dan mengukur dampaknya terhadap performa aplikasi. Analisis ini meliputi perbandingan hasil pengujian antara aplikasi sebelum dan sesudah di-obfuscate.

5. Kesimpulan Merumuskan kesimpulan akhir dari penelitian berdasarkan hasil analisis data. Kesimpulan harus menjawab rumusan masalah dan tujuan penelitian.

1.6 Sistematika Penulisan

Seminar ini disusun dengan sistematika sebagai berikut:

BAB 1 – PENDAHULUAN

Bab ini berisi latar belakang, rumusan masalah, tujuan penelitian, batasan masalah, metodologi penelitian, dan sistematika penulisan.

BAB 2 – TINJAUAN PUSTAKA

Bab ini membahas teori-teori dasar tentang perangkat lunak, rekayasa balik, obfuscation, code virtualization, dan VxLang.

BAB 3 – METODE PENELITIAN

Bab ini menjelaskan langkah-langkah penelitian, desain eksperimen, alat dan bahan, serta teknik analisis data.

BAB 4 – HASIL DAN PEMBAHASAN

Bab ini menyajikan hasil pengujian dan analisis, serta pembahasan terkait temuan penelitian.

BAB 5 – KESIMPULAN DAN SARAN

Bab ini merumuskan kesimpulan dan saran dari penelitian.

BAB 2
TINJAUAN PUSTAKA

2.1 Perangkat Lunak (*Software*)

Perangkat lunak adalah serangkaian instruksi, data, atau program yang digunakan untuk mengoperasikan komputer dan menjalankan tugas-tugas tertentu [3]. Perangkat lunak memberikan instruksi kepada perangkat keras (hardware) tentang apa yang harus dilakukan, bertindak sebagai perantara antara pengguna dan perangkat keras. Tanpa perangkat lunak, sebagian besar hardware komputer tidak akan berfungsi. Sebagai contoh, prosesor membutuhkan instruksi dari perangkat lunak untuk melakukan perhitungan, dan monitor membutuhkan driver perangkat lunak untuk menampilkan gambar. Perangkat lunak tidak memiliki wujud fisik dan bersifat intangible, berbeda dengan hardware yang dapat disentuh. Perangkat lunak didistribusikan dalam berbagai bentuk, seperti program yang diinstal pada komputer, aplikasi mobile, aplikasi web, dan embedded systems.

2.1.1 Perangkat Lunak Sistem

Perangkat lunak sistem merupakan fondasi yang memungkinkan perangkat lunak aplikasi dan pengguna berinteraksi dengan perangkat keras [3]. Fungsinya antara lain mengelola sumber daya sistem seperti memori, prosesor, dan perangkat input/output. Perangkat lunak sistem juga menyediakan layanan dasar seperti sistem file, manajemen proses, dan antarmuka pengguna. Contoh perangkat lunak sistem meliputi:

1. **Sistem Operasi (*Operating System*):** Bertindak sebagai platform untuk menjalankan perangkat lunak aplikasi. Sistem operasi mengelola sumber daya hardware, menyediakan antarmuka pengguna, dan menjalankan layanan sistem. Contoh: Microsoft Windows, macOS, Linux, Android, iOS.
2. **Driver Perangkat Keras (*Device Drivers*):** Program yang memungkinkan sistem operasi untuk berkomunikasi dengan perangkat keras tertentu, seperti printer, kartu grafis, kartu suara, webcam, dan mouse. Setiap perangkat keras membutuhkan driver khusus agar dapat berfungsi dengan baik.

2.1.2 Perangkat Lunak Aplikasi

Perangkat lunak aplikasi dirancang untuk memenuhi kebutuhan spesifik pengguna [3]. Kategori perangkat lunak aplikasi sangat luas dan beragam, berfokus pada penyelesaian

Gambar 2.1: Sistem Operasi Windows & Linux

tugas-tugas tertentu untuk pengguna. Perangkat lunak aplikasi berjalan di atas sistem operasi dan memanfaatkan layanan yang disediakan oleh sistem operasi.

- **Pengolah Kata (*Word Processors*):** Digunakan untuk membuat dan mengedit dokumen teks, memformat teks, menambahkan gambar dan tabel, dan melakukan tugas-tugas pengolah kata lainnya. Contoh: Microsoft Word, Google Docs
- **Perangkat Lunak Desain Grafis:** Digunakan untuk membuat dan mengedit gambar, ilustrasi, dan desain visual lainnya. Contoh: Adobe Photoshop, GIMP, Inkscape.

Gambar 2.2: Perangkat lunak aplikasi

2.1.3 Proses Kompilasi dan Eksekusi Perangkat Lunak

Proses menjalankan perangkat lunak melibatkan dua tahap utama: kompilasi dan eksekusi. Kedua tahap ini penting untuk mengubah kode sumber yang dapat dibaca manusia menjadi instruksi yang dapat dieksekusi oleh mesin. Berikut penjelasan lebih detail, dibagi menjadi dua sub-bagian:

2.1.3.1 Proses Kompilasi

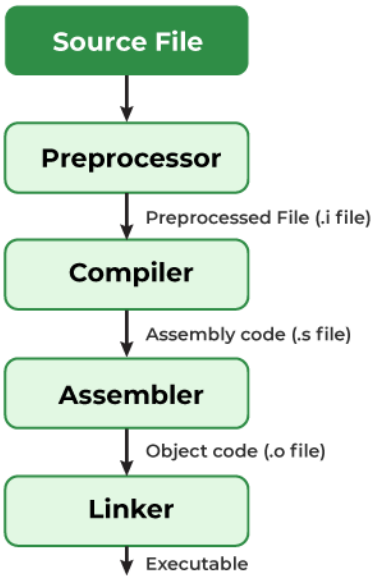
Proses kompilasi mengubah kode sumber (source code) yang ditulis dalam bahasa pemrograman tingkat tinggi menjadi kode mesin (machine code) atau kode objek (object code) [4]. Proses ini melibatkan beberapa tahapan, yang masing-masing dilakukan oleh program utilitas yang berbeda:

1. **Preprocessing:** Tahap pertama dalam proses kompilasi adalah preprocessing. Preprocessor menangani direktif-direktif preprocessor yang dimulai dengan simbol #, seperti #include dan #define dalam kode sumber.

2. **Compilation:** Pada tahap ini, compiler menerjemahkan kode sumber yang telah diproses menjadi assembly code. Assembly code adalah representasi mnemonic dari kode mesin, yang lebih mudah dibaca oleh manusia. Kompiler melakukan analisis sintaks dan semantik untuk memastikan kode sumber valid dan sesuai dengan aturan bahasa pemrograman. Kompiler juga melakukan optimasi kode untuk meningkatkan kinerja program.

3. **Assembly:** Assembler menerjemahkan assembly code menjadi kode objek (object code). Kode objek adalah representasi biner dari instruksi mesin, tetapi belum siap untuk dieksekusi. Kode objek berisi instruksi mesin dan data, tetapi belum terhubung dengan library eksternal.

4. **Linking:** Linker menggabungkan kode objek dari berbagai file sumber dan library menjadi satu file executable. Linker menyelesaikan referensi eksternal, mengalokasikan alamat memori untuk variabel dan fungsi, dan menghubungkan kode objek dengan library yang dibutuhkan. Output dari tahap linking adalah file executable yang siap dieksekusi.



Gambar 2.3: Alur kompilasi program [4]

2.1.3.2 Proses Eksekusi

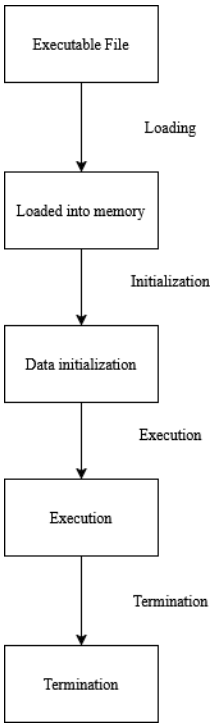
Setelah program dikompilasi menjadi file executable, proses eksekusi dimulai [5]. Proses eksekusi melibatkan beberapa tahapan yang dilakukan oleh sistem operasi:

1. **Loading:** Loader yaitu sebuah komponen dari sistem operasi, memuat file executable ke dalam memori utama (RAM). Loader mengalokasikan ruang memori

yang dibutuhkan oleh program, memuat instruksi dan data ke dalam memori, dan menginisialisasi program untuk eksekusi.

2. **Eksekusi:** Setelah program dimuat ke dalam memori, prosesor mulai mengeksekusi instruksi-instruksi yang terdapat dalam program. Prosesor mengambil instruksi satu per satu dari memori, mendekode instruksi, dan mengeksekusinya. Siklus ini berulang hingga program selesai dijalankan atau dihentikan.

3. **Terminasi:** Program berakhir ketika mencapai instruksi terminasi atau ketika terjadi kesalahan yang menyebabkan program berhenti secara paksa. Sistem operasi kemudian membebaskan sumber daya yang digunakan oleh program, seperti memori dan file.



Gambar 2.4: Alur eksekusi program [5]

2.2 *Software Control Flow*

Alur kendali perangkat lunak (software control flow) merupakan aspek fundamental dalam eksekusi program, merepresentasikan urutan instruksi yang dieksekusi oleh prosesor untuk mencapai tujuan fungsionalitas program [6]. Memahami software control flow adalah krusial dalam analisis kode, terutama dalam reverse engineering karena memungkinkan pemetaan jalur eksekusi, identifikasi bottleneck, dan potensi kerentanan. Dalam konteks ini, control flow bukan hanya sekadar urutan instruksi, tetapi juga representasi

logis dari bagaimana program merespons input, kondisi, dan interaksi dengan sistem operasi. Pemahaman yang komprehensif terhadap aspek ini memberikan dasar yang kuat dalam menganalisis perilaku program, baik secara statis melalui analisis kode sumber dan intermediate representation, maupun secara dinamis melalui observasi eksekusi runtime.

2.2.1 *Control Flow Instructions*

Instruksi alur kendali (control flow instructions) adalah mekanisme fundamental yang menentukan urutan eksekusi instruksi dalam suatu program [6]. Instruksi ini memungkinkan program untuk membuat keputusan, mengulang blok kode, dan melompat ke bagian kode yang berbeda. Cara instruksi-instruksi ini direpresentasikan dan diimplementasikan sangat bergantung pada tingkat bahasa pemrograman yang digunakan. Secara umum, kita membedakan antara bahasa tingkat tinggi dan bahasa tingkat rendah.

2.2.1.1 *High Level Languages*

Bahasa pemrograman tingkat tinggi (high-level languages), seperti Python, Java, C++, dan JavaScript, menyediakan abstraksi yang jauh dari detail hardware. Bahasa-bahasa ini fokus pada kemudahan penulisan dan pemahaman kode oleh programmer, dengan menggunakan sintaks yang lebih dekat dengan bahasa manusia. Instruksi control flow dalam bahasa tingkat tinggi diimplementasikan melalui konstruksi yang intuitif, seperti *if, else, for*, dan *while*.

Conditional Instructions:

- ***if statement:*** Memungkinkan program membuat keputusan berdasarkan kondisi, seperti pada contoh di Kode 2.1.

```
if (x > 10) {
 std::cout << "x lebih besar dari 10" << std::endl;
}
```

Kode 2.1: Contoh Penggunaan Pernyataan `if` dalam C++

- ***if-else statement:*** Menyediakan jalur alternatif jika kondisi *if* tidak terpenuhi, sebagaimana diilustrasikan pada Kode 2.2.

```
if (score >= 70) {
 std::cout << "Lulus" << std::endl;
} else {
 std::cout << "Tidak Lulus" << std::endl;
}
```

Kode 2.2: Contoh Penggunaan Pernyataan `if-else` dalam C++

- ***switch statement:*** Memungkinkan percabangan ke banyak kasus (lihat Kode 2.3).

```
int day = 2;
switch (day) {
 case 1:
 std::cout << "Senin" << std::endl;
 break;
 case 2:
 std::cout << "Selasa" << std::endl;
 break;
 default:
 std::cout << "Hari lain" << std::endl;
}
```

Kode 2.3: Contoh Penggunaan Pernyataan `switch` dalam C++

Loop Instructions:

- ***for loop:*** Mengeksekusi blok kode sejumlah kali tertentu, seperti tampak pada Kode 2.4.

```
for (int i = 0; i < 5; i++) {
 std::cout << i << std::endl;
}
```

Kode 2.4: Contoh Penggunaan Perulangan `for` dalam C++

- ***while loop:*** Mengeksekusi blok kode selama kondisi tertentu terpenuhi (Kode 2.5).

```
int count = 0;
while (count < 5) {
 std::cout << count << std::endl;
 count++;
}
```

Kode 2.5: Contoh Penggunaan Perulangan `while` dalam C++

Jump Instructions (Indirect - Function Calls):

- ***Function call:*** Memindahkan kontrol ke fungsi lain dan kembali setelah selesai, seperti pada contoh pemanggilan `myFunction` di Kode 2.6.

```
void myFunction() {
 std::cout << "Hello" << std::endl;
}
int main() {
 myFunction();
 return 0;
}
```

Kode 2.6: Contoh Pemanggilan Fungsi dalam C++

2.2.1.2 *Low Level Languages*

Bahasa pemrograman tingkat rendah (low-level languages), seperti Assembly language, bekerja lebih dekat dengan hardware. Instruksi-instruksinya langsung dikodekan ke dalam instruksi mesin yang dapat dieksekusi oleh prosesor. Bahasa tingkat rendah memberikan kontrol yang lebih besar atas hardware, tetapi seringkali lebih kompleks dan sulit untuk dipahami oleh manusia. Instruksi control flow pada tingkat ini melibatkan kode operasi (opcode) yang merepresentasikan operasi jump dan perbandingan secara langsung [7]. Dalam bagian ini, contoh akan difokuskan pada bahasa x86 Assembly.

Conditional Instructions:

- ***cmp (compare):*** Membandingkan dua nilai dan mengatur flags (bendera) kondisi (Kode 2.7).

```
cmp eax, 10
```

Kode 2.7: Contoh Instruksi Perbandingan `cmp` dalam Assembly x86

- ***jle, jge, je, jne (conditional jumps):*** Melompat ke label lain jika flags kondisi memenuhi syarat, seperti yang ditunjukkan pada Kode 2.8.

```
jle less_or_equal ; Lompat jika kurang dari atau sama dengan
jne not_equal ; Lompat jika tidak sama dengan
```

Kode 2.8: Contoh Instruksi Lompatan Kondisional dalam Assembly x86

Loop Instructions (Implementasi dengan Conditional Jumps):

- Menggunakan kombinasi instruksi perbandingan, pengurangan counter, dan conditional jump untuk membuat loop (lihat contoh pada Kode 2.9).

```
mov ecx, 5 ; set counter loop
loop_start:
; instruksi dalam loop
cmp ecx, 0
jne loop_start
```

Kode 2.9: Contoh Implementasi Perulangan Sederhana dalam Assembly x86

Jump Instructions (Direct):

- ***jmp (unconditional jump):*** Lompat ke label tanpa syarat (Kode 2.10).

```
jmp target_label
```

Kode 2.10: Contoh Instruksi Lompatan Tanpa Syarat `jmp` dalam Assembly x86

Jump Instructions (Indirect - function calls):

- ***call:*** Melompat ke alamat fungsi dan menyimpan alamat return (Kode 2.11).

```
call function_address
```

Kode 2.11: Contoh Instruksi Pemanggilan Fungsi `call` dalam Assembly x86

- ***ret:*** Mengembalikan kontrol dari fungsi (Kode 2.12).

```
ret
```

Kode 2.12: Contoh Instruksi Pengembalian Fungsi `ret` dalam Assembly x86

2.3 Rekayasa balik (*Reverse Engineering*)

Rekayasa balik (reverse engineering) adalah proses menganalisis suatu sistem, baik perangkat lunak, perangkat keras, atau sistem lainnya, untuk mengidentifikasi komponen-komponennya dan interaksi antar komponen, serta memahami cara kerja sistem tersebut tanpa akses ke kode sumber [8]. Tujuannya beragam, mulai dari pemahaman fungsionalitas, analisis keamanan untuk menemukan kerentanan, pemulihan desain, hingga modifikasi dan peningkatan sistem. Rekayasa balik dapat diterapkan pada berbagai skenario, misalnya untuk menganalisis malware, memahami format file yang tidak terdokumentasi, mempelajari teknik yang digunakan oleh pesaing atau modifikasi fungsionalitas perangkat lunak.

2.3.1 Jenis Analisis Rekayasa Balik

1. **Analisis Statis**
   Analisis statis melibatkan pemeriksaan kode tanpa menjalankannya. Analisis statis berfokus pada struktur dan logika kode, mencari pola dan kerentanan [9]. Alat yang digunakan dalam analisis statis meliputi:

   - ***Disassembler:*** Menerjemahkan kode mesin menjadi assembly code, sebuah representasi kode yang lebih mudah dibaca oleh manusia. Contoh: IDA Pro, Ghidra, Radare2.
   - ***Decompiler:*** Menerjemahkan kode mesin atau bytecode kembali ke kode sumber tingkat tinggi (misalnya, C++ atau Java). Decompiler membantu memahami logika program secara lebih mudah.
   - ***Code Analysis Tools:*** Alat yang digunakan untuk menganalisis kode secara otomatis, mencari kerentanan keamanan, pola kode yang buruk, dan masalah lainnya.

Gambar 2.5: Dekompilasi Aplikasi [1]

2. **Analisis Dinamik**
   Analisis dinamis melibatkan menjalankan program dan mengamati perilakunya. Analisis dinamis berfokus pada bagaimana program berinteraksi dengan lingkungannya, mencari kerentanan runtime dan memahami alur eksekusi [9]. Alat yang digunakan dalam analisis dinamis meliputi:

   - ***Debugger:*** Memungkinkan eksekusi program secara terkontrol, memeriksa nilai variabel, dan melacak alur eksekusi. Contoh: x64dbg, OllyDbg, GDB.
   - ***Profiler:*** Mengukur penggunaan sumber daya program, seperti waktu eksekusi, penggunaan memori, dan akses file. Profiler membantu mengidentifikasi bottleneck kinerja.

2.3.2 *Tampering*

Tampering merupakan suatu proses mengubah kode aplikasi untuk mempengaruhi perilakunya. Contoh Tampering perangkat lunak adalah mengubah kode aplikasi agar dapat melewati proses autentikasi dalam perangkat lunak tersebut. Hal ini dapat dilakukan dengan memodifikasi control flow pada proses autentikasi dalam programnya.
Tampering bisa dilakukan pada berbagai tingkatan dan bagian dari perangkat lunak, termasuk

- **Kode biner (*executable code*):** Mengubah instruksi mesin yang dijalankan oleh prosesor. Hal ini bisa digunakan untuk mematikan fitur tertentu, menambahkan fitur baru, mengubah perilaku program, atau menyisipkan kode berbahaya.
- **Data** Mengubah data konfigurasi, aset game, atau data sensitif lainnya yang digunakan oleh program.
- **Pustaka (*Library*):** Memodifikasi library yang digunakan oleh program untuk mengubah fungsionalitasnya.
- **Sumber Daya (*Resources*):** Mengubah teks, gambar, atau elemen lain yang digunakan oleh program.

2.3.3 Alat-alat untuk Rekayasa Balik

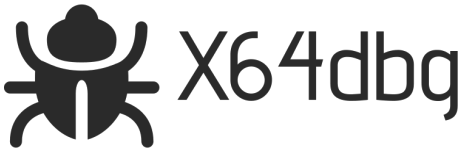
- **IDA Pro (Interactive Disassembler Pro):** Disassembler dan debugger komersial yang sangat populer dan powerful. IDA Pro mendukung berbagai arsitektur prosesor dan sistem operasi, menyediakan antarmuka yang canggih untuk analisis kode, dan mendukung plugin untuk ekstensibilitas [10].

Gambar 2.6: IDA Pro [10]

- **Ghidra:** Framework reverse engineering open-source yang dikembangkan oleh NSA (National Security Agency). Ghidra menawarkan fitur yang setara dengan IDA Pro, termasuk disassembler, decompiler, dan debugger. Ghidra juga mendukung scripting dan ekstensibilitas melalui plugin [11].

Gambar 2.7: Ghidra [11]

- **x64dbg:** Debugger open-source untuk platform Windows. x64dbg menyediakan antarmuka yang modern dan user-friendly, serta mendukung plugin dan scripting. x64dbg fokus pada analisis malware dan reverse engineering aplikasi Windows [12].



Gambar 2.8: x64dbg [12]

2.3.4 Disassembler: Statis dan Dinamis

Disassembler adalah alat fundamental dalam rekayasa balik yang berfungsi untuk menerjemahkan kode mesin (urutan byte yang dieksekusi langsung oleh prosesor) kembali ke dalam bentuk bahasa assembly yang dapat dibaca manusia [13]. Bahasa assembly adalah representasi simbolik tingkat rendah dari instruksi mesin, yang memberikan pandangan mendetail tentang operasi yang dilakukan oleh perangkat lunak. Pemahaman terhadap kode assembly hasil disassembly sangat krusial untuk menganalisis logika program, menemukan kerentanan, atau memahami bagaimana program berinteraksi dengan sistem. Terdapat dua pendekatan utama dalam disassembly: statis dan dinamis.

2.3.4.1 Disassembler Statis

Disassembler statis bekerja dengan menganalisis berkas executable (atau library) secara langsung di disk, tanpa perlu menjalankan program tersebut [14]. Alat ini membaca urutan byte dari berkas dan mencoba untuk memetakannya ke instruksi-instruksi assembly yang sesuai dengan arsitektur target (misalnya, x86, x86-64, ARM).

Cara Kerja Disassembler Statis: Proses disassembly statis umumnya melibatkan beberapa teknik, di antaranya:

- **Linear Sweep Disassembly (Pembongkaran Sapuan Linier):** Ini adalah pendekatan paling sederhana. Disassembler memulai dari alamat awal segmen kode (atau titik masuk yang diketahui) dan secara berurutan menerjemahkan byte demi byte menjadi instruksi assembly [13]. Pendekatan ini cepat tetapi memiliki kelemahan signifikan: ia tidak dapat dengan mudah membedakan antara kode dan data yang mungkin disisipkan di antara instruksi. Jika ada byte data yang secara keliru diinterpretasikan sebagai instruksi, sisa dari proses disassembly bisa menjadi tidak sinkron dan menghasilkan output yang salah.
- **Recursive Traversal Disassembly (Pembongkaran Penelusuran Rekursif):** Pendekatan ini lebih canggih dan akurat. Dimulai dari titik masuk yang diketahui, disassembler menelusuri alur kontrol program. Ketika instruksi percabangan (seperti `JMP`, `CALL`, atau `JNZ`) ditemui, disassembler akan mengikuti target dari percabangan tersebut dan melanjutkan proses disassembly dari sana [14], [15]. Teknik ini lebih baik dalam membedakan antara kode dan data karena hanya bagian yang dapat dijangkau melalui alur kontrol yang dianggap sebagai kode. Namun, ia dapat menghadapi kesulitan dengan percabangan tidak langsung (indirect jumps/calls) yang targetnya ditentukan saat runtime, atau teknik anti-disassembly yang sengaja mengacaukan alur kontrol.

Disassembler statis seperti Ghidra [11] dan IDA Pro [10] menggunakan kombinasi teknik ini, ditambah dengan analisis heuristik dan pemahaman struktur format berkas (seperti PE untuk Windows atau ELF untuk Linux) untuk mengidentifikasi segmen kode, data, tabel impor/ekspor, dan simbol-simbol debug (jika ada).

Keuntungan Disassembler Statis:

- **Analisis Komprehensif:** Dapat menganalisis seluruh program (atau bagian besar darinya) tanpa perlu mengeksekusinya, memberikan gambaran umum struktur dan logika.
- **Keamanan:** Tidak ada risiko menjalankan kode berbahaya karena program tidak dieksekusi.
- **Reproduktifitas:** Hasil analisis konsisten karena didasarkan pada berkas yang tidak berubah.

Keterbatasan Disassembler Statis:

- **Kode yang Dikemas (Packed) atau Dienkripsi:** Jika kode program dikemas (dikompresi dan mungkin dienkripsi) atau dilindungi oleh lapisan enkripsi, disassembler statis hanya akan melihat kode pembuka/dekriptornya, bukan kode asli program sampai kode tersebut didekripsi/dibuka saat runtime [13].
- **Kode yang Memodifikasi Diri Sendiri (Self-Modifying Code):** Disassembler statis tidak dapat secara akurat merepresentasikan kode yang mengubah instruksinya sendiri saat runtime.
- **Polimorfisme dan Metamorfisme:** Teknik yang digunakan malware untuk mengubah penampilan kode mereka di setiap infeksi atau eksekusi sangat menyulitkan analisis statis.
- **Percabangan Tidak Langsung dan Opaque Predicates:** Sulit untuk menentukan semua kemungkinan jalur eksekusi secara statis jika target lompatan atau kondisi percabangan sangat bergantung pada nilai runtime atau sengaja dibuat ambigu [14].
- **Kegagalan pada Kode Tervirtualisasi:** Ini adalah poin krusial terkait VxLang. Ketika kode asli telah ditransformasi menjadi bytecode untuk sebuah Instruction Set Architecture (ISA) kustom yang dijalankan oleh Virtual Machine (VM) internal, disassembler statis seperti Ghidra akan gagal. Ghidra dirancang untuk memahami ISA standar (seperti x86-64). Ia akan mencoba menginterpretasikan bytecode kustom dari VxLang sebagai instruksi x86-64, yang tentu saja tidak valid. Akibatnya, Ghidra akan menampilkan urutan byte yang tidak dikenali (sering direpresentasikan sebagai '???' atau instruksi yang salah) dan tidak mampu merekonstruksi alur logika asli program yang telah divirtualisasi. Proses virtualisasi oleh VxLang secara efektif menyembunyikan kode asli dari analisis statis konvensional.

2.3.4.2 Disassembler Dinamis (Debugger)

Disassembler dinamis, atau lebih tepatnya disassembly yang dilakukan secara dinamis, biasanya merupakan fitur integral dari sebuah debugger seperti x64dbg [12]. Proses disassembly ini terjadi saat program sedang berjalan (atau dijeda) di bawah kontrol debugger.

Cara Kerja Disassembler Dinamis: Ketika program dieksekusi di dalam debugger, debugger memiliki akses penuh ke ruang memori proses tersebut.

- **Pembongkaran Instruksi Saat Ini:** Debugger membaca byte-byte memori pada alamat yang ditunjuk oleh Instruction Pointer (misalnya, register RIP pada arsitektur x86-64) dan menerjemahkannya menjadi instruksi assembly yang akan dieksekusi berikutnya [13]. Ini dilakukan secara on-the-fly.

- **Penanganan Kode yang Di-unpack atau Didekripsi:** Jika program menggunakan packer atau enkripsi, disassembler dinamis akan melihat kode asli setelah packer tersebut selesai mengekstrak dan mendekripsi kode ke dalam memori.
- **Observasi Perubahan Kode:** Jika program menggunakan self-modifying code, disassembler dinamis (yang dipanggil berulang kali saat stepping) akan menampilkan instruksi yang benar-benar dieksekusi pada setiap langkah, bahkan jika instruksi tersebut baru saja dimodifikasi.

Keuntungan Disassembler Dinamis:

- **Mengatasi Kode Terproteksi (Packed/Encrypted):** Sangat efektif untuk menganalisis kode yang baru muncul di memori setelah proses unpacking atau dekripsi runtime.
- **Menangani Self-Modifying Code:** Dapat melihat bentuk kode yang sebenarnya dieksekusi.
- **Analisis Berdasarkan Eksekusi Nyata:** Hanya menganalisis kode yang benar-benar dijalankan, yang bisa lebih fokus.
- **Visibilitas Simbol Runtime:** Debugger dinamis, seperti x64dbg, beroperasi pada proses yang sedang berjalan dan memiliki akses ke informasi simbol yang dimuat oleh sistem operasi. Ini mencakup simbol dari Dynamic Link Libraries (DLL) sistem (misalnya, fungsi API Windows dari `kernel32.dll`) yang diselesaikan oleh OS loader saat runtime. Jika tersedia, berkas simbol debug (seperti PDB untuk Windows) juga dapat dimuat oleh debugger untuk menampilkan simbol-simbol internal aplikasi.

Keterbatasan Disassembler Dinamis:

- **Cakupan Terbatas:** Hanya bagian kode yang dieksekusi selama sesi debug yang dapat dianalisis. Mungkin ada bagian kode yang tidak pernah terjangkau.
- **Memakan Waktu:** Memerlukan interaksi manual untuk menelusuri kode, bisa lambat untuk program besar.
- **Teknik Anti-Debugging:** Program dapat menggunakan teknik untuk mendeteksi keberadaan debugger dan mengubah perilakunya atau menghentikan eksekusi [13].
- **Analisis Kode Tervirtualisasi oleh VxLang:** Meskipun x64dbg dapat melakukan disassembly terhadap instruksi-instruksi dari *interpreter* atau *Virtual Machine*

(VM) milik VxLang itu sendiri (karena VM tersebut adalah kode native), ia tidak akan secara langsung menampilkan kode asli Anda yang telah ditransformasi menjadi bytecode. Yang terlihat adalah kode dari VM yang sedang mengeksekusi bytecode tersebut. Jadi, meskipun x64dbg dapat melihat instruksi yang berjalan, instruksi tersebut bukanlah logika asli program dalam bentuk assembly aslinya, melainkan instruksi-instruksi dari lapisan virtualisasi VxLang. Ini tetap menyulitkan pemahaman logika asli secara langsung.

Secara ringkas, disassembler statis memberikan gambaran global namun rentan terhadap teknik obfuscation canggih seperti virtualisasi kode. Di sisi lain, disassembler dinamis (dalam debugger) dapat mengatasi beberapa bentuk proteksi runtime dan melihat simbol yang dimuat oleh OS, tetapi untuk kode yang divirtualisasi seperti oleh VxLang, ia akan menampilkan kode dari VM protektor, bukan kode asli yang telah diubah menjadi bytecode. Kegagalan Ghidra pada aplikasi yang divirtualisasi VxLang adalah karena Ghidra mengharapkan instruksi mesin standar, bukan bytecode dari ISA kustom yang hanya dipahami oleh VM VxLang.

2.4 *Obfuscation*

Obfuscation adalah teknik yang digunakan untuk mengubah kode sumber atau kode mesin menjadi bentuk yang lebih sulit dipahami oleh manusia, tanpa mengubah fungsionalitas program. Tujuan utama obfuscation adalah untuk mempersulit analisis dan reverse engineering, melindungi kekayaan intelektual, dan meningkatkan keamanan aplikasi. Obfuscation tidak membuat kode menjadi tidak mungkin untuk di-reverse engineer, tetapi meningkatkan waktu dan usaha yang dibutuhkan untuk melakukannya, sehingga membuat reverse engineering menjadi kurang menarik bagi penyerang [16].

2.4.1 Manfaat *Obfuscation*

- **Meningkatkan Keamanan:** Obfuscation mempersulit penyerang untuk memahami logika program, menemukan kerentanan, dan memodifikasi kode untuk tujuan jahat. Obfuscation dapat melindungi algoritma penting, kunci enkripsi, dan data sensitif lainnya.
- **Melindungi Kekayaan Intelektual:** Obfuscation dapat mempersulit pesaing untuk mencuri kode sumber, meniru fungsionalitas program, dan melanggar hak cipta. Ini penting terutama untuk perangkat lunak komersial dan aplikasi yang mengandung algoritma atau teknologi yang unik.

2.4.2 Jenis-jenis *Obfuscation*

Obfuscation dapat dilakukan pada diterapkan pada kode sumber, bytecode, atau kode biner.

2.4.2.1 Kode Sumber *Obfuscation*

Teknik ini mengubah kode sumber yang dapat dibaca manusia, sehingga sulit dipahami tanpa memengaruhi fungsinya [16].

- ***Layout obfuscation:*** mengubah tampilan kode.
  - ***Scrabling identifiers [17]:*** mengubah nama fungsi dan variabel.
  - ***Changing formatting [18]:*** menambahkan atau menghapus spasi putih dan baris baru.
  - ***Removing comments [18]:*** menghapus komentar penjelasan.
- ***Data obfuscation:*** menyembunyikan cara data disimpan dan diproses.
  - ***Data encocding [19]:*** Mengubah representasi data, misalnya mengenkripsi string atau mengubah nilai numerik dengan operasi matematika. Ini membuat data asli sulit dikenali langsung.
  - ***Instruction Substitution [20]*** Mengganti instruksi yang sederhana dengan instruksi yang lebih kompleks atau setara, tetapi lebih sulit dipahami.
  - ***Mixed boolean arithmetic [21], [22]*** Menggunakan operasi boolean (AND, OR, XOR, NOT) yang dikombinasikan dengan operasi aritmatika untuk membuat logika program lebih kompleks dan sulit diurai.
- ***Control flow obfuscation:*** mempersulit logika program.
  - ***Bogus control flow [23]:*** menambahkan kode palsu yang memengaruhi alur kontrol.
  - ***Opaque predicates [24]:*** menyisipkan kode sampah ke dalam pernyataan kondisional.
  - ***Control Flow flattening [25]:*** mengubah struktur program menjadi pernyataan switch yang kompleks.

2.4.2.2 *Bytecode Obfuscation*

Bytecode Obfuscation beroperasi pada kode perantara yang dihasilkan setelah kompilasi kode sumber. Hal ini khususnya relevan untuk bahasa seperti Java, .NET, LLVM dimana kode dikompilasi menjadi bytecode dan kemudian dijalankan pada mesin virtual [26] [27]. Tujuannya adalah untuk mempersulit rekayasa balik bytecode menjadi kode sumber dengan mudah.
Berikut Teknik-teknik obfuscation pada bytecode :

- ***Renaming [26] :*** Mengubah nama kelas, metode, dan variabel dalam bytecode untuk membuat kode sumber yang didekompilasi lebih sulit dibaca. Misalnya, metode bernama calculateSalary dapat diubah namanya menjadi method1
- ***Control Flow Obfucsation [26] :*** Meggunakna percabangan yang kompleks, kondisional, dan konstruksi berulang untuk membuat kode yang didekompilasi menjadi non-deterministik dan lebih sulit untuk diikuti
- ***String Encryption [26] :*** Mengenkripsi string yang tertanam dalam bytecode, yang hanya didekripsi saat dijalankan saat dibutuhkan. Hal ini mempersulit pencarian informasi atau data sensitif dengan menganalisis bytecode.
- ***Dummy Code Insertion [26] :*** Menambahkan kode yang tidak memengaruhi logika program, tetapi mempersulit dekompilasi dan analisis

2.4.2.3 *Kode Biner Obfuscation*

Kode Biner Obfuscation diterapkan pada kode akhir yang dapat dieksekusi mesin. Ini berfokus pada upaya membuat biner sulit dianalisis, dibongkar, dan dipahami. Berikut beberapa teknik yang digunakna untuk obfuscation pada kode biner :

- ***Code Packing [28] :*** kode asli yang dapat dieksekusi dikompresi atau dienkripsi menjadi biner yang dikemas, yang juga menyertakan kode bootstrap yang membongkar kode asli ke dalam memori saat runtime, sehingga melindungi kode asli dari analisis statis.
- ***Obfuscated Control Flow [28] :*** Menggunakan inderect calls dan jumps, dan memanipulasi instruksi panggilan dan ret untuk mempersulit mengikuti jalur eksekusi program.
- ***Chunked Control Flow [28] :*** Membagi kode menjadi blok-blok yang sangat kecil dengan instruksi lompatan antar blok untuk mengganggu pembongkaran linear assembly.

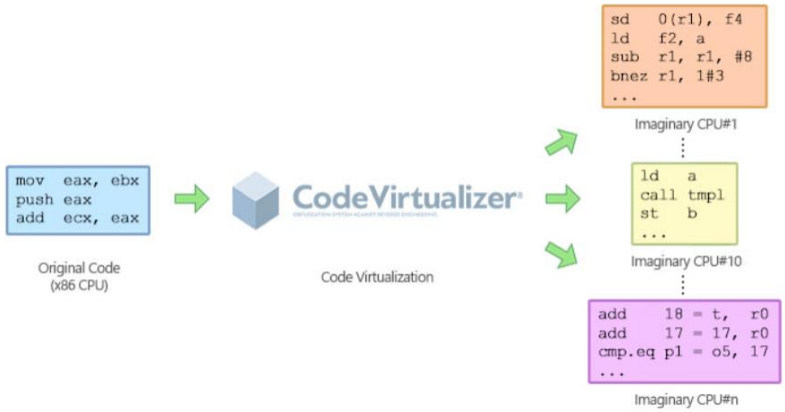
- ***Obfuscated Constants [28] :*** Menyembunyikan nilai konstan yang digunakan dalam kode melalui berbagai operasi aritmatika atau logika.
- ***Code Virtualization [1] :*** Menerjemahkan kode mesin menjadi instruksi virtual yang dieksekusi oleh mesin virtual khusus yang tertanam dalam aplikasi.

Gambar 2.9: Prosess Code Virtualization [1]

2.5 *Code Virtualization*

Code Virtualization atau juga disebut dengan VM-Based Code Obfuscation menerupakan suatu teknik obfuscation dimana kita menerjemahkan kode biner orignal menjadi byte-code baru berdasarkan Instruction Set Architecture (ISA) khusus (Gambar 2.9). Bytecode ini dapat dijalankan secara run-time dengan mesin virtual (i.e. intrepreter) yang tertanam pada aplikasinya [1], [29]. Teknik Code virtualization tidak akan mengembalikan sumber kode aslinya dalam memori sedangkan teknik Code Encryption tetap mengembalikan kodenya aslinya dalam memori saat dilakukan dekripsinya [30].

Set instruksi virtual ini merupakan kunci dari obfuscationnya yang digunakan untuk mapping relasi antara intruksi lokal dan instruksi virtual [29] . Intruksi virtual ini membuat aliran kontrol dari original program untuk tidak dapat dibaca dan mempersulit reverse-engineer program. Code Virtualization dapat menghasilkan berbagai mesin virtual dengan set instruksi virtual masing-masing. Dengan ini, setiap copy dari program dapat memiliki instruksi virtual khusus agar mencegah penyerang untuk mengenali op-code mesin virtual menggunakan metode Frequency Analysis [1], [29].



Gambar 2.10: Transformasi ISA x86 menjadi berbagai messin virtual [1]

Arsitektur umum mesin virtual untuk Code Virtualization memiliki komponen-komponen yang mirip dengan design CPU [31], [32].

1. ***VM entry:*** Perannya untuk simpan konteks eksekusi native (seperti register CPU atau flag) dan transisi ke konteks mesin virtual.

2. ***Fetch:*** Perannya adalah untuk mengambil, dari memori internal VM, opcode (virtual) yang akan ditiru, berdasarkan nilai Virtual Program Counter (vpc).

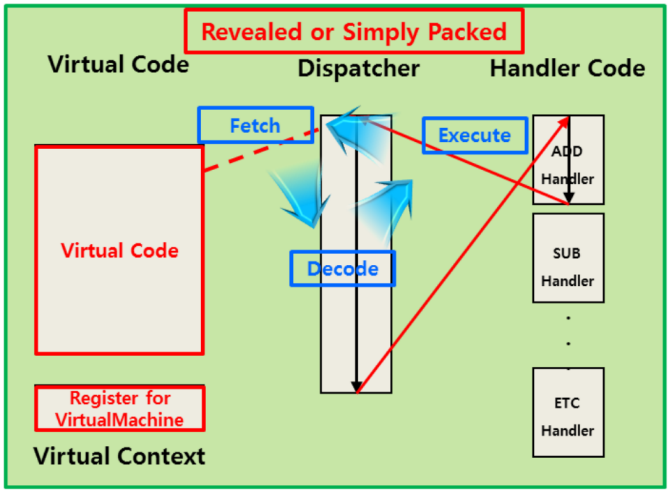
3. ***Decode:*** Perannya adalah untuk mendekode opcode yang diambil dan operan yang sesuai untuk menentukan instruksi ISA mana yang akan dieksekusi.

4. ***Dispatch:*** Setelah instruksi didekodekan, operator menentukan pengendali mana yang harus dijalankan dan mengatur konteksnya.

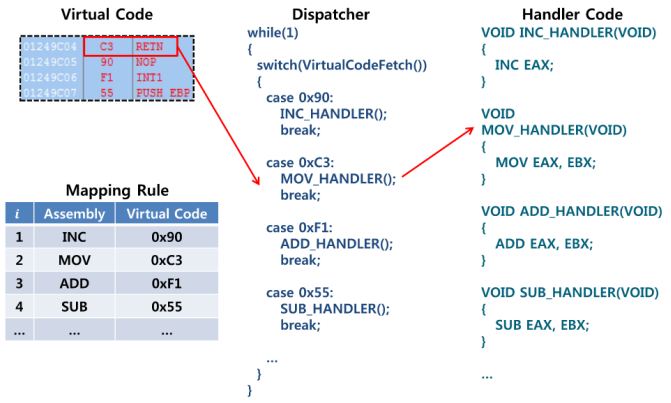
5. ***Handlers:*** Meniru instruksi virtual melalui rangkaian instruksi asli dan memperbarui konteks internal VM, biasanya vpc.

6. ***VM exit:*** Perannya untuk transisi dari konteks mesin virtual balik ke konteks eksekusi native.

Proses eksekusi Code Virtualization pada program dapat dilihat pada Gambar 2.11 dan contoh code virtualization pada intel assembly dapat dilihat pada Gambar 2.12.



Gambar 2.11: Alur eksekusi Code Virtualization [30]



Gambar 2.12: Transformasi kode asli menjadi kode virtual [30]

2.5.1 VxLang

VxLang adalah sebuah proyek obfuscation kode atau biner yang bertujuan untuk mencegah attacker melakukan tindakan reverse engineering, seperti analisis statis atau dinamis, perusakan file, dan akses ilegal ke memori. VxLang ini merupakan sebuah perangkat komprehensif yang terdiri dari packer/protector, alat obfuscation kode, dan alat virtualisasi kode. Proyek ini saat ini menargetkan executable Microsoft Windows PE (.exe/.dll/*.sys dan UEFI) pada x86-64, dengan dukungan untuk Linux ELF dan ARM yang direncanakan di masa mendatang.

2.5.1.1 Komponen-Komponen Utama VxLang

VxLang terdiri dari beberapa komponen utama yang bekerja sama untuk melindungi kode:

- ***Binary Protector:*** Komponen ini memodifikasi, mengompresi, dan mengenkripsi executable asli menjadi format file yang digunakan secara internal oleh VxLang. Binary Protector juga menggabungkan kode inti VxLang, yang dapat diperkuat dan diperluas dengan modul tambahan. Struktur fleksibel ini memungkinkan pengguna untuk dengan bebas memperluas fungsionalitas dan mengontrol inti VxLang.

- ***Code Obfuscator:*** Alat Obfuscation Kode ini mengubah kode native untuk menghalangi analisis, baik dengan memecah blok kode atau meratakan kode ke kedalaman yang sama. Metode obfuscation tambahan akan ditambahkan di masa mendatang. Obfuscation bertujuan untuk membuat kode lebih sulit dipahami tanpa mengubah fungsionalitasnya.

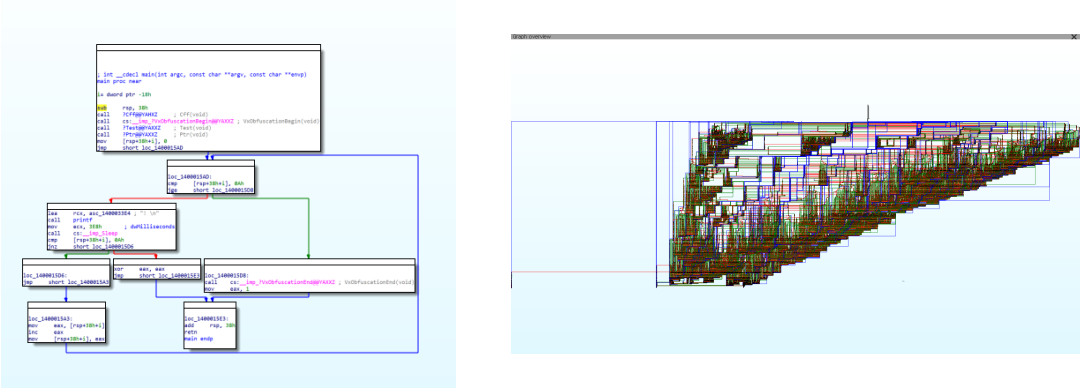
- ***Code Virtualizer:*** Alat Virtualisasi Kode ini mengubah kode menjadi bahasa assembly internal. Bahasa ini kemudian dikonversi menjadi bytecode dan dieksekusi oleh compiler internal. Virtualisasi kode menambahkan lapisan abstraksi yang signifikan, sehingga mempersulit reverse engineering karena attacker harus memahami mesin virtual VxLang untuk dapat memahami kode yang dilindungi.

2.5.1.2 Arsitektur dan Cara Kerja VxLang

- **Proteksi Biner:** Binary Protector mengambil executable asli dan mengubahnya menjadi format internal VxLang. Proses ini dapat mencakup kompresi untuk mengurangi ukuran file, enkripsi untuk melindungi konten, dan modifikasi struktur file untuk mempersulit analisis statis. Dengan mengubah format file, VxLang mempersulit alat reverse engineering standar untuk memproses dan menganalisis executable.

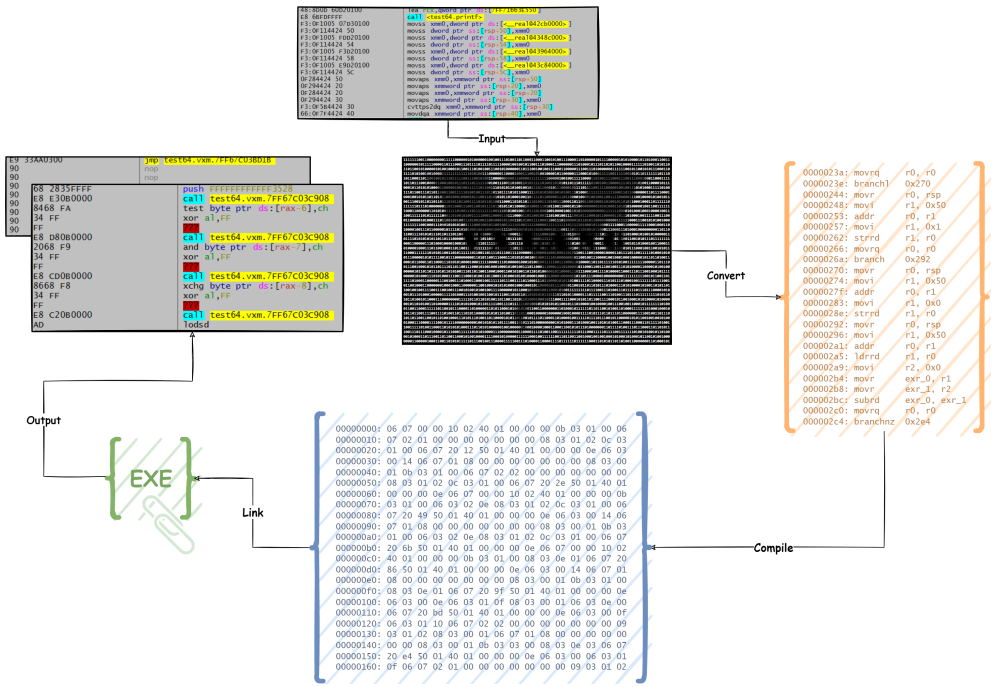
- **Obfuscation Kode:** Code Obfuscator mengubah kode native untuk membuatnya lebih sulit dipahami. Teknik-teknik yang digunakan dapat mencakup:

  - **Memecah blok kode:** Ini melibatkan pemisahan blok kode yang berurutan menjadi bagian-bagian yang lebih kecil dan menyisipkan lompatan di antara bagian-bagian tersebut. Hal ini mengganggu alur kontrol program dan mempersulit reverse engineer untuk mengikuti logika program.
  - **Meratakan kode ke kedalaman yang sama (Code Flattening):** Ini melibatkan pengubahan struktur alur kontrol bersarang (misalnya, loop dan percabangan) menjadi urutan instruksi yang lebih datar. Hal ini menghilangkan hierarki dalam kode dan membuatnya lebih sulit untuk dipahami.



Gambar 2.13: Sebelum dan Sesudah *Obfuscation* [2]

- **Virtualisasi kode:** Code Virtualizer mengubah kode native menjadi bytecode untuk mesin virtual VxLang. Bytecode ini kemudian dieksekusi oleh interpreter mesin virtual. Virtualisasi kode menambahkan lapisan abstraksi yang signifikan, karena attacker tidak lagi dapat menganalisis kode native secara langsung. Sebaliknya, mereka harus memahami arsitektur dan set instruksi mesin virtual VxLang, yang tidak terdokumentasi dan dapat dikustomisasi.



Gambar 2.14: VxLang *Code Virtualizer* [2]

2.5.1.3 VxLang SDK dan API untuk Obfuscation

Untuk menggunakan VxLang, pengembang dapat menggunakan Software Development Kit (SDK) yang tersedia. SDK ini dapat diperoleh dengan mengunduh library yang telah

dikompilasi atau dengan mendapatkan kode sumber dari GitHub dan membangunnya sendiri. Setelah mendapatkan SDK, langkah selanjutnya adalah mengatur header dan library links dalam kode proyek.

Di dalam header SDK, terdapat beberapa Application Programming Interfaces (API) yang berperan penting dalam melakukan obfuscation kode. API-API ini memungkinkan pengembang untuk menerapkan berbagai teknik obfuscation secara terarah pada kode sumber. Berikut adalah penjelasan mengenai API-API tersebut:

- **VL_OBFUSCATION_BEGIN/END:** API ini digunakan untuk menambahkan kode dummy (kode palsu) di antara kode asli. Penambahan kode dummy ini bertujuan untuk mengganggu proses disassembly dan decompilation. Disassembly adalah proses menerjemahkan kode mesin kembali ke dalam bentuk assembly, sedangkan decompilation adalah proses menerjemahkan kode mesin atau bytecode ke dalam bahasa pemrograman tingkat tinggi. Dengan adanya kode dummy, reverse engineer akan kesulitan untuk memahami alur eksekusi program yang sebenarnya karena kode dummy tersebut akan muncul dalam hasil disassembly dan decompilation, sehingga mengaburkan kode asli. Contoh penggunaannya dapat dilihat pada Kode 2.13.

```
void obfuscation_test(int x) {

  VL_OBFUSCATION_BEGIN;

  printf("Obfuscation Test!")

  VL_OBFUSCATION_END;

}
```

Kode 2.13: Contoh Penggunaan Makro `VL_OBFUSCATION_BEGIN/END` dari VxLang SDK

- **VL_CODE_FLATTENING_BEGIN/END:** API ini digunakan untuk mengacak urutan kode pada tingkat kedalaman yang sama. Proses ini juga melibatkan penambahan kode dummy dan membuat kode menjadi lebih sulit diinterpretasikan. Code flattening adalah teknik obfuscation yang mengubah struktur alur kontrol program menjadi lebih datar dan kompleks. Dengan mengacak urutan kode dan menambahkan kode dummy, reverse engineer akan kesulitan untuk merekonstruksi alur logika program yang sebenarnya. Ilustrasi penggunaan makro ini ditunjukkan pada Kode 2.14.

```
void code_flattening_test(int x) {
```

```
  VL_CODE_FLATTENING_BEGIN;

  printf("Code Flattening Test!")

  VL_CODE_FLATTENING_END;

}
```

Kode 2.14: Contoh Penggunaan Makro `VL_CODE_FLATTENING_BEGIN/END` dari VxLang SDK

- **VL_VIRTUALIZATION_BEGIN/END:** API ini digunakan untuk mengubah kode native yang ditentukan menjadi bytecode yang hanya dapat diinterpretasikan oleh Central Processing Unit (CPU) internal VxLang. Ini adalah inti dari teknik virtualisasi kode yang ditawarkan oleh VxLang. Kode native yang telah diubah menjadi bytecode tidak lagi dapat dieksekusi secara langsung oleh prosesor fisik, melainkan harus melalui interpreter mesin virtual VxLang. Ini secara signifikan meningkatkan kesulitan reverse engineering karena attacker harus memahami arsitektur dan set instruksi mesin virtual VxLang untuk dapat memahami kode yang dilindungi. Kode 2.15 memberikan contoh implementasinya.

```
void virtualization_test(int x) {

  VL_VIRTUALIZATION_BEGIN;

  printf("Virtualization Test!")

  VL_VIRTUALIZATION_END;

}
```

Kode 2.15: Contoh Penggunaan Makro `VL_VIRTUALIZATION_BEGIN/END` dari VxLang SDK

API-API ini dapat diterapkan pada kode sumber dengan cara menandai bagian-bagian kode yang ingin diobfuskasi atau divirtualisasi dengan menggunakan API BEGIN dan END yang sesuai.

2.5.1.4 Proses Virtualisasi VxLang

Setelah kode sumber ditandai menggunakan makro SDK dan dikompilasi menjadi *executable intermediate* yang tertaut dengan library VxLang, langkah selanjutnya adalah pemrosesan menggunakan *tool* VxLang. *Tool* ini dijalankan melalui *command-line* secara langsung pada *executable intermediate*. Contohnya:

```
vxlang.exe target_intermediate.exe
```

Tool ini akan menganalisis penanda virtualisasi dalam *executable intermediate*, menggantikan kode *native* yang ditandai dengan representasi *bytecode* internal, dan menyematkan *runtime Virtual Machine* (VM) yang diperlukan. Proses ini secara otomatis menghasilkan *executable* akhir yang telah tervirtualisasi, biasanya dengan menambahkan sufiks `_vxm` pada nama file asli (misalnya, `target_intermediate_vxm.exe`). *Executable* inilah yang kemudian digunakan untuk distribusi atau pengujian lebih lanjut. Detail mengenai opsi *command-line* lanjutan atau kustomisasi (jika ada) berada di luar cakupan tinjauan ini.

2.6 *Graphical User Interface Frameworks*

Dalam pengembangan aplikasi autentikasi sebagai bagian dari demonstrasi pada penelitian ini, digunakan dua framework antarmuka pengguna grafis (GUI) yang berbeda, yaitu Qt Framework dan Dear ImGUI.

2.6.1 *Qt Framework*

Qt Framework adalah sebuah framework pengembangan aplikasi lintas platform yang sangat populer dan komprehensif. Berdasarkan dokumentasi [33], Qt menyediakan berbagai macam tool dan library yang diperlukan untuk mengembangkan aplikasi dengan antarmuka pengguna yang kaya dan interaktif, serta fungsionalitas non-GUI seperti akses database, jaringan, dan lainnya.
Karakteristik *Qt Framework*:

- ***Cross Platform :*** Salah satu keunggulan utama Qt adalah kemampuannya untuk berjalan di berbagai sistem operasi seperti Windows, Linux, macOS, Android, iOS, dan bahkan sistem embedded. Ini memungkinkan pengembang untuk menulis kode sekali dan menjalankannya di berbagai platform tanpa perlu modifikasi signifikan.
- **Fitur yang Kaya :** Qt menyediakan berbagai macam widget dan layout untuk membangun antarmuka pengguna yang kompleks dan menarik. Selain itu, Qt juga menawarkan dukungan yang kuat untuk grafik 2D dan 3D, animasi, multimedia, dan integrasi dengan teknologi web.
- **Bahasa Pemrograman Utama :** Qt umumnya digunakan dengan bahasa pemrograman C++, namun juga memiliki binding untuk bahasa lain seperti Python (melalui PyQt atau PySide).

Dalam konteks ini, Qt Framework digunakan untuk membangun antarmuka pengguna yang lebih tradisional dan mungkin lebih kompleks untuk aplikasi autentikasi. Ini memungkinkan visualisasi alur kerja autentikasi, input pengguna (seperti username dan

password), dan tampilan hasil autentikasi dengan cara yang terstruktur dan mudah dipahami.

Gambar 2.15: Qt Logo [33]

2.6.2 *Dear ImGUI*

Dear ImGUI adalah sebuah library antarmuka pengguna grafis (GUI) immediate mode yang ringan dan tidak memerlukan banyak boilerplate untuk digunakan. Berdasarkan repositori GitHub-nya [34], Dear ImGUI dirancang untuk fokus pada kesederhanaan dan kemudahan integrasi ke dalam aplikasi yang sudah ada, terutama untuk keperluan debugging, tooling, atau prototipe cepat.
Karakteristik *Dear ImGUI*:

- ***Immediate Mode GUI :*** Berbeda dengan retained mode GUI seperti Qt, di mana widget dibuat dan dikelola secara eksplisit, Dear ImGUI bekerja dengan cara menggambar ulang seluruh antarmuka pada setiap frame. Ini membuat pengembangan UI menjadi lebih intuitif dan stateful secara implisit.
- **Ringan dan Cepat :** Dear ImGUI memiliki ukuran library yang kecil dan performa yang baik, sehingga cocok untuk aplikasi yang membutuhkan sumber daya minimal atau yang sudah memiliki rendering engine sendiri.
- **Fokus pada Fungsionalitas Inti :** Dear ImGUI menyediakan berbagai widget dasar seperti tombol, slider, teks input, dan window, yang cukup untuk membangun antarmuka pengguna yang fungsional untuk keperluan demonstrasi atau tooling.

Dear ImGUI digunakan untuk menampilkan informasi atau kontrol yang lebih teknis atau diagnostik terkait dengan proses autentikasi yang dilindungi oleh VxLang. Sifatnya yang ringan dan mudah diintegrasikan mungkin menjadikannya pilihan yang baik untuk menampilkan proses autentikasi.

2.7 OpenSSL

OpenSSL adalah sebuah toolkit yang kuat, berkualitas komersial, dan kaya fitur untuk protokol Transport Layer Security (TLS) dan Secure Sockets Layer (SSL). OpenSSL

merupakan fondasi kriptografi open-source yang banyak digunakan dalam berbagai aplikasi dan sistem untuk mengamankan komunikasi melalui jaringan [35].

Karakteristik OpenSSL:

- **Toolkit Kriptograf :** OpenSSL menyediakan implementasi dari berbagai algoritma kriptografi, termasuk algoritma simetris (seperti AES), algoritma asimetris (seperti RSA dan ECC), fungsi hash kriptografi (seperti SHA-256), dan banyak lagi. Ini menjadikannya pilihan yang fleksibel untuk berbagai kebutuhan keamanan.
- **Implementasi Standar Keamanan :** OpenSSL mengimplementasikan protokol keamanan standar seperti TLS dan SSL, yang digunakan untuk mengenkripsi komunikasi antara klien dan server di internet, seperti pada protokol HTTPS.

Gambar 2.16: OpenSSL Logo [35]

OpenSSL dalam penelitian digunakan untuk menjalankan algoritma enkripsi AES (Advanced Encryption Standard) dengan mode CBC (Cipher Block Chaining) dan panjang kunci 256-bit digunakan dari library OpenSSL. AES adalah algoritma enkripsi simetris yang sangat aman dan banyak digunakan. Mode CBC merupakan salah satu mode operasi blok cipher yang umum digunakan dan memerlukan Initialization Vector (IV) acak untuk setiap proses enkripsi guna meningkatkan keamanan. Panjang kunci 256-bit memberikan tingkat keamanan yang sangat tinggi terhadap serangan brute-force.

Penggunaan enkripsi AES-CBC-256 dari OpenSSL dalam percobaan ini bertujuan untuk mengukur performance overhead (beban kinerja) yang mungkin ditimbulkan oleh penerapan VxLang. Dengan membandingkan waktu eksekusi operasi enkripsi yang sama sebelum dan sesudah kode dilindungi oleh VxLang, dapat dianalisis dampak virtualisasi kode terhadap kinerja aplikasi. Pemilihan algoritma AES-CBC-256 sebagai tolok ukur kemungkinan didasarkan pada popularitasnya, tingkat keamanannya, dan ketersediaannya dalam library OpenSSL yang banyak digunakan.

BAB 3
METODE PENELITIAN

Bab ini menyajikan kerangka kerja metodologis yang digunakan untuk mencapai tujuan penelitian, yaitu mengimplementasikan dan menganalisis efektivitas *code virtualization* menggunakan VxLang dalam meningkatkan keamanan perangkat lunak terhadap *reverse engineering*. Pembahasan mencakup pendekatan penelitian, desain eksperimen, objek studi, instrumen dan bahan penelitian, prosedur pengumpulan data yang direncanakan termasuk ilustrasi alur kerja, serta teknik analisis data.

3.1 Pendekatan Penelitian

Penelitian ini menggunakan pendekatan **eksperimental kuantitatif**. Efektivitas VxLang dalam mempersulit *reverse engineering* akan dievaluasi secara kuantitatif dan melalui observasi sistematis. Ini mencakup pengukuran keberhasilan atau kegagalan upaya *bypass* autentikasi, analisis terhadap kemampuan *tool* analisis (*disassembler*, *debugger*) dalam memproses dan menginterpretasi kode (misalnya, jumlah instruksi yang berhasil di-*disassemble*, jumlah *string* kritis yang teridentifikasi), serta perbandingan waktu atau langkah yang diperlukan untuk mencapai tujuan analisis tertentu (jika dapat diukur secara konsisten). Observasi ini akan dicatat secara objektif untuk mendukung analisis.

3.2 Desain Eksperimen

Desain eksperimen yang digunakan adalah **perbandingan antara kelompok kontrol dan kelompok eksperimen** pada serangkaian objek studi.

- **Kelompok Kontrol:** Aplikasi studi kasus dan *benchmark* yang dikompilasi secara normal tanpa penerapan *code virtualization* VxLang.
- **Kelompok Eksperimen:** Aplikasi studi kasus dan *benchmark* yang sama, namun bagian kode kritikalnya telah diproses menggunakan *code virtualization* VxLang.

Variabel dalam penelitian ini adalah:

- **Variabel Independen:** Penerapan *code virtualization* VxLang (diterapkan vs tidak diterapkan).
- **Variabel Dependen:**

- Tingkat kesulitan *reverse engineering* logika autentikasi.
- Performa perangkat lunak (waktu eksekusi dan ukuran file).

Eksperimen akan difokuskan pada dua aspek utama: analisis keamanan pada fungsi autentikasi dan analisis *overhead* performa pada tugas komputasi spesifik.

3.3 Objek Studi

Objek studi yang akan dikembangkan dan dianalisis dalam penelitian ini meliputi:

1. **Aplikasi Studi Kasus Autentikasi:** Aplikasi simulasi login dalam tiga varian antarmuka (Konsol, Qt, Dear ImGui) dan dua mekanisme autentikasi (Kredensial *hardcoded*, Kredensial via *backend cloud*). Aplikasi ini akan menjadi target utama untuk analisis *reverse engineering* mendalam guna mengevaluasi efektivitas VxLang dalam mempersulit pemahaman dan manipulasi logika.

2. **Aplikasi Benchmark Performa:** Aplikasi untuk mengukur *overhead* pada tugas spesifik (Algoritma QuickSort, Enkripsi AES-CBC-256, Ukuran File Dasar).

3. **Studi Kasus Aplikasi Potensial Berbahaya dan Malware:** Kelompok objek studi ini bertujuan untuk mengevaluasi dampak virtualisasi VxLang pada perangkat lunak yang lebih kompleks dan memiliki karakteristik yang sering diasosiasikan dengan *malware* atau *Potentially Unwanted Application* (PUA). Ini mencakup:

    - **Lilith RAT:** Sebuah *Remote Administration Tool* (RAT) *open-source* [36]. Klien Lilith RAT akan diuji fungsionalitasnya secara menyeluruh setelah virtualisasi untuk memastikan integritas operasional. Selain itu, baik versi asli maupun tervirtualisasi akan dianalisis dampaknya terhadap deteksi oleh layanan pemindaian VirusTotal.
    - **Sembilan Sampel Malware/PUA Tambahan:** Untuk memperluas analisis dampak terhadap deteksi otomatis, sembilan sampel *malware* atau PUA lainnya akan disertakan. Sampel-sampel ini adalah: *Al-Khaser, donut, DripLoader, FilelessPELoader, JuicyPotato, ParadoxiaClient, PELoader, RunPE-In-Memory,* dan *SigLoader*. Untuk sampel-sampel ini, fokus utama adalah perbandingan tingkat deteksi oleh VirusTotal antara versi asli (non-virtualized) dan versi yang telah diproses dengan VxLang.

Semua objek studi (aplikasi autentikasi, benchmark, klien Lilith RAT, dan sembilan sampel malware/PUA lainnya) akan disiapkan dalam versi asli (non-*virtualized*) dan versi *virtualized* oleh VxLang untuk perbandingan. Server Lilith RAT akan digunakan dalam versi aslinya untuk pengujian fungsionalitas klien.

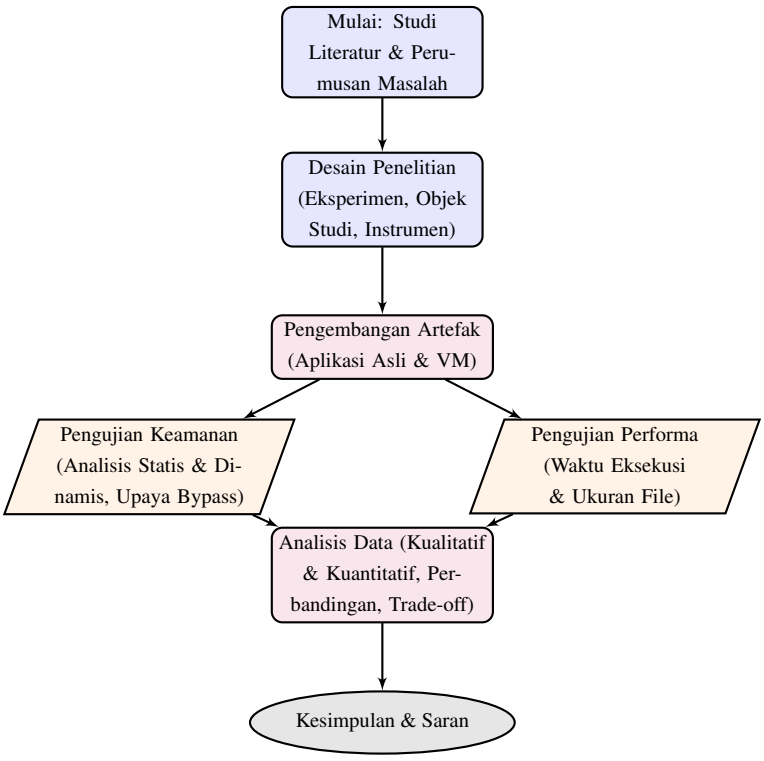
3.4 Instrumen dan Bahan Penelitian

Penelitian ini direncanakan menggunakan instrumen dan bahan berikut:

- **Perangkat Keras:** Komputer berbasis Windows.
- **Perangkat Lunak Pengembangan:** Compiler C++ (Clang/clang-cl), Build System (CMake, Ninja), Library/Framework (Qt, Dear ImGui, OpenSSL, libcurl, dll.), VxLang SDK, Teknologi Backend (Go, PostgreSQL, Docker).
- **Instrumen Analisis Keamanan:** Alat Analisis Statis (Ghidra), Alat Analisis Dinamis (x64dbg).
- **Instrumen Pengukuran Performa:** Library C++ (`std::chrono`, `std::filesystem`).
- **Lembar Observasi:** Untuk pencatatan kualitatif.

3.5 Prosedur Pengumpulan Data

Pengumpulan data akan mengikuti prosedur terstruktur yang mencakup studi literatur, persiapan artefak, pelaksanaan pengujian, dan pencatatan hasil. Alur kerja umum penelitian ini diilustrasikan pada Gambar 3.1.



Gambar 3.1: Diagram Alur Umum Tahapan Penelitian.

3.5.1 Studi Literatur

Tahap awal meliputi peninjauan literatur terkait *reverse engineering*, teknik *obfuscation*, *code virtualization* (khususnya VxLang), analisis statis/dinamis, dan metodologi pengukuran performa untuk membangun landasan teori dan memahami penelitian terkait.

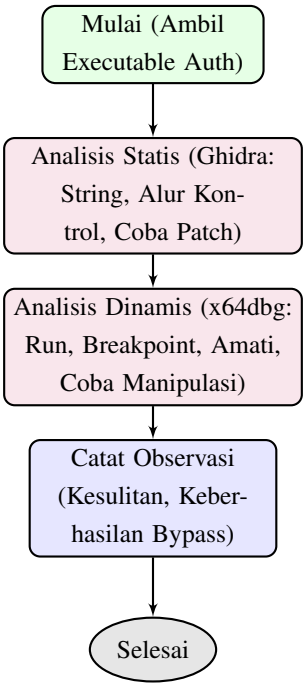
3.5.2 Persiapan Artefak

Tahap ini berfokus pada implementasi teknis objek studi. Ini mencakup pengembangan kode sumber untuk aplikasi autentikasi dan *benchmark* performa, integrasi VxLang SDK ke dalam kode, konfigurasi sistem *build* (CMake) untuk menghasilkan versi asli dan versi *intermediate* (dengan penanda VxLang), serta proses kompilasi. Versi *intermediate* kemudian diproses langsung menggunakan *tool command-line* `vxlang.exe`, yang secara otomatis menghasilkan *executable* akhir tervirtualisasi dengan nama `(nama_target)_vxm.exe`. Detail implementasi disajikan pada Bab 4.

3.5.3 Pengujian Keamanan Autentikasi

Pengujian ini bertujuan untuk mengevaluasi efektivitas VxLang dalam mempersulit analisis dan manipulasi logika autentikasi. Prosedur yang akan diikuti untuk setiap aplikasi autentikasi (asli dan *virtualized*) diilustrasikan pada Gambar 3.2 dan mencakup langkah-langkah berikut:

- **Analisis Statis (Ghidra):** Memuat *executable*, mencari *string* relevan, menganalisis *disassembly/decompilation* untuk memahami alur kontrol, mengidentifikasi instruksi kondisional kritis, dan mencoba melakukan *patching* statis untuk *bypass* autentikasi.
- **Analisis Dinamis (x64dbg):** Menjalankan *executable* dalam *debugger*, mencari *string/pattern* saat *runtime*, mengatur *breakpoint*, mengamati alur eksekusi dan nilai memori/register, serta mencoba melakukan manipulasi *runtime* (misalnya, *patching on-the-fly* atau mengubah *flags*/nilai) untuk *bypass* autentikasi.
- **Pencatatan Data dan Observasi:** Mencatat secara sistematis hasil dari setiap langkah analisis, termasuk: (a) keberhasilan atau kegagalan menemukan *string* kritis, (b) jumlah instruksi atau fungsi yang dapat diidentifikasi oleh *tool* analisis, (c) keberhasilan atau kegagalan upaya *bypass* mekanisme autentikasi (baik secara statis maupun dinamis), dan (d) observasi objektif lainnya mengenai perilaku *tool* analisis saat menghadapi kode yang tervirtualisasi dibandingkan dengan kode asli.



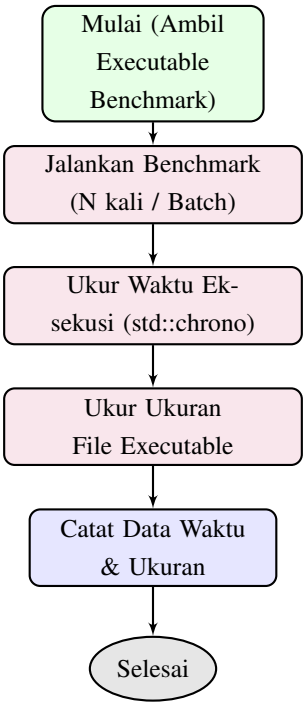
Gambar 3.2: Diagram Alur Prosedur Pengujian Keamanan Autentikasi.

Prosedur ini akan diulang untuk semua varian aplikasi autentikasi (Konsol, Qt, ImGui dalam versi *hardcoded* dan *cloud*), baik untuk versi asli maupun versi *virtualized*.

3.5.4 Pengujian Performa Overhead

Pengujian ini bertujuan mengukur dampak kuantitatif VxLang pada kinerja eksekusi dan ukuran file. Prosedur yang akan diikuti untuk setiap aplikasi *benchmark* (asli dan *virtualized*) diilustrasikan pada Gambar 3.3 dan melibatkan:

- **Pengukuran Waktu Eksekusi:** Menjalankan *benchmark* (QuickSort dan AES) berulang kali (N=100 untuk QuickSort, pemrosesan *batch* 1GB untuk AES) dan mencatat waktu eksekusi menggunakan `std::chrono::high_resolution_clock`.
- **Pengukuran Ukuran File:** Mengukur ukuran *byte* dari file *executable* akhir menggunakan `std::filesystem::file_size` atau utilitas OS.
- **Pencatatan Data:** Mencatat semua data waktu eksekusi (setiap *run* dan total/*average*) dan ukuran file untuk analisis selanjutnya.



Gambar 3.3: Diagram Alur Prosedur Pengujian Performa.

Prosedur ini akan diulang untuk semua aplikasi *benchmark* (QuickSort, Encryption, Size), baik untuk versi asli maupun versi *virtualized*.

3.5.5 Pengujian Studi Kasus Aplikasi Potensial Berbahaya dan Malware

Pengujian terhadap Lilith RAT dan sembilan sampel *malware/PUA* lainnya bertujuan untuk mengevaluasi dampak virtualisasi VxLang pada aplikasi yang lebih kompleks dari segi fungsionalitas (khusus untuk Lilith RAT), kesulitan analisis (khusus untuk Lilith RAT), dan deteksi otomatis oleh layanan pemindaian *malware*. Prosedur pengumpulan data mencakup:

1. **Persiapan Artefak Lilith RAT:** Meliputi kompilasi kode sumber klien Lilith versi asli dan versi *intermediate* yang telah ditandai makro VxLang, diikuti pemrosesan dengan *tool* VxLang untuk menghasilkan *executable* klien tervirtualisasi. Detail implementasi disajikan pada Bab 4 (Sub-bab 4.4).

2. **Persiapan Sampel Malware/PUA Tambahan:** Sembilan sampel *malware/PUA* lainnya (*Al-Khaser, donut, DripLoader, FilelessPELoader, JuicyPotato, ParadoxiaClient, PELoader, RunPE-In-Memory, SigLoader*) juga disiapkan dalam dua versi: versi asli (non-virtualized) dan versi yang telah diproses menggunakan *tool command-line* `vxlang.exe` untuk menghasilkan *executable* tervirtualisasi. Proses virtualisasi untuk sampel-sampel ini bertujuan untuk menerapkan proteksi VxLang

pada bagian-bagian kode yang dianggap relevan atau pada keseluruhan *executable* jika memungkinkan, mengikuti praktik serupa dengan yang diterapkan pada objek studi lainnya.

3. **Pengujian Fungsionalitas (Spesifik Lilith RAT):** Menjalankan klien Lilith tervirtualisasi dan server Lilith asli pada dua mesin terpisah dalam satu jaringan lokal. Skenario pengujian meliputi verifikasi koneksi, kemampuan menerima dan mengeksekusi perintah dari server (misalnya, akses *command prompt* jarak jauh, enumerasi direktori, dan pembacaan isi berkas pada mesin klien). Observasi dicatat untuk memastikan fungsionalitas inti RAT tetap berjalan normal setelah virtualisasi.

4. **Analisis Deteksi VirusTotal:** Semua sepuluh sampel perangkat lunak (klien Lilith RAT dan sembilan sampel *malware/PUA* lainnya), baik dalam versi asli (*non-virtualized*) maupun versi yang telah tervirtualisasi oleh VxLang, akan diunggah ke layanan VirusTotal. Hasil pemindaian dari berbagai *engine* antivirus akan dicatat dan dibandingkan, meliputi jumlah total deteksi oleh *engine* VirusTotal.

3.6 Teknik Analisis Data

Data yang telah dikumpulkan akan dianalisis menggunakan teknik berikut:

- **Analisis Keamanan (Efektivitas Proteksi):** Data hasil pengujian keamanan, seperti tingkat keberhasilan *bypass* autentikasi, kemampuan *tool* analisis dalam mengurai kode (misalnya, persentase *string* kritis yang berhasil disembunyikan, jumlah fungsi yang tidak teridentifikasi pada kode tervirtualisasi dibandingkan kode asli), dan metrik kuantitatif lain yang relevan, akan dianalisis secara deskriptif dan komparatif. Observasi sistematis terhadap perilaku *tool* (misalnya, kegagalan *disassembly*, ketidakmampuan menemukan referensi silang) akan digunakan untuk memperkuat analisis kuantitatif dalam menilai efektivitas proteksi VxLang antara kelompok kontrol dan eksperimen.

- **Performa:** Perhitungan statistik deskriptif (rata-rata, standar deviasi), perhitungan *overhead* waktu eksekusi (persentase), dan perhitungan peningkatan ukuran file. Data akan disajikan dalam tabel dan grafik perbandingan.

- **Analisis Trade-off:** Sintesis hasil analisis keamanan dan performa untuk mengevaluasi keseimbangan antara peningkatan proteksi dan dampak pada kinerja.

Hasil analisis ini akan menjadi dasar penarikan kesimpulan.

BAB 4
IMPLEMENTASI

Bab ini menyajikan penjabaran mendalam mengenai realisasi teknis dari metodologi penelitian yang telah diuraikan pada Bab 3. Fokus utama adalah pada langkah-langkah konkret yang diambil untuk membangun lingkungan eksperimen, mengembangkan artefak perangkat lunak yang diuji, dan mengaplikasikan teknik *code virtualization* menggunakan platform VxLang. Pembahasan mencakup detail penyiapan lingkungan, arsitektur aplikasi studi kasus, implementasi *benchmark* performa, dan proses integrasi VxLang itu sendiri.

4.1 Penyiapan Lingkungan Pengembangan

Fondasi dari penelitian eksperimental ini adalah lingkungan pengembangan yang konsisten dan terdefinisi dengan baik. Hal ini krusial untuk memastikan reliabilitas dan reproduktifitas hasil. Komponen-komponen kunci yang dikonfigurasi adalah sebagai berikut:

- **Sistem Operasi:** Seluruh proses pengembangan, kompilasi, dan eksekusi pengujian dilakukan pada **Microsoft Windows 11 (64-bit)**. Pemilihan Windows sebagai platform utama didasarkan pada target *executable* PE (Portable Executable) yang umum didukung oleh *tool* proteksi perangkat lunak seperti VxLang.

- **Compiler: Clang** (versi 19.1.3), diakses melalui antarmuka **`clang-cl`**, dipilih sebagai *compiler* C++. Penggunaan `clang-cl` memastikan kompatibilitas biner dengan toolchain Microsoft Visual C++ (MSVC), yang seringkali menjadi prasyarat atau lingkungan yang didukung optimal oleh VxLang, sambil tetap memanfaatkan kemampuan analisis dan optimasi modern dari Clang. Proyek ini dikompilasi dengan standar bahasa **C++17**.

- **Build System & Generator: CMake** (versi 3.31) digunakan sebagai *meta-build system* untuk mengelola kompleksitas *build* lintas target dan dependensi. File `CMakeLists.txt` mendefinisikan target-target *build*, dependensi, dan opsi kompilasi. **Ninja** (versi 1.12.1) digunakan sebagai *build generator* di bawah CMake untuk mempercepat proses kompilasi paralel. Konfigurasi `CMAKE_EXPORT_COMPILE_COMMANDS` diaktifkan untuk memfasilitasi integrasi dengan *tool* analisis kode.

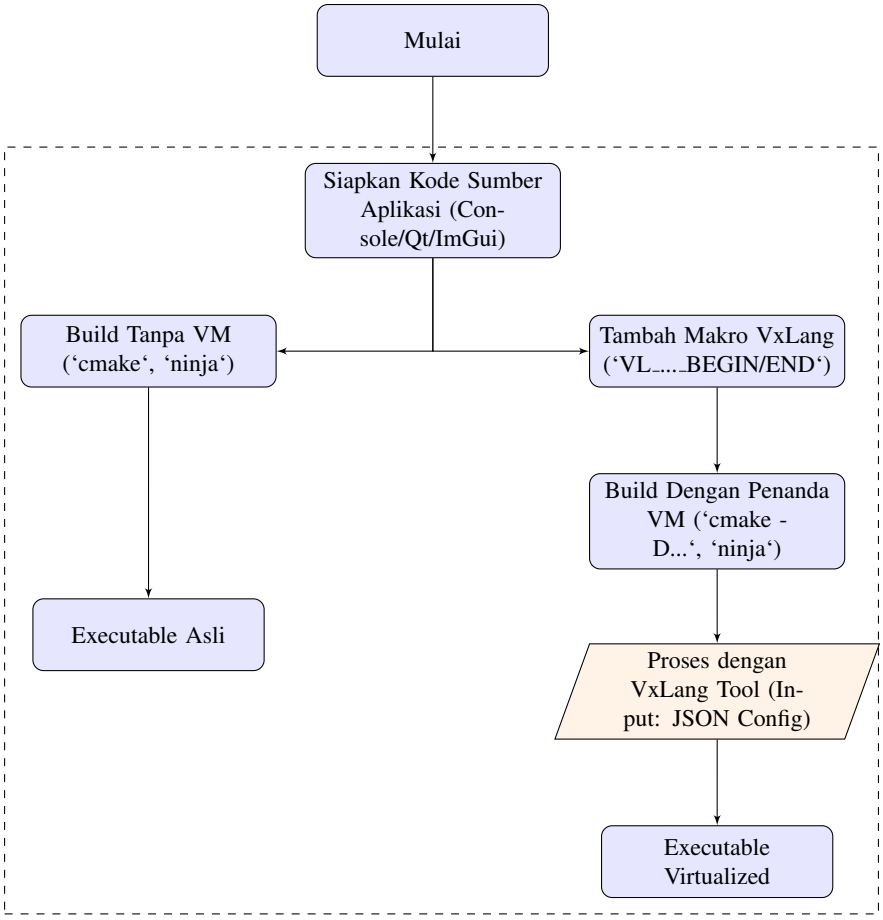
- **Integrated Development Environment (IDE):** Neovim dengan LSP Clangd digunakan untuk efisiensi dalam penulisan kode, navigasi, dan *debugging* awal.
- **Manajemen Dependensi & Library Pihak Ketiga:** Library eksternal dikelola secara manual dengan menempatkan *header* di direktori `includes` dan `deps`, serta file library (`.lib`) di direktori `lib`. Library utama yang digunakan meliputi:
  - **VxLang SDK:** Terdiri dari file *header* (`includes/vxlang/vxlib.h`) yang mendefinisikan makro penanda virtualisasi (`VL_VIRTUALIZATION_BEGIN`, `VL_VIRTUALIZATION_END`) dan library statis (`lib/vxlib64.lib`) yang diperlukan oleh proses virtualisasi.
  - **Qt Framework:** Versi **6.*x*** (MSVC 2022 64-bit) diintegrasikan menggunakan `find_package(Qt6)` CMake. Modul `Widgets` digunakan untuk komponen GUI.
  - **Dear ImGui:** Library Dear ImGui beserta *backend* **GLFW** dan **OpenGL3** di-*compile* sebagai bagian dari proyek (`deps/*.cpp`).
  - **libcurl:** Digunakan untuk komunikasi HTTP pada aplikasi autentikasi varian *cloud*.
  - **OpenSSL:** Versi **3.*x*** (`libssl`, `libcrypto`) digunakan untuk implementasi enkripsi AES pada *benchmark* performa.
  - **nlohmann/json:** Library C++ *header-only* untuk menangani data JSON pada varian *cloud*.

4.2 Implementasi Pengujian Autentikasi

Bagian ini berfokus pada pengembangan aplikasi yang mensimulasikan proses login pengguna, yang kemudian menjadi subjek analisis *reverse engineering* sebelum dan sesudah penerapan VxLang. Pendekatan analisis mencakup analisis statis menggunakan Ghidra dan analisis dinamis menggunakan x64dbg, keduanya bertujuan mengidentifikasi dan mencoba mem-*bypass* logika autentikasi.

Diagram alur persiapan *executable* disajikan pada Gambar 4.1. Proses analisis statis dan dinamis, termasuk upaya *bypass*, diilustrasikan pada Gambar 4.2. Proses ini melibatkan kompilasi kode sumber menjadi dua versi: versi asli (non-virtualized) dan versi *intermediate* yang telah ditandai dengan makro VxLang dan ditautkan dengan pustaka VxLang (`vxlib64.lib`). Versi *intermediate* ini kemudian diproses lebih lanjut menggunakan *tool command-line* `vxlang.exe`. Konfigurasi lisensi vxlang ditangani melalui *file* `vxlang.vxm` yang berada satu direktori dengan *vxlang.exe*. Analisis dinamis dimulai

dengan langkah serupa analisis statis, yaitu mencari *string* atau pola kode yang relevan di dalam *debugger* (x64dbg) untuk membantu menemukan lokasi logika autentikasi. Setelah lokasi potensial ditemukan, *breakpoint* dipasang. Upaya *bypass* kemudian dilakukan baik secara statis (mem-*patch* file *executable* menggunakan Ghidra) maupun secara dinamis (mem-*patch* instruksi *jump* kondisional atau memanipulasi register/memori secara langsung di x64dbg saat program berjalan).

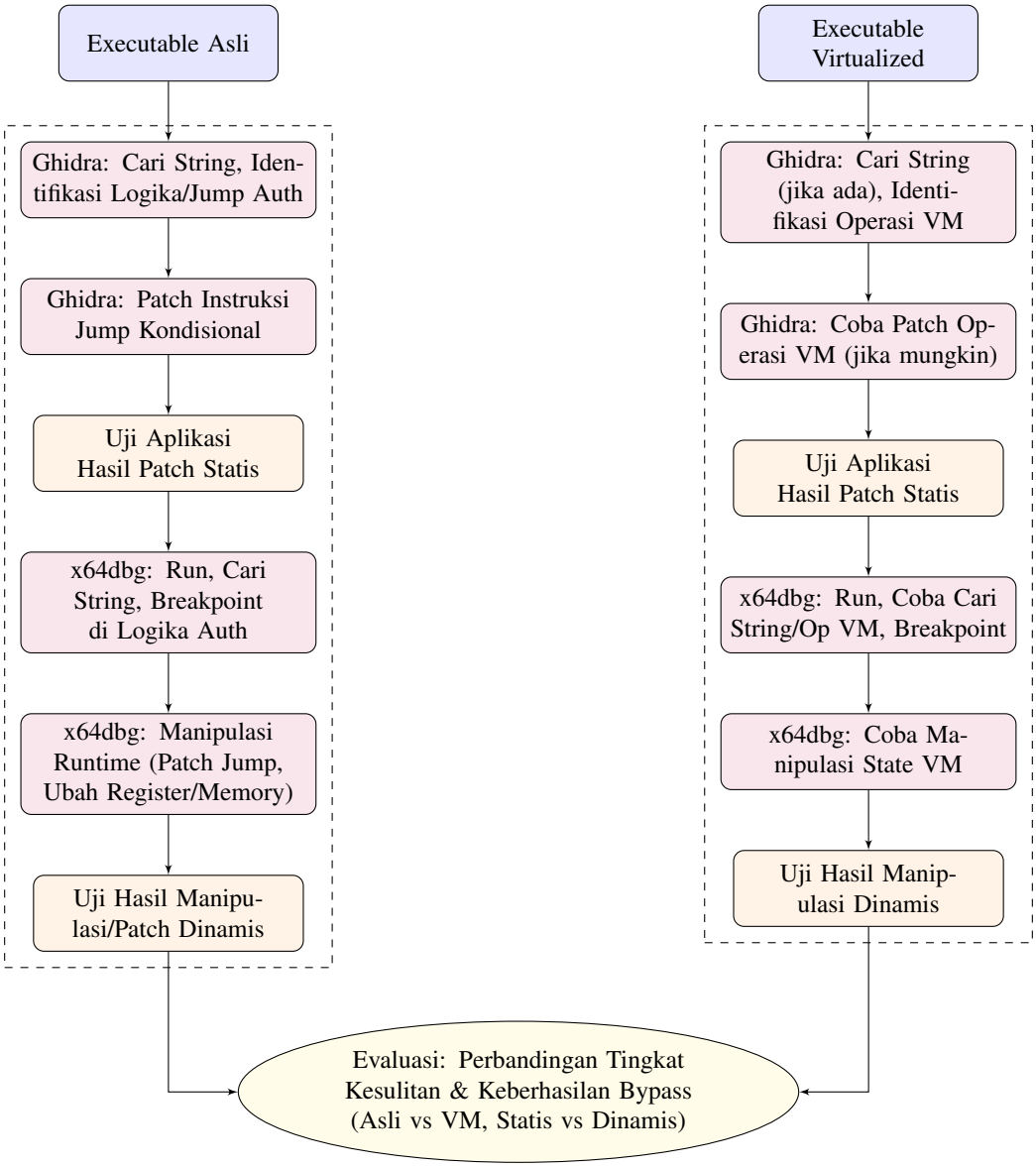


Gambar 4.1: Diagram Alur Persiapan Executable untuk Pengujian Autentikasi.

4.2.1 Aplikasi Studi Kasus Autentikasi

Tiga jenis aplikasi dengan dua varian mekanisme autentikasi dikembangkan:

1. **Aplikasi Konsol (`console`, `console_cloud`):** Aplikasi CLI sederhana.
    - **Varian Hardcoded (`src/console/console.cpp`):** Membandingkan input `std::cin` dengan *string* literal (*"seno"*, *"rahman"*) menggunakan `std::string::compare()`.

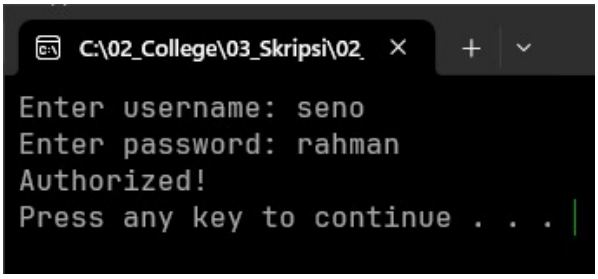


Gambar 4.2: Diagram Alur Analisis Upaya Bypass Autentikasi.

```
// ... setelah input username & password ...
if (inputUsername.compare("seno") == 0 &&
    inputPassword.compare("rahman") == 0) {
  std::cout << "Authorized!" << std::endl;
} else {
  std::cout << "Not authorized." << std::endl;
}
// ...
```

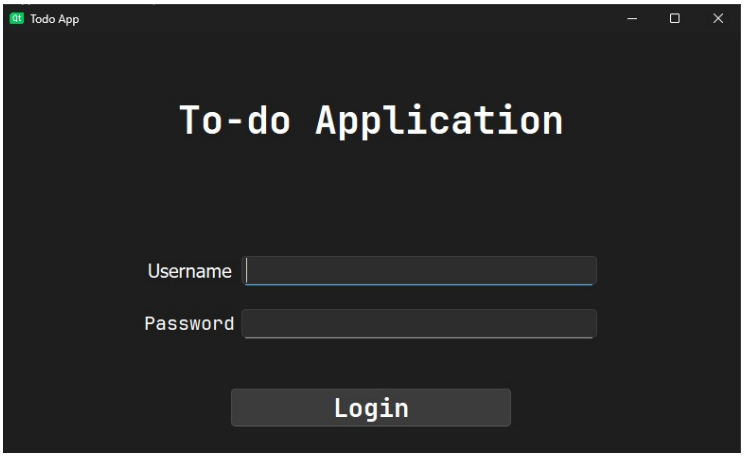
Kode 4.1: Logika Perbandingan Hardcoded (Konsol)

- **Varian Cloud (`src/console/console_cloud.cpp`):** Menggunakan `send_login_request` dari `includes/cloud.hpp` untuk mengirim kredensial via HTTP POST ke `http://localhost:9090/login`.



Gambar 4.3: Tampilan Aplikasi Autentikasi Varian Konsol saat dijalankan.

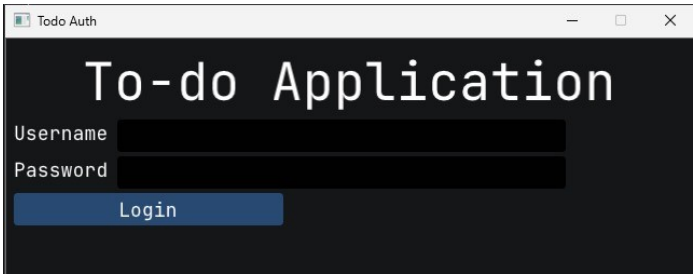
2. **Aplikasi Qt (`app_qt`, `app_qt_cloud`):** Aplikasi GUI menggunakan Qt Widgets. UI dari `src/app_qt/forms/todo_auth.ui`.
   - **Varian Hardcoded (`src/app_qt/src/todo_auth.cpp`):** Logika serupa Kode 4.1 ditempatkan dalam *slot* `on_button_login_clicked()`, menggunakan `ui->input_User->text()` dan `ui->input_Pass->text()`. Hasil via `QMessageBox`.
   - **Varian Cloud (`src/app_qt/src/todo_auth_cloud.cpp`):** Slot memanggil `send_login_request`.



Gambar 4.4: Tampilan Antarmuka Aplikasi Autentikasi Varian Qt.

3. **Aplikasi Dear ImGui (`app_imgui`, `app_imgui_cloud`):** Aplikasi GUI *immediate mode* menggunakan Dear ImGui, GLFW, OpenGL.

    - **Varian Hardcoded (`src/app_imgui/login.cpp`):** Logika perbandingan dalam `Login::LoginWindow` saat `ImGui::Button("Login")` ditekan. Hasil ditampilkan via `MessageBoxW`.
    - **Varian Cloud (`src/app_imgui/login_cloud.cpp`):** Memanggil `send_login_request` saat tombol login ditekan, menampilkan hasil via `MessageBoxW`.



Gambar 4.5: Tampilan Antarmuka Aplikasi Autentikasi Varian Dear ImGui.

Implementasi Fungsi Permintaan Cloud (`cloud.hpp`): Fungsi `send_login_request` bertanggung jawab untuk mengemas kredensial ke dalam format JSON dan mengirimkannya ke server *backend* menggunakan library `libcurl`.

4.2.2 Implementasi Sisi Server (Varian Cloud)

Backend API lokal untuk varian *cloud* diimplementasikan sebagai berikut:

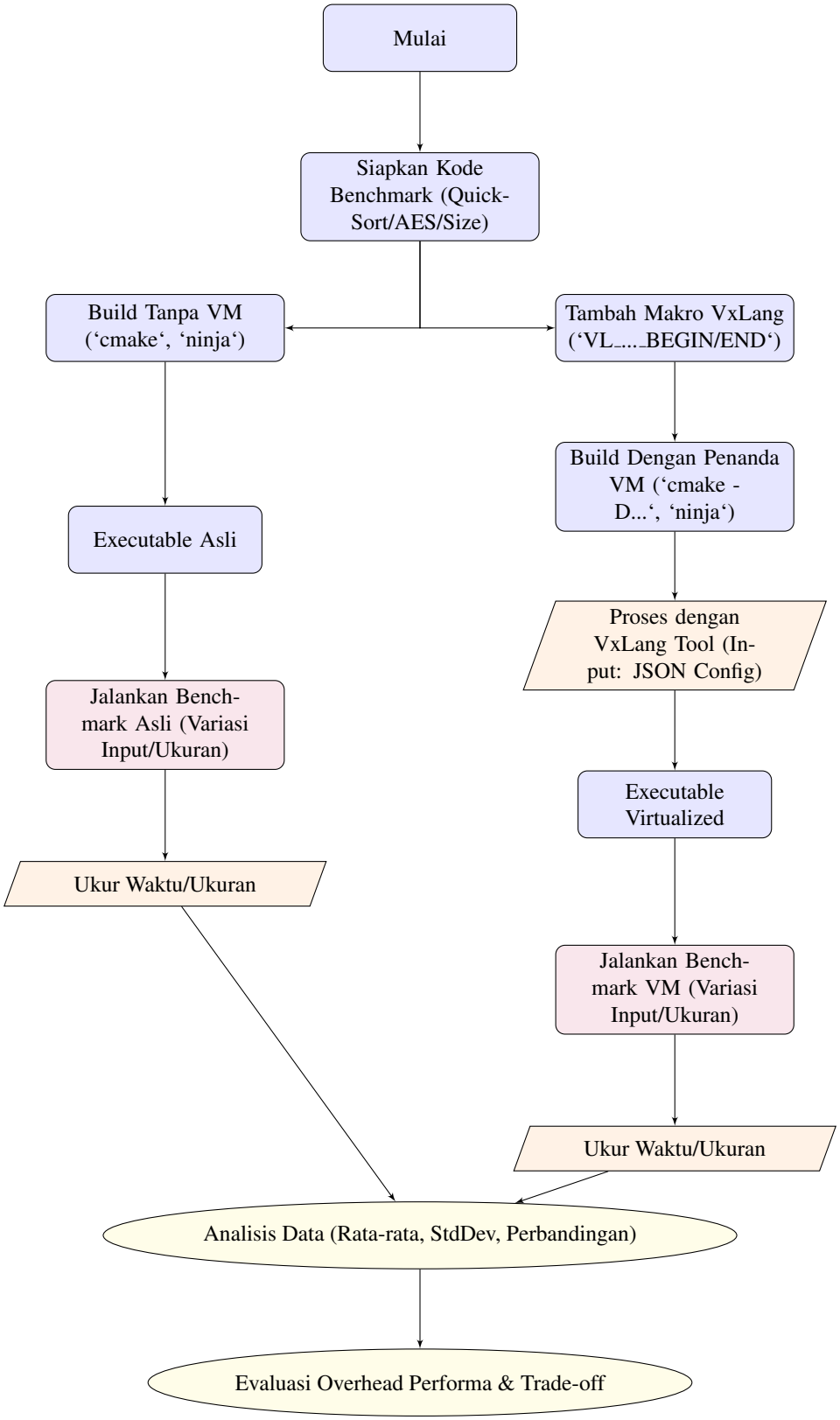
- **Teknologi:** Go (Golang) untuk API (`server/backend/main.go`), PostgreSQL 15 untuk database, Docker dan Docker Compose untuk *deployment* (`server/docker-compose.yml`).
- **API Endpoint `/login` (POST):** Menerima JSON, mengambil *salt/hash* dari DB, menghitung ulang *hash* input menggunakan PBKDF2-SHA256 (100k iterasi), membandingkan *hash* (*constant time*), mengembalikan `"success": true/false`.
- **API Endpoint `/register` (POST):** Menerima JSON, generate *salt*, hitung *hash* PBKDF2, simpan ke DB.
- **Database Schema (`server/postgres/init.sql`):** Tabel `users(username, password_hash, salt)`. Menyisipkan user *default* (*seno*/rahman) dengan *hash/salt* precomputed.

4.3 Implementasi Pengujian Performa

Diagram alur untuk persiapan dan pelaksanaan pengujian performa diilustrasikan pada Gambar 4.6. Serupa dengan persiapan *executable* untuk studi kasus autentikasi, artefak untuk *benchmark* performa juga disiapkan dalam dua versi: asli dan *intermediate* yang ditandai VxLang. *Executable intermediate* tersebut kemudian diproses oleh *tool command-line* `vxlang.exe` untuk menghasilkan versi tervirtualisasi akhir yang diuji.

4.3.1 Benchmark Algoritma Quick Sort (`QuickSort`)

- **Implementasi (`src/performance/quick_sort.cpp`):** Menggunakan `std::vector<int>`, fungsi rekursif `quickSort`, dan `partition`.
- **Data:** Vektor acak (1-1.000.000) ukuran bervariasi (100 s/d 3.000.000 elemen).
- **Pengukuran:** `std::chrono::high_resolution_clock` mengukur waktu eksekusi `quickSort`. Diulang 100 kali per ukuran data. Rata-rata dan standar deviasi dihitung.
- **Integrasi VxLang:** `VL_VIRTUALIZATION_BEGIN/END` membungkus *seluruh isi* fungsi rekursif `quickSort`.



Gambar 4.6: Diagram Alur Persiapan dan Pengujian Performa.

```cpp
// Potongan dari src/performance/quick_sort.cpp
std::pair<double, double> measureSortingTime(size_t size, int runs = 10)
{
  std::vector<double> times;
  for (int i = 0; i < runs; ++i) {
    std::vector<int> data = generateRandomVector(size);
    auto start = std::chrono::high_resolution_clock::now();
    quickSort(data, 0, data.size() - 1); // Fungsi yang divirtualisasi
    auto stop = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> duration = stop - start;
    times.push_back(duration.count());
  }
  // ... (Calculate average and stdDev) ...
  return {average, stdDev};
}
```

Kode 4.2: Pengukuran Waktu Eksekusi QuickSort

4.3.2 Benchmark Enkripsi AES-CBC-256 (`Encryption`)

- **Implementasi:** Kelas `AESCipher` (`aes.h`, `aes.cpp`) menggunakan API `EVP` OpenSSL untuk AES-256-CBC.
- **Data:** 1 GB data acak (1 juta blok @ 1024 byte), diproses per *batch* (misal, 10.000 blok/batch).
- **Pengukuran:** `chrono` mengukur total waktu enkripsi/dekripsi seluruh *batch*. *Throughput* (MB/s) dihitung.
- **Integrasi VxLang:** `VL_VIRTUALIZATION_BEGIN/END` membungkus *looping* pemanggilan `aes.encrypt()`/`decrypt()` di dalam fungsi `measureBatch...Time`.

4.3.3 Pengukuran Ukuran File

- **Implementasi:** Ukuran file *executable* diukur menggunakan fungsi `std::filesystem::file_size` atau melalui *file explorer* Windows. Aplikasi `Size` (`src/performance/size.cpp`) dibuat khusus dengan data *dummy* tersemat (`dummy.bin` via `dummy.rc`) untuk mensimulasikan aplikasi dengan aset data internal yang besar.
- **Target Pengukuran:** Pengukuran ukuran file dilakukan pada **semua** target *executable* yang dihasilkan, baik versi asli maupun versi *virtualized* (`*_vm.exe`), termasuk `QuickSort`, `Encryption`, `Size`, `console`, `app_imgui`, dan `app_qt`, sesuai data pada Tabel 5.7.

- **Integrasi VxLang pada Target `Size`:** Makro `VL_VIRTUALIZATION_BEGIN/END` tetap disertakan dalam `main` pada target `Size` untuk memastikan *runtime* VxLang disertakan pada versi `size_vm.exe`, sehingga memungkinkan perbandingan ukuran yang fokus pada *overhead runtime* itu sendiri.

4.3.4 Integrasi dan Proses Virtualisasi VxLang

Penerapan VxLang **dilakukan** dengan langkah-langkah berikut:

1. **Penempatan Makro:** Makro `VL_VIRTUALIZATION_BEGIN/END` **ditempatkan** membungkus logika autentikasi inti, fungsi rekursif `quickSort`, loop enkripsi/dekripsi, dan blok dummy di `Size`. Contoh penempatan pada aplikasi konsol hardcoded:

```
// ... (Input username/password) ...
VL_VIRTUALIZATION_BEGIN; // <-- Makro Awal

if (inputUsername.compare("seno") == 0 &&
    inputPassword.compare("rahman") == 0) {
  // ... (Authorized code) ...
} else {
  // ... (Not authorized code) ...
}

VL_VIRTUALIZATION_END; // <-- Makro Akhir
// ...
```

Kode 4.3: Penempatan Makro VxLang pada Logika Autentikasi

2. **Konfigurasi Build CMake:** File `CMakeLists.txt` memegang peranan sentral dalam mengelola proses kompilasi untuk menghasilkan kedua jenis *executable*: asli (non-virtualized) dan *intermediate* (`*_vm.exe`) yang siap diproses oleh VxLang. Konfigurasi ini mencakup:

   - Definisi *preprocessor* `USE_VL_MACRO`: Opsi ini, biasanya dikontrol melalui *flag* CMake (misalnya, `-DUSE_VL_MACRO=1`), digunakan untuk mengaktifkan makro-makro VxLang (`VL_VIRTUALIZATION_BEGIN`, dll.) yang didefinisikan dalam `includes/vxlang/vxlib.h`. Jika `USE_VL_MACRO` tidak didefinisikan, makro-makro tersebut menjadi kosong sehingga tidak ada kode VxLang yang aktif.
   - Penautan Pustaka VxLang: Untuk *build intermediate* (`*_vm.exe`), pustaka statis VxLang (`vxlib64.lib`) perlu ditautkan. Pustaka ini berisi fungsi-fungsi yang dipanggil oleh makro VxLang saat `USE_VL_MACRO` aktif.

- Penamaan Output: *Build system* dikonfigurasi untuk menghasilkan nama *file output* yang berbeda untuk versi asli dan versi *intermediate*, misalnya `nama_target.exe` untuk versi asli dan `nama_target_vm.exe` untuk versi *intermediate*.

Sebagai ilustrasi, berikut adalah contoh penyederhanaan logika yang dapat ditemukan dalam `CMakeLists.txt` (fungsi `add_project_executables` pada proyek ini mengimplementasikan logika serupa secara lebih komprehensif):

```
# Contoh logika CMake untuk target 'console'
# (Versi sederhana untuk ilustrasi)

# Target Asli (Non-VM)
add_executable(console src/console/console.cpp ...)
set_target_properties(console PROPERTIES OUTPUT_NAME "console")
# ... (tidak ada USE_VL_MACRO, tidak ada link ke vxlib64)

# Target Intermediate (untuk VxLang)
add_executable(console_vm_cmake_example src/console/console.cpp ...)
target_compile_definitions(console_vm_cmake_example PRIVATE USE_VL_MACRO)
target_link_libraries(console_vm_cmake_example PRIVATE vxlib64)
set_target_properties(console_vm_cmake_example PROPERTIES OUTPUT_NAME
"console_vm")
```

Kode 4.4: Contoh Ilustratif Konfigurasi CMake untuk Build Asli dan Intermediate VxLang

3. **Proses Kompilasi dan Virtualisasi Aktual:** Proses keseluruhan untuk menghasilkan *executable* tervirtualisasi melibatkan langkah-langkah berikut, yang diorkestrasi oleh kombinasi CMake, Ninja, dan skrip `virtualize.bat`:

   - **Build Asli:** Jalankan perintah CMake (misalnya, `cmake -G Ninja -B build_asli ..`) diikuti dengan perintah Ninja (`ninja -C build_asli`). Proses ini akan menghasilkan *executable* asli tanpa modifikasi VxLang di direktori *output* yang ditentukan (misalnya, `build_asli/bin/`).
   - **Build Intermediate (`*_vm.exe`):** Jalankan perintah CMake dengan *flag* untuk mengaktifkan VxLang (misalnya, `cmake -G Ninja -B build_vm -DUSE_VL_MACRO=1 ..`) diikuti dengan perintah Ninja (`ninja -C build_vm`). Proses ini menghasilkan *executable intermediate* (misalnya, `console_vm.exe`) di direktori *output* yang sesuai. *Executable* ini sudah mengandung penanda VxLang dari makro dan tertaut dengan `vxlib64.lib`.
   - **Proses Virtualisasi VxLang (`*_vxm.exe`):** *Executable intermediate* (`*_vm.exe`) kemudian diproses menggunakan *tool command-line*

vxlang.exe. Skrip virtualize.bat dalam proyek ini mengotomatiskan langkah ini untuk semua target yang relevan. Contoh baris perintah yang mungkin dijalankan oleh skrip ini untuk target console_vm.exe adalah:

```
REM Contoh dari virtualize.bat (disesuaikan untuk kejelasan)
set "vxlang_tool_path=vxlang\vxlang.exe"
set "intermediate_exe_path=bin\console\Release\console_vm.exe"

REM Perintah aktual untuk menjalankan virtualisasi
%vxlang_tool_path% %intermediate_exe_path%
```

Kode 4.5: Contoh Baris Perintah dari Skrip untuk Menjalankan VxLang Tool

Pemanggilan vxlang.exe pada console_vm.exe akan menganalisis penanda virtualisasi, mengganti kode *native* yang ditandai dengan representasi *byte-code* internal, dan menyematkan *runtime Virtual Machine* (VM) yang diperlukan. Proses ini secara otomatis menghasilkan *file executable* akhir yang telah tervirtualisasi dengan nama yang dimodifikasi, yaitu console_vm.vxm.exe, yang disimpan di direktori yang sama dengan *file input*-nya. *Executable* *_vxm.exe inilah yang kemudian menjadi subjek utama pengujian keamanan dan analisis dampak performa dalam penelitian ini.

4.4 Studi Kasus: Aplikasi Potensial Berbahaya dan Malware

Selain aplikasi studi kasus autentikasi dan *benchmark* performa yang dikembangkan secara khusus, penelitian ini juga melibatkan analisis terhadap aplikasi *Remote Administration Tool* (RAT) *open-source* bernama Lilith. Penggunaan Lilith bertujuan untuk menguji efektivitas dan dampak VxLang pada perangkat lunak yang lebih kompleks dan memiliki karakteristik yang sering diasosiasikan dengan *malware* atau *Potentially Unwanted Application* (PUA).

4.4.1 Deskripsi Singkat Lilith RAT

Lilith adalah sebuah RAT yang dikembangkan menggunakan C++ dan dirancang untuk platform Windows. Kode sumbernya tersedia secara publik [36], memungkinkan untuk kompilasi dan modifikasi. Fungsionalitas utama yang ditawarkan oleh Lilith mencakup kemampuan untuk:

- Membuat koneksi terbalik (*reverse connection*) dari klien ke server.
- Mengeksekusi perintah pada sistem klien secara jarak jauh melalui antarmuka *command-line*.

- Melakukan *keylogging* untuk merekam ketikan papan ketik pada mesin klien.
- Menyediakan mekanisme persistensi sederhana.
- Mentransfer berkas antara server dan klien (fungsionalitas ini ada dalam kode sumbernya, meskipun mungkin tidak menjadi fokus utama pengujian virtualisasi dalam penelitian ini).

Karena fungsionalitas tersebut, Lilith RAT menjadi objek studi yang menarik untuk melihat bagaimana virtualisasi kode dapat mempersulit analisis terhadap perangkat lunak dengan potensi penggunaan ganda (*dual-use*).

4.4.2 Persiapan *Executable* Lilith untuk Analisis

Proses persiapan *executable* klien Lilith untuk analisis virtualisasi mengikuti langkah-langkah serupa dengan aplikasi studi kasus lainnya, dengan penekanan pada identifikasi dan penandaan bagian kode yang kritikal.

1. **Akuisisi dan Kompilasi Kode Sumber Asli:** Kode sumber Lilith RAT (khususnya komponen klien) diunduh dari repositori publiknya. Kompilasi dilakukan menggunakan lingkungan pengembangan yang telah disiapkan (Clang/clang-cl, CMake, Ninja dengan standar C++17) untuk menghasilkan *executable* klien Lilith versi asli (`Lilith_Client.exe`). Komponen server Lilith juga dikompilasi dalam versi aslinya dan digunakan untuk menguji fungsionalitas klien.
2. **Identifikasi dan Penandaan Kode Kritis dengan Makro VxLang:** Mengingat kompleksitas Lilith, virtualisasi tidak diterapkan pada keseluruhan kode sumber klien. Sebaliknya, fokus diberikan pada fungsi-fungsi dan blok-blok kode yang dianggap paling krusial untuk operasional RAT dan yang paling mungkin menjadi target analisis oleh *reverse engineer*. Makro `VL_VIRTUALIZATION_BEGIN` dan `VL_VIRTUALIZATION_END` dari VxLang SDK disisipkan secara strategis untuk membungkus bagian-bagian berikut dalam kode sumber klien Lilith (`src/Lilith/client/`):
    - **Pemrosesan Perintah Inti:** Dalam fungsi seperti `Client::ProcessPacketType` (pada berkas `client.cpp`), logika yang menangani paket instruksi dari server dan memanggil fungsi `General::processCommand` divirtualisasi. Bagian yang memproses perintah CMD jarak jauh melalui objek `CMD::cmdptr` juga ditandai. Contoh penempatan makro secara ilustratif dapat dilihat pada Lampiran 11 (Kode 12).

- **Inisialisasi dan Koneksi:** Bagian kode dalam konstruktor `Client::Client` atau fungsi `Client::Connect` yang bertanggung jawab untuk inisialisasi koneksi jaringan ke server juga dipertimbangkan untuk virtualisasi untuk mengaburkan detail mekanisme koneksi.
- **Loop Utama Pemrosesan Paket:** Loop utama dalam `Client::ClientThread` yang terus menerus menerima dan memproses paket dari server menjadi target virtualisasi untuk mempersulit pemahaman alur kerja komunikasi.
- **Fungsi Sensitif Tambahan:** Beberapa fungsi pendukung dalam `general.cpp` atau `cmdRedirect.cpp` yang terkait langsung dengan eksekusi perintah atau interaksi sistem juga ditandai, jika relevan.

Penempatan makro dilakukan dengan hati-hati untuk memastikan fungsionalitas utama tetap terjaga sambil memaksimalkan kesulitan analisis.

3. **Build *Executable Intermediate* (`Lilith_Client_vm.exe`):** Setelah penandaan kode, klien Lilith dikompilasi ulang dengan mengaktifkan *flag preprocessor* `USE_VL_MACRO` melalui konfigurasi CMake. *Executable* yang dihasilkan (`Lilith_Client_vm.exe`) secara otomatis tertaut dengan pustaka VxLang (`vxlib64.lib`).

4. **Proses Virtualisasi Akhir dengan VxLang Tool:** *Executable intermediate* `Lilith_Client_vm.exe` kemudian diproses menggunakan *tool command-line* `vxlang.exe`. Proses ini diotomatisasi melalui skrip `virtualize.bat`, yang menjalankan perintah serupa dengan:

```
REM Contoh dari virtualize.bat untuk Lilith Client
set "vxlang_tool_path=vxlang\vxlang.exe"
set
"lilith_intermediate_path=bin\Lilith_Client\Release\Lilith_Client_vm.exe"
%vxlang_tool_path% %lilith_intermediate_path%
```

Kode 4.6: Perintah Virtualisasi untuk Klien Lilith RAT

Perintah ini menghasilkan *executable* klien Lilith yang telah tervirtualisasi sepenuhnya, yaitu `Lilith_Client_vm.vxm.exe`. Berkas inilah yang kemudian digunakan untuk pengujian fungsionalitas, analisis keamanan statis/dinamis, dan pengujian deteksi VirusTotal yang hasilnya dipaparkan pada Bab 5.3.

Persiapan yang cermat ini memastikan bahwa kedua versi klien Lilith (asli dan tervirtualisasi) dapat dibandingkan secara adil untuk mengevaluasi dampak VxLang.

4.4.3 Tantangan dalam Penempatan Makro VxLang pada Lilith RAT

Proses penerapan makro virtualisasi VxLang pada kode sumber klien Lilith RAT mengungkapkan sebuah aspek krusial, yaitu sensitivitas fungsionalitas program terhadap penempatan makro yang kurang tepat. Meskipun tujuan utama virtualisasi adalah untuk melindungi bagian kode yang paling kritikal, pengalaman dalam studi kasus ini menunjukkan bahwa tidak semua blok kode dapat divirtualisasi secara langsung tanpa menimbulkan efek samping yang merugikan.

Sebagai contoh, upaya awal untuk memvirtualisasi blok kode yang lebih besar atau yang memiliki interaksi kompleks dengan alur kontrol dan pemanggilan fungsi eksternal pada Lilith RAT menemui kendala. Salah satu observasi spesifik terjadi pada penanganan paket `PacketType::CMDCommand` dalam fungsi `Client::ProcessPacketType`. Ketika seluruh blok `case` untuk `CMDCommand` dibungkus dengan makro `VL_VIRTUALIZATION_BEGIN` dan `VL_VIRTUALIZATION_END`, seperti pada ilustrasi berikut:

```
// Ilustrasi penempatan makro yang awalnya menyebabkan masalah
case PacketType::CMDCommand: {
    VL_VIRTUALIZATION_BEGIN; //<- Awalnya di sini
    std::string msg;
    if (!GetString(msg)) // Fungsi untuk mendapatkan string dari
    jaringan
      return false;

    if (CMD::cmdptr != NULL) {
      CMD::cmdptr->writeCMD(msg); // Fungsi untuk menulis perintah ke
      CMD
    } else {
      SendString("Initiate a CMD session first.", PacketType::Warning);
    }
    VL_VIRTUALIZATION_END; //<- Awalnya di sini
    break;
}
```

Kode 4.7: Contoh Ilustratif Penempatan Awal Makro VxLang pada Lilith RAT yang Menimbulkan Masalah Fungsionalitas

Penempatan makro seperti pada Kode 4.7 menyebabkan aplikasi klien Lilith mengalami *crash* atau *hang* ketika server mengirimkan perintah yang berkaitan dengan `PacketType::CMDCommand`. Fungsionalitas baru dapat dipulihkan ketika virtualisasi diterapkan secara lebih granular, misalnya dengan memindahkan makro untuk hanya membungkus pemanggilan fungsi spesifik di dalam logika tersebut, seperti di dalam fungsi `CMD::cmdptr->writeCMD(msg)` itu sendiri, bukan seluruh blok `case`.

Kendala serupa juga ditemui pada bagian kode yang bertanggung jawab untuk mengirimkan respons perintah kembali ke server:

```
// Ilustrasi penempatan makro lain yang menyebabkan masalah

VL_VIRTUALIZATION_BEGIN; //<- Penempatan ini menyebabkan hang/crash
SendString(General::processCommand(msg), PacketType::Instruction);
VL_VIRTUALIZATION_END;
```

Kode 4.8: Contoh Ilustratif Penempatan Makro VxLang Lain yang Menimbulkan Masalah

Ketika pemanggilan fungsi `SendString` (yang melibatkan operasi jaringan) bersama dengan `General::processCommand` divirtualisasi seperti pada Kode 4.8, klien Lilith akan mengalami *crash* dan terputus dari server saat server mencoba mengaktifkan fitur seperti `remoteControl cmd`.

Temuan ini mengindikasikan bahwa *tool* virtualisasi seperti VxLang mungkin memiliki interaksi yang kompleks dengan struktur kode tertentu, terutama yang melibatkan alur kontrol non-linear (seperti `switch-case`), operasi I/O, atau pemanggilan fungsi yang mengubah konteks eksekusi secara signifikan. Oleh karena itu, keberhasilan implementasi VxLang tidak hanya bergantung pada identifikasi bagian kode yang sensitif, tetapi juga pada eksperimen dan pengujian fungsional yang cermat untuk memastikan bahwa penempatan makro virtualisasi tidak merusak integritas operasional aplikasi. Hal ini menekankan pentingnya pendekatan virtualisasi yang selektif dan teruji.

4.4.4 Persiapan Sampel Malware Tambahan untuk Analisis VirusTotal

Selain Lilith RAT, penelitian ini melibatkan analisis dampak virtualisasi VxLang terhadap deteksi oleh VirusTotal untuk sembilan sampel *malware* atau *Potentially Unwanted Application* (PUA) tambahan. Sampel-sampel ini, yang diperoleh dari berbagai sumber repositori publik atau koleksi sampel analisis, adalah: *Al-Khaser, donut, DripLoader, FilelessPELoader, JuicyPotato, ParadoxiaClient, PELoader, RunPE-In-Memory,* dan *SigLoader*.

Untuk setiap sampel *malware/PUA* tersebut, dua versi *executable* disiapkan untuk perbandingan di VirusTotal:

1. **Versi Asli (Non-Virtualized):** Ini adalah berkas *executable* asli dari sampel *malware/PUA* yang bersangkutan, tanpa modifikasi atau pemrosesan lebih lanjut oleh VxLang.

2. **Versi Virtualized (VM):** Berkas *executable* asli dari setiap sampel *malware/PUA* kemudian diproses menggunakan *tool command-line* `vxlang.exe`. Mengingat sifat *closed-source* dari sebagian besar sampel ini dan tidak adanya kode sumber untuk penempatan makro SDK secara manual, proses virtualisasi dilakukan dengan mengarahkan *tool* VxLang untuk memproses *executable* secara keseluruhan. *Tool* VxLang akan mencoba mengidentifikasi dan memvirtualisasi bagian-bagian kode yang dapat diprosesnya. Hasil dari proses ini adalah *executable* baru (biasanya dengan sufiks `_vxm.exe`) yang telah tervirtualisasi.

Penting untuk dicatat bahwa efektivitas dan kelengkapan proses virtualisasi oleh *tool* VxLang pada *executable* biner tanpa penandaan makro manual dapat bervariasi tergantung pada struktur internal dan teknik proteksi (jika ada) yang sudah ada pada sampel *malware* asli. Namun, pendekatan ini mensimulasikan skenario di mana pengguna VxLang mungkin mencoba melindungi *executable* pihak ketiga atau *legacy code* di mana modifikasi kode sumber tidak memungkinkan.

Kedua versi (*non-virtualized* dan *virtualized*) dari masing-masing sepuluh sampel perangkat lunak (Lilith RAT dan sembilan sampel tambahan) kemudian diunggah ke VirusTotal untuk analisis perbandingan tingkat deteksi.

BAB 5
HASIL PENELITIAN

5.1 Analisis Pengujian Autentikasi VxLang

5.1.1 Analisis Statis

Analisis statis dilakukan untuk memahami mekanisme autentikasi aplikasi tanpa menjalankan kode. Alat yang digunakan dalam analisis ini adalah Ghidra, sebuah *software reverse engineering* yang menyediakan kemampuan *disassembly* dan *decompilation*. Tujuan utama dari analisis statis ini adalah untuk mengidentifikasi lokasi kode yang menangani proses autentikasi dan mencari potensi celah keamanan yang dapat dimanfaatkan untuk melewati proses tersebut.

5.1.1.1 Analisis Aplikasi Non-Virtualized

Pada aplikasi non-virtualized (konsol, Qt, dan ImGUI), proses analisis dimulai dengan mencari *string* yang relevan dengan autentikasi, seperti ”Authentication Failed” atau *string* yang mungkin digunakan sebagai *username* atau *password*. Setelah *string* tersebut ditemukan, langkah selanjutnya adalah menelusuri referensi silang (*cross-references*) untuk melihat di mana *string* tersebut digunakan dalam kode program. Hal ini membantu dalam mengidentifikasi fungsi atau blok kode yang bertanggung jawab untuk menampilkan pesan kegagalan autentikasi, yang sering kali berdekatan dengan logika autentikasi itu sendiri.

Sebagai contoh, pada aplikasi *app_imgui*, analisis *disassembly* menggunakan Ghidra mengungkapkan potongan kode berikut:

```
; (Kode memuat input password ke register, misal RDI) ...

; (Kode memuat password hardcoded "rahman", misal ke alamat yg
ditunjuk RDX)

LEA             RDX,[s_rahman_140110551]    = "rahman"
MOV             RCX,RDI
CALL            VCRUNTIME140.DLL::memcmp ; Bandingkan input & hardcoded
pwd
CMP             RBX,0x6
TEST            EAX,EAX                  ; Cek hasil memcmp (EAX=0 jika
sama)
JNZ             LAB_140003226            ; Lompat ke blok gagal jika
EAX != 0

; (Blok kode jika autentikasi berhasil)

LAB_140003226:  ; Label untuk blok gagal
LEA             RDX,[u_Authentication_Failed_1401104da] ; Load string
"Auth Failed"
CALL            qword ptr [->USER32.DLL::MessageBoxW] ; Tampilkan pesan
gagal
```

Kode 5.1: Snippet Assembly: Perbandingan Password dan Lompatan Kondisional (Non-Virtualized)

Dalam konteks kode di atas, pemanggilan `memcmp` (di operasi `LEA`) membandingkan password yang dimasukkan pengguna dengan nilai ”rahman” yang *hardcoded* pada aplikasinya. Hasilnya diperiksa oleh `TEST EAX,EAX` (`140003216`). Jika password tidak sama, `EAX` tidak akan nol, dan instruksi `JNZ` (`140003218`, Jump if Not Zero) akan mengalihkan eksekusi ke `LAB_140003226`, di mana pesan ”Authentication Failed” ditampilkan.

Untuk memvalidasi potensi celah keamanan, instruksi ‘JNZ LAB_140003226‘ dapat diubah (*patched*) menjadi ‘JZ LAB_140003226‘ atau instruksi lain yang akan selalu mengarahkan program untuk melewati blok kode yang menampilkan pesan kegagalan autentikasi. Dalam kasus ini, mengubah ‘JNZ‘ menjadi ‘JZ‘ (Jump if Zero) akan menyebabkan lompatan terjadi hanya jika hasil perbandingan adalah nol (yang menandakan autentikasi berhasil), sehingga secara efektif membalikkan logika dan memungkinkan akses tanpa otorisasi.

Lebih lanjut, analisis pada bagian *defined data* (.rdata) juga mengungkapkan adanya *string* ‘”seno”‘ dan ‘”rahman”‘. Keberadaan *string-string* ini sangat mengindikasikan bahwa aplikasi menyimpan *username* dan *password* secara *hard-coded*. Praktik menyimpan kredensial secara langsung dalam kode program sangat berbahaya dari sudut pandang keamanan. String seperti ini mudah ditemukan melalui analisis statis, seperti yang telah

dilakukan, sehingga penyerang dapat dengan mudah memperoleh informasi *username* dan *password* tanpa perlu melakukan *reverse engineering* yang mendalam atau menjalankan aplikasi. Dalam kasus ini, *username* ‘”seno”‘ dan *password* ‘”rahman”‘ yang ditemukan dalam *defined data* dapat dieksploitasi untuk melewati mekanisme autentikasi.

```
                    s_##input_password_14011053b
14011053b 23 23 69      ds          "##input_password"
          6e 70 75
          74 5f 70 ...

                    DAT_14011054c
14011054c 73            ??          73h    s
14011054d 65            ??          65h    e
14011054e 6e            ??          6Eh    n
14011054f 6f            ??          6Fh    o
140110550 00            ??          00h

                    s_rahman_140110551
140110551 72 61 68      ds          "rahman"
          6d 61 6e 00

                    s_To-do_Application_140110558


140110558 54 6f 2d      ds          "To-do Application"
          64 6f 20
          41 70 70 ...
```

Gambar 5.1: *Hardcoded username & password* pada *defined data*

Analisis serupa juga dilakukan pada aplikasi konsol dan Qt, di mana pola perbandingan string dan penggunaan instruksi kondisional untuk mengontrol alur program berdasarkan hasil autentikasi juga ditemukan dan dapat dimanipulasi dengan cara yang serupa untuk melewati proses autentikasi.

5.1.1.2 Analisis Aplikasi Non-Virtualized (Versi Cloud)

Untuk mengatasi risiko *hard-coded* username dan password, mekanisme autentikasi telah diubah menjadi berbasis *cloud*. Dalam versi *cloud* ini, aplikasi mengirimkan username dan password dalam format JSON ke server HTTP, yang kemudian melakukan verifikasi terhadap database PostgreSQL. Dengan perubahan ini, username dan password tidak lagi disimpan secara langsung di dalam aplikasi klien.

Meskipun kredensial tidak lagi *hard-coded*, analisis statis tetap relevan untuk memahami bagaimana aplikasi klien berinteraksi dengan server. Sebagai contoh, pada aplikasi *console_cloud*, analisis *disassembly* menggunakan Ghidra pada fungsi yang menangani autentikasi menunjukkan potongan kode berikut:

```
; ... (Kode setelah fungsi send_login_request kembali, hasil
mungkin di AL) ...
MOV      AL, byte ptr [RBP + local_69] ; Ambil hasil boolean?
TEST     AL, 0x1                       ; Cek bit 0 dari AL
JNZ      LAB_14000175d                 ; Lompat jika hasil != 0
(Sukses?)
JMP      LAB_14000178e                 ; Lompat jika hasil == 0
(Gagal?)

LAB_14000175d:  ; Blok kode jika sukses
; ... (Kode menampilkan "Authorized!") ...
jmp      LAB_END_AUTH_CHECK                    ; Lompat ke akhir

LAB_14000178e:  ; Blok kode jika gagal
; ... (Kode menampilkan "Not authorized") ...
```

Kode 5.2: Snippet Assembly: Pemeriksaan Hasil Autentikasi Cloud (Non-Virtualized)

Pada kode di atas, instruksi `TEST AL, 0x1` kemungkinan memeriksa flag atau nilai boolean yang dikembalikan oleh fungsi autentikasi cloud (misalnya, 1 untuk sukses, 0 untuk gagal). Instruksi `JNZ LAB_14000175d` kemudian mengarahkan alur eksekusi ke blok ”Authorized!” jika hasil tes tidak nol (sukses), dan sebaliknya akan jatuh ke `JMP LAB_14000178e` atau langsung mengeksekusi blok ”Not authorized”. Sama seperti kasus sebelumnya, instruksi `JNZ` ini dapat dimanipulasi (misal, diubah menjadi `JMP LAB_14000175d` atau `NOP`) untuk memaksa alur ke blok sukses, meskipun server menolak autentikasi. Konteks assembly yang lebih lengkap dapat dilihat pada Lampiran 3 (Kode 5).

Sama seperti pada kasus *hard-coded* sebelumnya, instruksi ‘JNZ LAB_14000175d‘ dapat diubah menjadi ‘JZ LAB_14000175d‘. Dengan perubahan ini, program akan selalu melompat ke bagian yang menampilkan ”Authorized!” tanpa perlu melakukan verifikasi yang sebenarnya dari server. Meskipun username dan password tidak lagi disimpan di dalam aplikasi, logika di sisi klien yang menentukan apakah autentikasi berhasil atau gagal masih dapat dimanipulasi.

Ini menunjukkan bahwa meskipun memindahkan logika autentikasi ke server dan menghindari *hard-coded credential* meningkatkan keamanan, sisi klien aplikasi masih dapat menjadi target untuk *reverse engineering*. Penyerang dapat memodifikasi aplikasi klien untuk selalu menganggap autentikasi berhasil, meskipun server mungkin menolak permintaan autentikasi. Oleh karena itu, keamanan menyeluruh memerlukan perlindungan tidak hanya pada kredensial tetapi juga pada logika bisnis aplikasi di sisi klien.

Analisis serupa juga dilakukan pada aplikasi konsol dan ImGUI versi *cloud*, di mana

pola pemeriksaan kondisi dan jump kondisional yang menentukan status autentikasi juga ditemukan dan berpotensi untuk dimanipulasi.

5.1.1.3 Analisis Aplikasi Virtualized

Analisis statis pada aplikasi yang telah divirtualisasi menggunakan VxLang menunjukkan tingkat kesulitan yang jauh lebih tinggi dibandingkan versi non-virtualized. Observasi kualitatif ini didukung secara kuat oleh data kuantitatif yang diperoleh dari ringkasan analisis Ghidra terhadap berbagai aplikasi studi kasus, seperti yang disajikan pada Tabel 5.1 dan Tabel 5.2. Data komprehensif ini mengungkapkan dampak signifikan virtualisasi terhadap berbagai metrik yang relevan untuk analisis statis.

Legenda untuk Tabel 5.1 dan 5.2:

- **Aplikasi**: Nama aplikasi atau benchmark yang diuji.
- **Versi**: Menunjukkan apakah aplikasi adalah versi asli (Non-Virt.) atau versi yang telah divirtualisasi oleh VxLang (Virtualized).
- **Mem. Blocks**: Jumlah Blok Memori (*Memory Blocks*) yang diidentifikasi oleh Ghidra.
- **Instructions**: Jumlah Instruksi Mesin (*Instructions*) yang berhasil di-*disassemble* oleh Ghidra.
- **Def. Data**: Jumlah Data Terdefinisi (*Defined Data*) yang dikenali oleh Ghidra.
- **Functions**: Jumlah Fungsi (*Functions*) yang teridentifikasi oleh Ghidra.
- **Symbols**: Jumlah Simbol (*Symbols*) (misalnya, nama fungsi, variabel global) yang dikenali oleh Ghidra.
- **Data Types**: Jumlah Tipe Data (*Data Types*) yang diidentifikasi oleh Ghidra.
- **Data Cat.**: Jumlah Kategori Data (*Data Categories*) yang diklasifikasikan oleh Ghidra.
- **Perubahan %**: Persentase perubahan nilai metrik pada versi virtualisasi dibandingkan dengan versi non-virtualisasi. Nilai negatif menunjukkan penurunan.

Tabel 5.1: Perbandingan Metrik Analisis Statis Ghidra untuk Aplikasi Autentikasi (Non-Virtualized vs. Virtualized)

Aplikasi	Versi	Mem. Blocks	Instructions	Def. Data	Functions	Symbols	Data Types	Data Cat.
app_imgui	Non-Virt.	9	246329	11485	3005	33620	420	24
	Virtualized	11	157	210	25	98	87	12
	Perubahan %	+22.22%	**-99.94%**	**-98.17%**	**-99.17%**	**-99.71%**	-79.29%	-50.00%
app_imgui_cloud	Non-Virt.	9	278461	13249	4063	42807	442	24
	Virtualized	11	180	219	25	117	86	12
	Perubahan %	+22.22%	**-99.94%**	**-98.35%**	**-99.38%**	**-99.73%**	-80.54%	-50.00%
app_qt	Non-Virt.	9	6104	1578	538	2113	366	24
	Virtualized	11	214	174	25	103	68	8
	Perubahan %	+22.22%	**-96.49%**	**-88.97%**	**-95.35%**	**-95.13%**	-81.42%	-66.67%
app_qt_cloud	Non-Virt.	9	39844	3416	1700	11920	399	25
	Virtualized	11	163	210	24	127	69	8
	Perubahan %	+22.22%	**-99.59%**	**-93.85%**	**-98.59%**	**-98.93%**	-82.71%	-68.00%
console	Non-Virt.	8	3090	726	261	1018	234	18
	Virtualized	10	174	146	20	88	60	9
	Perubahan %	+25.00%	**-94.37%**	**-79.89%**	**-92.34%**	**-91.36%**	-74.36%	-50.00%
console_cloud	Non-Virt.	8	36024	2556	1370	10529	277	18
	Virtualized	10	167	173	20	81	66	9
	Perubahan %	+25.00%	**-99.54%**	**-93.23%**	**-98.54%**	**-99.23%**	-76.17%	-50.00%

Tabel 5.2: Perbandingan Metrik Analisis Statis Ghidra untuk Aplikasi Benchmark dan Lainnya (Non-Virtualized vs. Virtualized)

Aplikasi	Versi	Mem. Blocks	Instructions	Def. Data	Functions	Symbols	Data Types	Data Cat.
encryption	Non-Virt.	8	6282	849	368	1920	228	17
	Virtualized	10	159	155	20	77	62	9
	Perubahan %	+25.00%	**-97.47%**	**-81.74%**	**-94.57%**	**-95.99%**	-72.81%	-47.06%
quick_sort	Non-Virt.	8	4370	700	311	1472	229	17
	Virtualized	10	265	146	20	109	60	9
	Perubahan %	+25.00%	**-93.94%**	**-79.14%**	**-93.57%**	**-92.60%**	-73.80%	-47.06%
size	Non-Virt.	9	698	936	419	2342	277	18
	Virtualized	11	214	155	23	96	68	10
	Perubahan %	+22.22%	**-69.34%**	**-83.44%**	**-94.51%**	**-95.90%**	-75.45%	-44.44%

Data kuantitatif yang disajikan pada Tabel 5.1 dan Tabel 5.2 secara konsisten menunjukkan dampak signifikan dari virtualisasi kode menggunakan VxLang terhadap metrik-metrik analisis statis yang dihasilkan oleh Ghidra. Beberapa temuan utama dari data ini adalah:

- **Penurunan Drastis pada Metrik Kritis untuk Analisis:** Untuk hampir semua aplikasi yang diuji, terjadi penurunan yang sangat signifikan (umumnya lebih dari **90%**, dan seringkali mendekati **99%**) pada jumlah **Instruksi** (*Instructions*), **Fungsi** (*Functions*), dan **Simbol** (*Symbols*) yang dapat dikenali oleh Ghidra setelah aplikasi divirtualisasi. Jumlah **Data Terdefinisi** (*Defined Data*) juga mengalami penurunan drastis, berkisar antara **79%** hingga **98%**. Sebagai contoh spesifik, pada aplikasi `app_imgui`, jumlah instruksi yang terdeteksi turun dari 246.329 menjadi hanya 157 (**-99.94%**), fungsi dari 3.005 menjadi 25 (**-99.17%**), dan simbol dari 33.620 menjadi 98 (**-99.71%**). Pola serupa terulang pada aplikasi autentikasi dan benchmark lainnya yang diuji. **Penurunan masif ini secara langsung mencerminkan efektivitas VxLang dalam mengaburkan struktur fundamental program pada tingkat biner**. Ghidra, yang dirancang untuk menganalisis instruksi mesin standar (seperti x86-64), tidak mampu menginterpretasikan *bytecode* kustom yang dihasilkan oleh proses virtualisasi VxLang. Akibatnya, sebagian besar kode asli menjadi ”tidak terlihat” atau tidak dapat dipahami oleh *disassembler*.

- **Perubahan pada Blok Memori (*Memory Blocks*):** Jumlah blok memori menunjukkan peningkatan moderat setelah virtualisasi, umumnya sekitar **11%** hingga **25%** untuk sebagian besar aplikasi yang tercantum. Peningkatan ini kemungkinan disebabkan oleh penambahan segmen memori baru yang digunakan oleh *Virtual Machine* (VM) VxLang itu sendiri untuk menyimpan *runtime* VM, *bytecode* aplikasi, dan struktur data internal VM.

- **Pengurangan Struktur Data Tambahan:** Metrik seperti **Tipe Data** (*Data Types*) dan **Kategori Data** (*Data Categories*) juga menunjukkan penurunan yang konsisten, masing-masing berkisar antara **70%-80%** dan **40%-60%** pada aplikasi yang diuji. Ini mengindikasikan bahwa proses virtualisasi juga menyembunyikan atau mengubah cara struktur data internal dan metadata program dikenali oleh Ghidra, lebih lanjut mempersulit pemahaman terhadap organisasi data dalam aplikasi yang dilindungi.

- **Konsistensi di Seluruh Aplikasi yang Dianalisis:** Penting untuk dicatat bahwa pola perubahan ini konsisten di berbagai jenis aplikasi yang diuji dan disajikan dalam tabel, yaitu aplikasi autentikasi sederhana (konsol, Qt, ImGui) dan aplikasi *benchmark* (enkripsi, *quick sort*, ukuran). Hal ini menunjukkan bahwa dampak virtualisasi VxLang terhadap hasil analisis statis bersifat umum pada sampel aplikasi yang dipertahankan dalam analisis ini. Pengecualian kecil adalah pada metrik Instruksi untuk aplikasi `size` yang penurunannya ”hanya” **-69.34%**, kemungkinan karena sebagian besar ukuran *file* `size` adalah data mentah yang disematkan, bukan

kode yang dapat dieksekusi dan divirtualisasi.

Temuan kuantitatif ini secara meyakinkan mendukung observasi kualitatif bahwa *code virtualization* menggunakan VxLang sangat efektif dalam menghalangi analisis statis. Dengan menghilangkan atau mengaburkan sebagian besar instruksi, fungsi, simbol, dan data yang dapat dikenali, VxLang memaksa *reverse engineer* untuk menghadapi representasi program yang sangat berbeda dan tidak terdokumentasi. Upaya untuk memahami alur logika, mengidentifikasi bagian kode yang relevan (misalnya, logika autentikasi), atau mencari kerentanan melalui analisis statis konvensional menjadi sangat tidak praktis dan kemungkinan besar tidak akan berhasil tanpa pemahaman mendalam tentang arsitektur internal VM VxLang dan pemetaan *bytecode*-nya.

Kesulitan ini timbul karena, sebagaimana telah diuraikan pada Bab 2 (Sub-bab 2.3.4), *disassembler* statis seperti Ghidra mengandalkan kemampuannya untuk memetakan urutan byte ke instruksi *assembly* yang diketahui untuk arsitektur target [13], [14]. Virtualisasi kode menggantikan kode *native* ini dengan *bytecode* dari ISA kustom, yang tidak dapat diinterpretasikan dengan benar oleh Ghidra [15]. **Hasilnya adalah hilangnya informasi struktural program, seperti yang tercermin dalam penurunan drastis metrik-metrik yang diukur.**

Secara keseluruhan, data kuantitatif dari ringkasan analisis Ghidra ini memberikan bukti kuat bahwa *code virtualization* menggunakan VxLang secara signifikan meningkatkan kompleksitas analisis statis, menjadi penghalang utama bagi upaya *reverse engineering*.

5.1.2 Analisis Dinamis

Analisis dinamis dilakukan dengan menjalankan aplikasi di bawah *debugger* x64dbg untuk mengamati perilaku saat *runtime*. Tujuannya adalah untuk memverifikasi temuan dari analisis statis dan memahami bagaimana aplikasi berinteraksi dengan *input* pengguna, khususnya dalam proses autentikasi.

5.1.2.1 Analisis Aplikasi Non-Virtualized

Untuk aplikasi non-virtualized, analisis dinamis dilakukan dengan mencari string yang relevan dengan proses autentikasi, seperti username dan password yang ditemukan pada analisis statis (”seno” dan ”rahman”). *Breakpoint* dipasang pada lokasi di mana *string* ini digunakan atau dibandingkan.

Pada aplikasi *app_qt*, saat dijalankan di bawah x64dbg dan diberikan *input* yang salah, *debugger* mengarahkan eksekusi ke blok kode berikut:

```
; ... (Kode membandingkan input password dengan "rahman" via
operator==) ...
; Hasil perbandingan (boolean) mungkin disimpan di AL atau flag
register

mov al, byte ptr ss:[rbp-41]  ; Ambil hasil perbandingan
test al, 1                    ; Cek apakah hasilnya true (misal: 1)

; Perhatikan bahwa logika JNE/JE mungkin berbeda tergantung
optimasi compiler
; Asumsi: JNE melompat jika perbandingan GAGAL (tidak sama)

jne SHORT failure_label       ; Lompat ke blok GAGAL jika password
TIDAK sama
; ... (Kode jika password BENAR) ...

jmp END_OF_AUTH

failure_label:
; ... (Kode setup untuk menampilkan "Authentication Failed" via
QMessageBox) ...

lea rdx, qword ptr ds:[<"Authentication Failed">]
; ... (call ke fungsi QMessageBox) ...
```

Kode 5.3: Snippet Assembly: Lompatan Kondisional Setelah Perbandingan Password (Dinamis, Non-Virtualized)

Instruksi `test al, 1` memeriksa hasil perbandingan password. Jika password salah (misalnya, `al` adalah 0), instruksi `jne failure_label` (Jump if Not Equal/Zero) akan dieksekusi, mengarahkan program ke blok yang menampilkan ”Authentication Failed”. Dalam analisis dinamis, *debugger* seperti x64dbg memungkinkan *patching on-the-fly*. Dengan mengubah instruksi `jne` menjadi `je` (Jump if Equal/Zero) atau `jmp` (lompatan tanpa syarat ke blok sukses), pemeriksaan password dapat dilewati secara efektif saat *runtime*. Konteks assembly yang lebih lengkap dari sesi debugging ini dapat dilihat di Lampiran 3 (Kode 6).

Sama seperti pada analisis statis, celah keamanan dapat dieksploitasi dengan memodifikasi alur eksekusi. Dalam x64dbg, instruksi ‘jne‘ dapat diubah menjadi ‘je‘ (Jump if Equal) secara langsung pada saat *runtime*. Dengan melakukan perubahan ini, program akan melompat ke bagian yang seharusnya dijalankan hanya jika autentikasi berhasil, meskipun *input* yang diberikan salah. Operasi negasi ini secara efektif melewati pemeriksaan autentikasi.

5.1.2.2 Analisis Aplikasi Virtualized

Analisis dinamis pada aplikasi yang telah divirtualisasi menggunakan VxLang menghadirkan tantangan yang signifikan dibandingkan versi non-virtualized, meskipun beberapa aspek *runtime* dapat diobservasi. Data kuantitatif dari analisis menggunakan x64dbg, seperti yang disajikan pada Tabel 5.3 dan Tabel 5.4, memberikan gambaran lebih lanjut mengenai dampak virtualisasi.

Legenda untuk Tabel 5.3 dan 5.4:

- **Aplikasi**: Nama aplikasi atau benchmark yang diuji.
- **Versi**: Menunjukkan apakah aplikasi adalah versi asli (Non-VM) atau versi yang telah divirtualisasi oleh VxLang (VM), atau persentase perubahan (Perubahan %).
- **Instr. Count**: Perkiraan Jumlah Instruksi (*Instruction Count*) yang diobservasi selama sesi analisis dinamis di x64dbg.
- **Mem. Sections**: Jumlah Segmen Memori (*Memory Sections*) yang terdaftar oleh x64dbg.
- **Def. Symbols**: Jumlah Simbol Terdefinisi (*Defined Symbols*) yang dapat diobservasi di x64dbg.
- **Key Str. Found**: Status penemuan *string* kunci relevan saat analisis dinamis (Ya/Tidak/-).

Tabel 5.3: Perbandingan Metrik Analisis Dinamis x64dbg untuk Aplikasi Autentikasi (Non-VM vs. VM)

Aplikasi	Versi	Instr. Count	Mem. Sections	Def. Symbols	Key Str. Found
app_imgui	Non-VM	259046	7	251	Ya
	VM	258923	10	19	Tidak
	Perubahan %	-0.05%	+42.86%	**-92.43%**	-
app_imgui_cloud	Non-VM	296024	7	266	Ya
	VM	295890	10	20	Tidak
	Perubahan %	-0.05%	+42.86%	**-92.48%**	-
app_qt	Non-VM	8022	7	209	Ya
	VM	8011	10	15	Tidak
	Perubahan %	-0.14%	+42.86%	**-92.82%**	-
app_qt_cloud	Non-VM	49692	7	231	Ya
	VM	49744	10	19	Tidak
	Perubahan %	+0.10%	+42.86%	**-91.77%**	-
console	Non-VM	5797	6	88	Ya
	VM	5843	9	12	Tidak
	Perubahan %	+0.79%	+50.00%	**-86.36%**	-
console_cloud	Non-VM	44223	6	110	Ya
	VM	44169	9	15	Tidak
	Perubahan %	-0.12%	+50.00%	**-86.36%**	-

Tabel 5.4: Perbandingan Metrik Analisis Dinamis x64dbg untuk Aplikasi Benchmark (Non-VM vs. VM)

Aplikasi	Versi	Instr. Count	Mem. Sections	Def. Symbols	Key Str. Found
encryption	Non-VM	8336	6	94	Ya
	VM	8207	9	13	Tidak
	Perubahan %	-1.55%	+50.00%	**-86.17%**	-
quick_sort	Non-VM	6580	6	83	Ya
	VM	6492	9	12	Tidak
	Perubahan %	-1.34%	+50.00%	**-85.54%**	-
size	Non-VM	9948	7	95	Ya
	VM	9870	10	12	Tidak
	Perubahan %	-0.78%	+42.86%	**-87.37%**	-

Observasi utama adalah sebagai berikut:

- **Visibilitas Instruksi *Native* dari *Virtual Machine* (VM) VxLang vs. Jumlah Instruksi Teramati:** Sebagaimana diobservasi secara kualitatif, setelah aplikasi tervirtualisasi dimuat sepenuhnya, **x64dbg mampu menampilkan urutan instruksi *native* x86-64 yang valid yang sedang dieksekusi.** Instruksi-instruksi ini adalah bagian dari *interpreter* VM VxLang. Data pada Tabel 5.3 dan Tabel 5.4 menunjukkan bahwa "Jumlah Instruksi" yang teramati selama sesi analisis pada versi tervirtualisasi seringkali tidak jauh berbeda dibandingkan versi non-virtualisasi. Perubahannya berkisar antara -1.55% (pada `encryption`) hingga +0.79% (pada `console`), yang menunjukkan bahwa jumlah total instruksi yang dilalui selama observasi tidak secara drastis berubah. **Penting untuk ditekankan bahwa jumlah instruksi ini pada versi VM tidak secara langsung merefleksikan kompleksitas atau kemudahan analisis logika aplikasi asli.** Sebaliknya, angka ini lebih mencerminkan aktivitas VM VxLang itu sendiri dalam menjalankan *bytecode*. Fakta bahwa instruksi *native* VM terlihat tidak berarti logika aplikasi asli menjadi transparan.

- **Peningkatan Jumlah Segmen Memori (*Memory Sections*):** Data secara konsisten menunjukkan peningkatan jumlah segmen memori untuk aplikasi yang divirtualisasi, yaitu sekitar **+42.86%** untuk aplikasi yang awalnya memiliki 7 segmen memori (seperti `app_imgui` dan `app_qt`) dan **+50.00%** untuk aplikasi yang awalnya memiliki 6 segmen memori (seperti `console` dan `encryption`). Peningkatan ini kemungkinan besar disebabkan oleh pemetaan segmen memori tambahan yang

diperlukan oleh *runtime* VxLang untuk VM itu sendiri, *bytecode*, dan data internal lainnya.

- **Penurunan Drastis Simbol Terdefinisi (*Defined Symbols*):** Salah satu dampak paling signifikan yang teramati secara kuantitatif adalah penurunan jumlah simbol terdefinisi yang dapat diobservasi oleh x64dbg. Untuk semua aplikasi yang diuji, terjadi penurunan drastis, umumnya lebih dari **85%** (misalnya, `app_imgui` dari 251 menjadi 19 simbol, perubahan **-92.43%**; `encryption` dari 94 menjadi 13 simbol, perubahan **-86.17%**). **Hilangnya informasi simbolik ini sangat mempersulit upaya *reverse engineer* untuk memahami konteks kode, menavigasi antar fungsi, atau mengidentifikasi variabel penting saat analisis dinamis.**

- **Kegagalan Menemukan *String* Kunci (*Key Strings Found*):** Metrik ini menunjukkan efektivitas paling jelas dari VxLang dalam analisis dinamis. Untuk **semua** aplikasi yang divirtualisasi, *string-string* kunci yang relevan dengan autentikasi (seperti ”Authentication Failed”, ”Authorized”, atau kredensial jika ada) **tidak berhasil ditemukan** melalui pencarian memori standar atau referensi *string* di x64dbg saat aplikasi berjalan (ditandai sebagai ”Tidak” pada tabel). Sebaliknya, pada semua versi non-virtualisasi, *string* kunci ini dapat ditemukan (”Ya”). **Ini membuktikan bahwa VxLang efektif dalam menyembunyikan atau mengenkripsi konstanta *string* yang ditargetkan, bahkan dari inspeksi memori saat *runtime*.**

- **Kesulitan Utama Tetap pada Abstraksi Logika Aplikasi:** Meskipun instruksi *native* dari VM VxLang dapat dilihat, tantangan fundamental dalam analisis dinamis tetap ada: logika aplikasi yang sebenarnya (misalnya, proses perbandingan kredensial dan pengambilan keputusan autentikasi) telah ditransformasi menjadi representasi *bytecode* internal. Seorang analis yang menggunakan x64dbg hanya melihat eksekusi dari instruksi-instruksi VM VxLang, bukan eksekusi langsung dari logika aplikasi dalam bentuk *native* aslinya. Hal ini membuat pemahaman terhadap alur logika aplikasi yang sebenarnya dan identifikasi titik-titik keputusan krusial menjadi sangat sulit tanpa pemahaman mendalam mengenai arsitektur VM VxLang.

- **Kegagalan Upaya *Bypass* Sederhana:** Sebagai akibat langsung dari abstraksi logika aplikasi ke dalam VM dan hilangnya informasi kontekstual seperti simbol dan *string* kunci, upaya untuk melakukan *bypass* mekanisme autentikasi melalui teknik *patching runtime* sederhana (seperti memodifikasi satu instruksi *jump* kondisional sebagaimana berhasil dilakukan pada versi non-virtualisasi) menjadi tidak

efektif. Titik keputusan logika autentikasi tidak lagi terekspos sebagai instruksi *native* yang mudah diidentifikasi dan dimodifikasi.

Temuan kuantitatif dan kualitatif ini selaras dengan prinsip dasar *code virtualization* yang dijelaskan pada Bab 2 (Sub-bab 2.3.4.1). Meskipun *debugger* dinamis seperti x64dbg dapat me-*disassemble* dan melangkah melalui instruksi *native* dari *interpreter* VM VxLang itu sendiri [13], logika aplikasi asli yang telah divirtualisasi tetap tersembunyi dalam bentuk *bytecode*. Penurunan drastis simbol yang dapat diobservasi dan kegagalan menemukan *string* kunci secara signifikan menghambat proses analisis.

Dengan demikian, meskipun instruksi *native* dari VM VxLang dapat diobservasi saat *runtime*, analisis dinamis terhadap bagian kode aplikasi yang telah divirtualisasi dengan VxLang tetap menghadirkan tantangan besar dalam hal melacak data sensitif dan memahami serta memanipulasi alur logika aplikasi inti. Ini secara signifikan meningkatkan kompleksitas dan waktu yang dibutuhkan untuk melakukan *reverse engineering* yang bertujuan untuk *bypass* atau modifikasi.

5.2 Analisis Performa *Overhead* VxLang

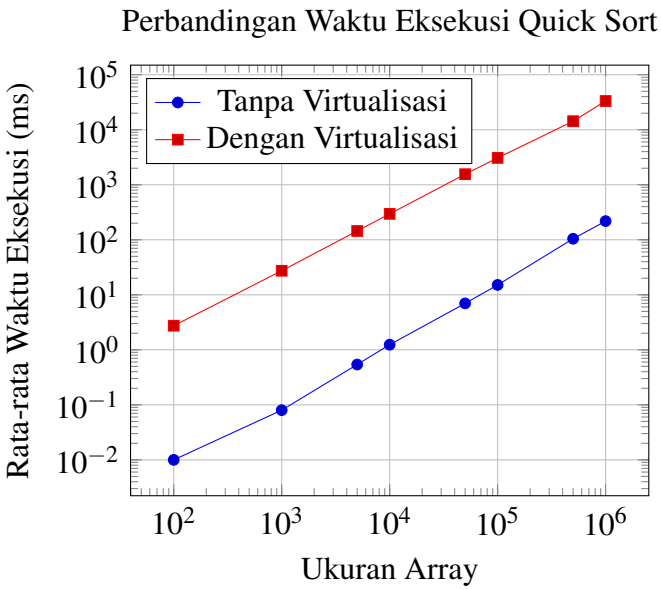
Hasil dari percobaan performa overhead dan perubahan ukuran file setelah penerapan virtualisasi kode menggunakan VxLang. Percobaan ini dilakukan pada algoritma Quick Sort dan enkripsi AES-CBC-256.

5.2.1 Hasil Pengujian Performa *Quick Sort*

Tabel 5.5 menyajikan hasil pengukuran waktu rata-rata dan standar deviasi dari algoritma Quick Sort yang dijalankan sebanyak 100 kali untuk setiap ukuran array, baik sebelum maupun sesudah virtualisasi menggunakan VxLang.

Tabel 5.5: Hasil Pengujian Waktu Eksekusi Quick Sort (ms)

Ukuran Array	Tanpa Virtualisasi		Dengan Virtualisasi	
	Rata-rata Waktu (ms)	**Standar Deviasi (ms)**	**Rata-rata Waktu (ms)**	**Standar Deviasi (ms)**
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



Gambar 5.2: Perbandingan Waktu Eksekusi Algoritma Quick Sort antara Versi Tanpa dan Dengan Virtualisasi VxLang.

Berdasarkan Tabel 5.5 dan Gambar 5.2, terlihat adanya peningkatan waktu eksekusi yang sangat signifikan pada algoritma Quick Sort setelah divirtualisasi menggunakan VxLang. Peningkatan ini konsisten terjadi seiring dengan bertambahnya ukuran array. Sebagai contoh:

- Untuk array berukuran 100 elemen, waktu eksekusi rata-rata meningkat dari 0.01 ms menjadi 2.74 ms, yang menunjukkan *overhead* sekitar 27,300%.
- Untuk array berukuran 100.000 elemen, waktu eksekusi meningkat dari 15.12 ms menjadi 3.080.30 ms, dengan *overhead* sekitar 20,272%.

- Untuk array terbesar yang diuji pada versi virtualisasi (1.000.000 elemen), waktu eksekusi meningkat dari 218.32 ms menjadi 33.292.91 ms, menghasilkan *overhead* sekitar 15,150%.

Peningkatan standar deviasi juga menunjukkan bahwa waktu eksekusi pada versi virtualisasi menjadi lebih bervariasi, terutama pada ukuran array yang lebih besar. Hal ini mengindikasikan adanya *overhead* yang substansial dan kurang prediktabilitas yang diperkenalkan oleh mesin virtual VxLang dalam mengeksekusi instruksi virtual dibandingkan dengan eksekusi kode *native*.

5.2.2 Hasil Pengujian Performa Enkripsi AES-CBC-256

Tabel 5.6 menyajikan hasil benchmarking enkripsi AES-CBC-256 dengan 1.000.000 blok data (total 976 MB), sebelum dan sesudah virtualisasi menggunakan VxLang.

Tabel 5.6: Hasil Pengujian Performa Enkripsi AES-CBC-256

Metrik	**Tanpa Virtualisasi**	**Dengan Virtualisasi**
Total Waktu Enkripsi (ms)	1,878.52	9,330.73
Total Waktu Dekripsi (ms)	1,304.75	8,649.74
Rata-rata Waktu per Blok Enkripsi (ms)	0.00188	0.00933
Rata-rata Waktu per Blok Dekripsi (ms)	0.00130	0.00865
Throughput Enkripsi (MB/s)	519.86	104.66
Throughput Dekripsi (MB/s)	748.46	112.90
Throughput Gabungan (MB/s)	634.16	108.78

Hasil pengujian enkripsi AES-CBC-256 menunjukkan *overhead* performa yang signifikan setelah penerapan VxLang.

- Total waktu enkripsi meningkat dari 1,878.52 ms menjadi 9,330.73 ms, yang merupakan peningkatan sekitar 396.7%.
- Total waktu dekripsi mengalami peningkatan dari 1,304.75 ms menjadi 8,649.74 ms, atau sekitar 562.9%.

Peningkatan waktu eksekusi ini berdampak langsung pada penurunan *throughput* (kecepatan pemrosesan data):

- *Throughput* enkripsi menurun dari 519.86 MB/s menjadi 104.66 MB/s (penurunan sekitar 79.9%).
- *Throughput* dekripsi menurun dari 748.46 MB/s menjadi 112.90 MB/s (penurunan sekitar 84.9%).

- *Throughput* gabungan (rata-rata enkripsi dan dekripsi) menurun dari 634.16 MB/s menjadi 108.78 MB/s, yang merupakan penurunan sekitar 82.8%.

Hasil ini mengkonfirmasi bahwa virtualisasi kode dengan VxLang memperkenalkan *overhead* yang cukup besar pada operasi komputasi intensif seperti enkripsi.

5.2.3 Hasil Pengujian Ukuran File

Tabel 5.7 menyajikan ukuran file *executable* (dalam KB) untuk berbagai program sebelum dan sesudah virtualisasi menggunakan VxLang.

Tabel 5.7: Hasil Pengujian Ukuran File (KB)

Program	**Ukuran File**	
	Tanpa Virtualisasi	**Virtualiasi**
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

Hasil pengukuran ukuran file menunjukkan bahwa penerapan virtualisasi kode menggunakan VxLang secara konsisten meningkatkan ukuran file *executable*.

- Untuk program-program kecil dan sederhana seperti `quick_sort` (98 KB menjadi 1.537 KB, peningkatan sekitar 15.7x), `console` (92 KB menjadi 1.577 KB, peningkatan sekitar 17.1x), dan `Lilith_Client` (84 KB menjadi 1.554 KB, peningkatan sekitar 18.5x), peningkatannya sangat signifikan, seringkali lebih dari 15 kali lipat.

- Untuk aplikasi GUI yang lebih besar atau yang menggunakan library eksternal seperti `app_imgui` (1.675 KB menjadi 2.330 KB, peningkatan sekitar 1.39x) dan `app_qt_cloud` (315 KB menjadi 1.671 KB, peningkatan sekitar 5.3x), persentase peningkatannya lebih kecil, namun absolutnya tetap menunjukkan penambahan ukuran yang cukup besar (ratusan hingga ribuan KB).

- Aplikasi `size` yang dirancang dengan aset data tersemat besar (97.771 KB) menunjukkan peningkatan ukuran yang relatif paling kecil secara persentase (menjadi

112.324 KB, peningkatan sekitar 1.15x atau 15%), namun ini tetap berarti penambahan sekitar 14.5 MB. Ini mengindikasikan bahwa *overhead* utama berasal dari *runtime* VxLang itu sendiri, dan dampaknya lebih terasa pada aplikasi kecil.

Peningkatan ukuran ini, kemungkinan besar disebabkan oleh penambahan *interpreter* mesin virtual VxLang dan representasi *bytecode* dari kode asli ke dalam *executable* yang dilindungi. Peningkatan ini konsisten di semua jenis aplikasi yang diuji.

5.3 Analisis Aplikasi Potensial Berbahaya dan Malware dengan VxLang

5.3.1 Pengujian Fungsionalitas Lilith RAT Tervirtualisasi

Bagian ini menyajikan hasil analisis terhadap aplikasi RAT Lilith, baik dalam versi asli maupun versi yang telah divirtualisasi menggunakan VxLang. Analisis berfokus pada pengujian fungsionalitas klien Lilith setelah virtualisasi untuk memastikan integritas operasionalnya, serta dampak virtualisasi terhadap deteksi oleh layanan pemindaian *malware* VirusTotal.

Untuk memastikan bahwa proses virtualisasi menggunakan VxLang tidak merusak fungsionalitas inti dari aplikasi yang kompleks seperti Lilith RAT, sebuah pengujian fungsional secara langsung dilakukan. Skenario pengujian ini dirancang untuk memverifikasi kemampuan operasional dasar klien Lilith yang telah divirtualisasi ketika berinteraksi dengan server Lilith dalam lingkungan jaringan lokal.

5.3.1.1 Skenario dan Konfigurasi Pengujian

Pengujian fungsionalitas Lilith RAT yang telah divirtualisasi dilakukan dengan konfigurasi sebagai berikut:

- **Server Lilith:** Dijalankan pada laptop pertama dengan alamat IP lokal `192.168.1.235`. Server Lilith mendengarkan koneksi masuk pada *port* `1337`. Aplikasi server yang digunakan adalah versi asli yang tidak dimodifikasi (`Lilith_Server.exe`).
- **Klien Lilith:** Dijalankan pada laptop kedua, yang berada dalam jaringan lokal yang sama, dengan alamat IP `192.168.1.15`. Aplikasi klien yang diuji adalah versi yang telah diproses dan divirtualisasi oleh VxLang (`Lilith_Client_vm.vxm.exe`).
- **Target Demonstrasi Fungsionalitas:** Untuk mendemonstrasikan kemampuan akses sistem berkas dan eksekusi perintah jarak jauh, sebuah berkas teks bernama `password.txt` dibuat pada direktori kerja aplikasi klien Lilith di laptop kedua. Berkas ini berisi *string* sederhana: `"THIS IS A SECRET"`.

5.3.1.2 Langkah-Langkah Demonstrasi dan Hasil Observasi

Setelah kedua aplikasi (server Lilith dan klien Lilith yang tervirtualisasi) dijalankan pada masing-masing laptop, serangkaian interaksi dilakukan dari terminal server, dan hasilnya diobservasi:

1. **Deteksi Koneksi Klien:** Terminal pada server Lilith segera menampilkan notifikasi bahwa koneksi dari klien telah berhasil dibuat, dengan pesan: `"Client Connected! ID: 0 | IP: 192.168.1.15"`. Ini mengkonfirmasi bahwa klien tervirtualisasi mampu melakukan inisiasi koneksi jaringan dengan benar.

2. **Menghubungkan ke Sesi Klien:** Operator di sisi server memasukkan perintah `connect 0` untuk secara aktif terhubung dan mengontrol sesi klien dengan ID 0.

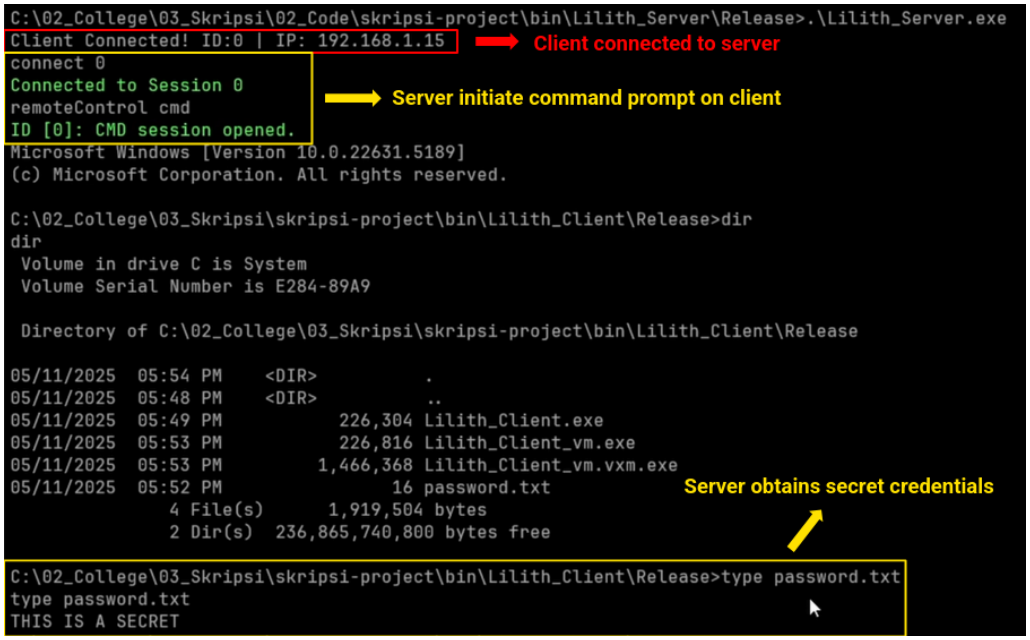
3. **Aktivasi Kontrol Jarak Jauh (CMD):** Perintah `remoteControl cmd` kemudian dimasukkan oleh operator server. Terminal server merespons dengan pesan dari klien: `"ID [0]: CMD session opened."`, yang menandakan bahwa permintaan untuk membuka sesi *command prompt* jarak jauh pada mesin klien telah berhasil diproses.

4. **Akses ke Direktori Klien:** Setelah sesi CMD jarak jauh aktif, *prompt command prompt* pada terminal server berubah, menunjukkan bahwa konteks eksekusi kini berada di direktori kerja aplikasi klien pada laptop kedua, yaitu `C:\02_College\03_Skripsi_\skripsi-project\bin\Lilith_Client\Release>`.

5. **Verifikasi Isi Direktori:** Operator server menjalankan perintah `dir` dalam sesi CMD jarak jauh. Hasil dari perintah ini, yang ditampilkan di terminal server, mencantumkan berkas `password.txt` dalam daftar isi direktori klien, memvalidasi kemampuan untuk melakukan enumerasi direktori.

6. **Membaca Isi Berkas Jarak Jauh:** Langkah terakhir adalah menjalankan perintah `type password.txt` pada terminal server. Perintah ini dieksekusi pada mesin klien, dan isinya (string `"THIS IS A SECRET"`) berhasil diambil dan ditampilkan kembali pada terminal server.

Seluruh rangkaian interaksi ini, yang menunjukkan keberhasilan server dalam mengontrol klien tervirtualisasi dan mengakses berkasnya, didokumentasikan dalam bentuk tangkapan layar terminal server yang disajikan pada Gambar 5.3.



Gambar 5.3: Demonstrasi Fungsionalitas Lilith RAT Tervirtualisasi dari Terminal Server: Koneksi, Akses CMD, dan Pembacaan Berkas.

5.3.1.3 Analisis Hasil Pengujian Fungsionalitas Lilith

Demonstrasi fungsionalitas ini secara meyakinkan membuktikan bahwa proses virtualisasi kode menggunakan VxLang pada aplikasi Lilith RAT **tidak mengganggu atau merusak fungsionalitas intinya**. Klien Lilith yang telah divirtualisasi tetap mampu menjalankan operasi-operasi fundamental seperti membangun koneksi jaringan, menerima dan memproses perintah dari server, serta berinteraksi dengan sistem berkas pada mesin klien untuk membaca data.

Temuan ini memiliki implikasi penting. Ini menunjukkan bahwa VxLang, sebagai sebuah teknik proteksi, dapat diterapkan pada aplikasi yang relatif kompleks dengan berbagai modul fungsional (seperti komunikasi jaringan, pemrosesan input/output, dan interaksi sistem operasi) tanpa menyebabkan kegagalan operasional. Kemampuan untuk mempertahankan fungsionalitas asli adalah prasyarat krusial sebelum mengevaluasi lebih lanjut efektivitas keamanan dan dampak performa dari sebuah teknik proteksi perangkat lunak. Jika proteksi merusak fungsi dasar aplikasi, maka teknik tersebut tidak praktis untuk digunakan, terlepas dari seberapa kuat tingkat keamanannya. Dalam kasus ini, Lilith RAT tervirtualisasi tetap menjadi alat yang fungsional, yang berarti analisis terhadap kesulitan *reverse engineering* dan *overhead* performa menjadi relevan dan bermakna.

Penting untuk dicatat bahwa keberhasilan fungsionalitas klien Lilith yang tervirtualisasi ini dicapai setelah melalui beberapa iterasi penyesuaian penempatan makro VxLang. Sebagaimana telah diuraikan pada Bab 4.4.3, upaya awal untuk memvirtualisasi blok kode yang lebih besar atau yang melibatkan interaksi I/O jaringan secara langsung dalam

lingkup virtualisasi seringkali menyebabkan ketidakstabilan aplikasi atau kegagalan fungsional. Sebagai contoh, pembungkusan langsung seluruh blok `case` dalam penanganan paket atau pemanggilan fungsi pengiriman data jaringan (`SendString`) dengan makro virtualisasi mengakibatkan *crash*. Fungsionalitas yang stabil baru tercapai ketika makro virtualisasi diterapkan secara lebih hati-hati dan selektif pada bagian-bagian kode yang lebih terisolasi atau pada level pemanggilan fungsi tertentu yang tidak secara langsung mengelola state I/O kompleks. Temuan ini menyoroti bahwa meskipun virtualisasi dapat diterapkan pada aplikasi kompleks seperti Lilith RAT, prosesnya memerlukan pertimbangan teknis yang mendalam mengenai interaksi antara kode yang divirtualisasi dengan sisa sistem dan batasan dari *tool* virtualisasi itu sendiri.

5.3.2 Analisis Deteksi Malware menggunakan VirusTotal

Untuk mengevaluasi bagaimana virtualisasi kode VxLang mempengaruhi deteksi oleh perangkat lunak antivirus/antimalware secara lebih luas, sepuluh sampel perangkat lunak—termasuk klien Lilith RAT dan sembilan sampel *malware* lainnya—diunggah ke layanan VirusTotal. Tabel 5.8 merangkum jumlah deteksi oleh *engine* antivirus VirusTotal untuk setiap sampel sebelum (Non-VM) dan sesudah (VM) diterapkan virtualisasi VxLang. Total jumlah *engine* pemindai yang aktif di VirusTotal saat pengujian adalah 72.

Tabel 5.8: Perbandingan Jumlah Deteksi VirusTotal untuk Berbagai Sampel Malware (Non-VM vs. VM dari 72 Engine)

Malware	**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**

Observasi utama dari hasil analisis VirusTotal pada kesepuluh sampel ini adalah:

- **Variasi Dampak Virtualisasi terhadap Deteksi:** Hasil deteksi menunjukkan

dampak yang bervariasi antar sampel. Untuk lima dari sepuluh sampel (*Lilith_Client, Al-Khaser, donut, DripLoader, ParadoxiaClient*), terjadi **penurunan** jumlah deteksi setelah virtualisasi. Penurunan paling signifikan terlihat pada sampel *donut* (-11 deteksi, dari 30 menjadi 19) dan *Lilith_Client* serta *Al-Khaser* (keduanya -4 deteksi). Hal ini mengindikasikan bahwa untuk sampel-sampel ini, VxLang kemungkinan berhasil mengaburkan *signature* statis atau heuristik yang dikenali oleh sebagian *engine* antivirus.

- **Peningkatan Deteksi pada Beberapa Sampel:** Sebaliknya, untuk lima sampel lainnya—*FilelessPELoader* (16 menjadi 21), *JuicyPotato* (9 menjadi 20), *PELoader* (14 menjadi 17), *RunPE-In-Memory* (12 menjadi 16), dan *SigLoader* (16 menjadi 17)—terjadi **peningkatan** jumlah deteksi setelah virtualisasi. Peningkatan paling drastis terlihat pada *JuicyPotato* (+11 deteksi) dan *FilelessPELoader* (+5 deteksi). Fenomena ini menunjukkan bahwa meskipun virtualisasi dapat menyembunyikan *signature* asli dari *malware*, proses virtualisasi itu sendiri atau karakteristik dari *executable* yang tervirtualisasi (misalnya, penggunaan perilaku seperti *packer*, perubahan entropi, atau penambahan segmen kode VM) dapat memicu deteksi oleh *engine* antivirus lain. Beberapa *engine* mungkin menandai *file* yang diproteksi secara umum sebagai mencurigakan atau "*RiskWare*".

- **Tidak Ada Eliminasi Deteksi Total dan Rata-rata Perubahan Minimal:** Meskipun ada perubahan signifikan pada sampel individual, tidak ada sampel yang menjadi sepenuhnya tidak terdeteksi (0 deteksi) setelah virtualisasi. Secara keseluruhan, rata-rata perubahan jumlah deteksi untuk kesepuluh sampel adalah +0.4. Ini mengindikasikan bahwa secara agregat, virtualisasi VxLang pada sampel-sampel ini tidak menghasilkan penurunan deteksi yang signifikan secara umum; dampaknya sangat bergantung pada interaksi antara sampel spesifik, cara VxLang memprosesnya, dan bagaimana berbagai *engine* antivirus merespons terhadap karakteristik *malware* asli versus karakteristik *wrapper* virtualisasi.

- **Perubahan Karakteristik Deteksi (Observasi Kualitatif):** Selain perubahan jumlah deteksi, penting untuk dicatat bahwa label ancaman seringkali berubah. Untuk sampel yang mengalami penurunan deteksi, label bisa berubah dari spesifik (misalnya, nama keluarga malware) menjadi lebih generik (misalnya, "Trojan" umum) atau menjadi deteksi berbasis AI/ML dan heuristik. Untuk sampel yang mengalami peningkatan deteksi, label baru mungkin mencerminkan deteksi terhadap lapisan proteksi VxLang itu sendiri sebagai sesuatu yang mencurigakan atau dikemas.

5.3.2.1 Pembahasan Hasil Analisis VirusTotal

Hasil analisis VirusTotal yang beragam ini memberikan perspektif yang lebih bernuansa mengenai efektivitas *code virtualization* sebagai teknik penghindaran deteksi.

1. **Pengaburan *Signature* vs. Deteksi Protektor:** Penurunan deteksi pada separuh sampel menegaskan kemampuan VxLang untuk mengaburkan *signature* statis. Namun, peningkatan deteksi pada separuh sampel lainnya, terutama pada *JuicyPotato* dan *FilelessPELoader*, menunjukkan bahwa lapisan virtualisasi itu sendiri dapat menjadi target deteksi. *Engine* antivirus mungkin dilatih untuk mengenali pola yang dihasilkan oleh *virtualizer* atau *packer* umum, dan VxLang, meskipun bertujuan untuk proteksi, dapat secara tidak sengaja memicu *flag* ini.

2. **Ketergantungan pada Sampel Malware dan Proses Virtualisasi:** Efektivitas penghindaran deteksi sangat bergantung pada karakteristik asli *malware* dan bagaimana VxLang berinteraksi dengan struktur kodenya, terutama ketika memproses *executable* biner secara langsung tanpa panduan makro. *Malware* yang sudah sangat diobfuskasi atau memiliki struktur yang kompleks mungkin tidak banyak berubah profil deteksinya, atau bahkan bisa menjadi lebih terdeteksi jika lapisan VxLang menambahkan artefak yang dikenali oleh *engine* antivirus sebagai mencurigakan.

3. **Evolusi Teknik Deteksi Antivirus:** *Engine* antivirus terus berkembang. Ketergantungan pada *signature* statis berkurang, sementara analisis heuristik, perilaku, dan AI/ML meningkat. Teknik seperti *code virtualization* mungkin efektif terhadap deteksi berbasis *signature* yang lebih tua, tetapi mungkin kurang efektif atau bahkan kontraproduktif terhadap mekanisme deteksi yang lebih modern yang mencari anomali struktural, perilaku yang mencurigakan dari *software* yang diproteksi, atau bahkan ciri-ciri dari *virtualizer* itu sendiri.

4. **Implikasi Keamanan:** Temuan ini menggarisbawahi bahwa meskipun VxLang dapat secara signifikan mempersulit *reverse engineering* manual, dampaknya terhadap deteksi otomatis oleh antivirus bersifat kompleks. Pada beberapa kasus, ia dapat membantu menghindari deteksi berbasis *signature*, namun pada kasus lain, ia justru dapat meningkatkan kecurigaan dari *engine* antivirus. Ini menunjukkan bahwa tidak ada solusi tunggal untuk penghindaran deteksi, dan efektivitasnya harus dievaluasi per kasus.

Secara keseluruhan, analisis VirusTotal pada berbagai sampel *malware* menunjukkan bahwa *code virtualization* dengan **VxLang memiliki dampak yang beragam dan tidak**

selalu menghasilkan penurunan tingkat deteksi. Faktor-faktor seperti karakteristik sampel asli dan sensitivitas *engine* antivirus terhadap teknik proteksi memainkan peran penting.

BAB 6
KESIMPULAN DAN SARAN

6.1 Kesimpulan

Berdasarkan implementasi dan analisis yang telah dilakukan terhadap efektivitas *code virtualization* menggunakan VxLang untuk mempersulit *reverse engineering*, dapat ditarik kesimpulan sebagai berikut:

1. **Implementasi VxLang:** Implementasi *code virtualization* menggunakan VxLang melibatkan penandaan bagian kode sumber yang kritikal menggunakan makro SDK (`VL_VIRTUALIZATION_BEGIN/END`), kompilasi menjadi *executable intermediate* (`*_vm.exe`) yang tertaut dengan library VxLang, dan pemrosesan *executable intermediate* tersebut secara langsung menggunakan *tool command-line* `vxlang.exe` untuk menghasilkan *executable* akhir yang tervirtualisasi (`*_vxm.exe`). Proses ini mengubah kode *native* yang ditandai menjadi *bytecode* yang dieksekusi oleh *virtual machine* (VM) internal VxLang. **Penelitian ini juga menemukan bahwa penempatan makro virtualisasi secara strategis dan hati-hati sangat krusial, karena penempatan yang kurang tepat pada struktur kode tertentu, terutama yang melibatkan alur kontrol kompleks atau operasi I/O, dapat merusak fungsionalitas aplikasi, sebagaimana teramati pada studi kasus Lilith RAT.**

2. **Efektivitas Keamanan:** *Code virtualization* menggunakan VxLang terbukti **efektif secara signifikan** dalam meningkatkan kesulitan *reverse engineering* baik secara statis maupun dinamis.

    - Pada **analisis statis** (menggunakan Ghidra), kode yang divirtualisasi menunjukkan hilangnya *string-string* penting, pengurangan drastis data terdefinisi, munculnya banyak operasi *assembly* yang tidak dikenali ('???'), dan abstraksi alur kontrol logika inti (seperti perbandingan dan lompatan kondisional pada fungsi autentikasi), sehingga identifikasi dan *patching* langsung menjadi sangat sulit.
    - Pada **analisis dinamis** (menggunakan x64dbg), meskipun instruksi *native* dari *Virtual Machine* (VM) VxLang dapat diobservasi setelah aplikasi dimuat, pelacakan alur logika aplikasi asli dan upaya manipulasi *runtime* untuk *bypass* autentikasi tetap sangat terhambat. *String-string* krusial yang telah divirtualisasi (misalnya, pesan status autentikasi seperti "Authentication Failed")

tidak dapat ditemukan melalui metode pencarian standar di dalam *debugger*. Logika keputusan inti aplikasi telah terabstraksi ke dalam mekanisme eksekusi VM, sehingga upaya untuk mengidentifikasi dan memodifikasi instruksi *jump* kondisional secara langsung—yang efektif pada versi non-virtualisasi—menjadi **tidak berhasil** pada kode yang dilindungi VxLang. Akibatnya, manipulasi *runtime* untuk melewati logika (seperti autentikasi) menjadi jauh lebih kompleks dan memerlukan pemahaman mendalam tentang cara kerja VM VxLang.

3. **Dampak Performa dan Ukuran File:** Penerapan VxLang memperkenalkan ***overhead* performa yang substansial** dan **peningkatan ukuran file yang signifikan**.

    - Waktu eksekusi untuk tugas komputasi intensif seperti algoritma QuickSort dan enkripsi/dekripsi AES-CBC-256 meningkat drastis (ratusan hingga puluhan ribu persen untuk QuickSort, dan beberapa ratus persen untuk AES dengan penurunan *throughput* yang tajam).
    - Ukuran file *executable* meningkat secara signifikan, terutama pada program kecil (bisa lebih dari 10 kali lipat), kemungkinan besar karena penyertaan *interpreter* VM VxLang dan *bytecode*.

4. **Trade-off Keamanan vs. Performa:** Terdapat *trade-off* yang jelas antara peningkatan keamanan terhadap *reverse engineering* yang ditawarkan oleh VxLang dengan penurunan kinerja eksekusi dan penambahan ukuran file. VxLang memberikan lapisan proteksi yang kuat namun dengan biaya performa yang tidak dapat diabaikan.

5. **Pengaruh terhadap Deteksi Malware:** Studi kasus menggunakan aplikasi RAT Lilith dan sembilan sampel *malware/PUA* lainnya menunjukkan bahwa virtualisasi kode dengan VxLang memiliki dampak yang **bervariasi** terhadap tingkat deteksi oleh layanan pemindaian *malware* VirusTotal. Pada sekitar separuh sampel yang diuji, jumlah vendor yang mendeteksi sampel sebagai berbahaya menurun setelah virtualisasi, seringkali dengan perubahan karakteristik deteksi dari spesifik menjadi lebih generik atau berbasis heuristik/AI. Namun, pada separuh sampel lainnya, jumlah deteksi justru **meningkat** setelah virtualisasi. Hal ini mengindikasikan bahwa meskipun VxLang dapat mengaburkan *signature* statis asli, proses virtualisasi itu sendiri atau artefak yang dihasilkannya dapat dikenali sebagai mencurigakan oleh beberapa *engine* antivirus. Secara keseluruhan, virtualisasi VxLang tidak menjamin penghindaran deteksi total dan dampaknya sangat bergantung pada sampel spesifik serta sensitivitas *engine* pendeteksi.

Secara keseluruhan, penelitian ini menunjukkan bahwa *code virtualization* dengan VxLang adalah teknik yang ampuh untuk melindungi perangkat lunak dari analisis dan manipulasi oleh *reverse engineer*, namun pengembang perlu mempertimbangkan dampaknya terhadap performa dan ukuran aplikasi secara cermat.

6.2 Saran

Berdasarkan kesimpulan dari penelitian ini, berikut adalah beberapa saran yang dapat diberikan:

1. **Bagi Pengembang Perangkat Lunak:**
    - **Virtualisasi Selektif:** Mengingat *overhead* performa yang signifikan, disarankan untuk menerapkan *code virtualization* VxLang secara selektif hanya pada bagian kode yang paling kritikal dan sensitif (misalnya, mekanisme validasi lisensi, algoritma inti yang merupakan kekayaan intelektual, fungsi anti-cheat, atau bagian dari logika autentikasi yang kompleks), bukan pada keseluruhan aplikasi.
    - **Evaluasi Performa Mendalam:** Sebelum mengimplementasikan VxLang pada produk komersial, lakukan pengujian performa yang menyeluruh pada kasus penggunaan nyata aplikasi tersebut untuk memastikan dampaknya masih dapat diterima oleh pengguna akhir dan tidak mengganggu fungsionalitas secara signifikan.
    - **Investigasi Batasan dan Interaksi Penempatan Makro VxLang:** Melakukan penelitian lebih mendalam mengenai batasan-batasan teknis terkait penempatan makro VxLang. Ini dapat mencakup identifikasi jenis-jenis konstruksi kode, pola alur kontrol, atau interaksi dengan API sistem/jaringan yang rentan menyebabkan masalah fungsionalitas ketika divirtualisasi. Hasilnya dapat berupa panduan atau heuristik yang lebih baik bagi pengembang dalam menerapkan VxLang.
    - **Kombinasi Teknik:** Pertimbangkan untuk mengkombinasikan VxLang dengan teknik *obfuscation* atau proteksi lainnya (seperti anti-debugging, enkripsi data, atau *code flattening* pada bagian yang tidak divirtualisasi) untuk menciptakan lapisan keamanan yang lebih mendalam (*defense-in-depth*).
2. **Untuk Penelitian Selanjutnya:**
    - **Analisis Keamanan Lebih Lanjut:** Melakukan analisis terhadap ketahanan VxLang menggunakan teknik *reverse engineering* yang lebih canggih

atau *tool deobfuscation* otomatis yang mungkin dikembangkan khusus untuk menargetkan proteksi berbasis VM.

- **Optimasi Performa:** Meneliti kemungkinan untuk mengurangi *overhead* performa yang disebabkan oleh VxLang, misalnya dengan menganalisis dan mengoptimalkan *interpreter* VM internalnya, atau mengeksplorasi arsitektur VM yang berbeda.
- **Keamanan Interpreter VM:** Menganalisis potensi kerentanan pada *interpreter* VM VxLang itu sendiri yang mungkin dapat dieksploitasi untuk membongkar atau memanipulasi *bytecode*.
- **Studi Komparatif:** Membandingkan efektivitas keamanan dan *overhead* performa VxLang dengan solusi *code virtualization* atau *obfuscation* komersial maupun *open-source* lainnya (misalnya, VMProtect, Themida, LLVM Obfuscator).
- **Dukungan Platform Lain:** Melakukan analisis serupa jika/ketika VxLang telah mendukung platform lain seperti Linux ELF atau arsitektur ARM, untuk melihat apakah efektivitas dan dampaknya konsisten.
- **Pengaruh Konfigurasi:** Menyelidiki lebih lanjut pengaruh berbagai opsi konfigurasi yang ditawarkan oleh VxLang (jika ada, seperti tingkat virtualisasi/obfuscation yang berbeda, fitur anti-tamper) terhadap tingkat kesulitan *reverse engineering* dan *overhead* performa.
- **Analisis Mendalam terhadap Interaksi VxLang dengan Berbagai Jenis Malware/PUA:** Mengingat hasil deteksi VirusTotal yang bervariasi, penelitian lebih lanjut diperlukan untuk memahami faktor-faktor spesifik pada *malware/PUA* (misalnya, bahasa pemrograman, teknik *packing* awal, target API) yang menyebabkan penurunan atau justru peningkatan deteksi setelah virtualisasi VxLang. Ini dapat membantu mengidentifikasi skenario di mana VxLang paling efektif atau paling berisiko terdeteksi.
- **Evaluasi Risiko Deteksi sebagai "Packed/Protected Software":** Pengembang yang menggunakan VxLang untuk proteksi perangkat lunak sah perlu menyadari potensi bahwa aplikasi mereka dapat ditandai sebagai mencurigakan atau "*RiskWare*" oleh beberapa *engine* antivirus hanya karena penggunaan lapisan virtualisasi yang agresif, meskipun tidak ada niat jahat.
- **Pengembangan Teknik Deteksi untuk Kode Tervirtualisasi:** Menyelidiki pengembangan teknik atau *tool* baru yang dapat lebih efektif mendeteksi atau bahkan melakukan *deobfuscation* parsial terhadap kode yang telah dilindungi oleh *virtualizer* seperti VxLang, terutama dari perspektif analisis *malware*.

DAFTAR REFERENSI

[1] Oreans, *Code virtualizer*, https://www.oreans.com/CodeVirtualizer.php, May 2006. Accessed: Nov. 11, 2024.

[2] *Vxlang documentation*, https://vxlang.github.io/, Mar. 2025. Accessed: Mar. 17, 2025.

[3] geeksforgeeks, *Software and its types*, https://www.geeksforgeeks.org/software-and-its-types/, Aug. 2023. Accessed: Nov. 4, 2024.

[4] geeksforgeeks, *Compiling a c program: Behind the scenes*, https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/, Oct. 2024. Accessed: Nov. 10, 2024.

[5] D. Mukesha, *How is c program compiled and executed*, https://medium.com/@danymukesha/here-is-a-flowchart-that-shows-the-steps-that-are-involved-in-compiling-and-running-a-c-program-736f72c501a4, Feb. 2024. Accessed: Nov. 15, 2024.

[6] codeacademy, *Control flow*, https://www.codecademy.com/resources/docs/general/control-flow, Jun. 2023. Accessed: Nov. 27, 2024.

[7] USC Viterbi School of Engineering, *Cs356 unit 5 x86 control flow*, https://ee.usc.edu/~redekopp/cs356/slides/CS356Unit5_x86_Control. Accessed: Nov. 12, 2024.

[8] geeksforgeeks, *Reverse engineering - software engineering*, https://www.geeksforgeeks.org/software-engineering-reverse-engineering/, Jan. 2024. Accessed: Nov. 4, 2024.

[9] Sec-Dudes, *Hands on: Dynamic and static reverse engineering*, https://secdude.de/index.php/2019/08/01/about-dynamic-and-static-reverse-engineering/, Aug. 2019. Accessed: Dec. 18, 2024.

[10] Hex-Rays, *Ida pro*, https://hex-rays.com/ida-pro, May 1991. Accessed: Nov. 22, 2024.

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

[12] D. Ogilvie, *X64dbg*, https://x64dbg.com, May 2014. Accessed: Nov. 11, 2024.

[13] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA: No Starch Press, 2012, ISBN: 978-1593272906.

[14] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley, 2011, ISBN: 978-0764574818.

[15] C.-K. Ko and J. Kinder, "Static disassembly of obfuscated binaries," in *Proceedings of the 2nd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '07)*, San Diego, California, USA: ACM, Jun. 2007, pp. 31–38, ISBN: 978-1-59593-722-9. DOI: 10.1145/1250017.1250024.

[16] 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.

[17] 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.

[18] 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.

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

[20] S. Darwish, S. Guirguis, and M. 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. 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, Mar. 2018. Accessed: Dec. 25, 2024.

[27] Yakov, *Using llvm to obfuscate your code during compilation*, https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation, Jun. 2020. Accessed: Dec. 20, 2024.

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

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

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

[32] Hackcyom, *Hackcyom*, https://www.hackcyom.com/2024/09/vm-obfuscation-overview/, Sep. 2024. Accessed: Nov. 22, 2024.

[33] Qt, *Qt framework*, https://doc.qt.io/qt-6/qt-intro.html, month = may, year = 1995, urldate = 2025-03-17, tag = Qt,

[34] ImGui, *Dear imgui*, https://github.com/ocornut/imgui, Aug. 2014. Accessed: Mar. 17, 2025.

[35] Eric A. Young, Tim J. Hudson, *Openssl*, https://docs.openssl.org/master/man7/ossl-guide-introduction/, Jan. 2015. Accessed: Mar. 17, 2025.

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

LAMPIRAN

KODE SUMBER LENGKAP APLIKASI AUTENTIKASI

Berikut adalah kode sumber lengkap untuk beberapa bagian penting dari aplikasi studi kasus autentikasi yang dibahas pada Bab 4.

```
#include <iostream>
#include <string>
#include <vector> // Contoh include lain jika ada

// Jika menggunakan VxLang, header-nya di-include di sini
#include "vxlang/vxlib.h"

int main(int, char *[]) {
    std::string inputUsername;
    std::string inputPassword;

    std::cout << "Enter username: ";
    std::cin >> inputUsername;
    std::cout << "Enter password: ";
    std::cin >> inputPassword;

    VL_VIRTUALIZATION_BEGIN;

    if (inputUsername.compare("seno") == 0 &&
        inputPassword.compare("rahman") == 0) {
        std::cout << "Authorized!" << std::endl;
    } else {
        std::cout << "Not authorized." << std::endl;
    }

    VL_VIRTUALIZATION_END;

    system("pause"); // Hanya untuk Windows agar console tidak
    langsung tertutup
    return 0;
}
```

Kode 1: Kode Lengkap: Aplikasi Konsol Varian Hardcoded (`console.cpp`)

```
#include <iostream>
#include <string>
#include "cloud.hpp" // Asumsi header ini berisi
send_login_request
```

```
// Jika menggunakan VxLang, header-nya di-include di sini
#include "vxlang/vxlib.h"

int main(int, char *[]) {
    std::string inputUsername;
    std::string inputPassword;

    std::cout << "Enter username: ";
    std::cin >> inputUsername;
    std::cout << "Enter password: ";
    std::cin >> inputPassword;

    bool isAuthenticated = false; // Default value

    VL_VIRTUALIZATION_BEGIN;

    isAuthenticated = send_login_request(inputUsername, inputPassword);

    if (isAuthenticated) {
        std::cout << "Authorized!" << std::endl;
    } else {
        std::cout << "Not authorized." << std::endl;
    }

    VL_VIRTUALIZATION_END;

    system("pause"); // Hanya untuk Windows
    return 0;
}
```

Kode 2: Kode Lengkap: Aplikasi Konsol Varian Cloud (`console_cloud.cpp`)

```
#include <string>
#include <iostream> // Untuk debug, bisa dihapus
#include <curl/curl.h>
#include <nlohmann/json.hpp>
#include "cloud.hpp" // Deklarasi fungsi jika dipisah

#ifdef _WIN32
#include <windows.h> // Diperlukan untuk HWND jika digunakan
#else
using HWND = void*; // Placeholder untuk non-Windows
#endif

// Callback function untuk menulis data yang diterima dari cURL
ke string
// Sumber: https://curl.se/libcurl/c/getinmemory.html
```

```
static size_t WriteCallback(void *contents, size_t size, size_t nmemb,
void *userp) {
    size_t realsize = size * nmemb;
    std::string *mem = (std::string*)userp;
    mem->append((char*)contents, realsize);
    return realsize;
}


bool send_login_request(const std::string &username,
                        const std::string &password, HWND /*window*/ /*=
                        NULL*/) {
    CURL *curl;
    CURLcode res;
    std::string readBuffer; // String untuk menyimpan respons server
    std::string url = "http://localhost:9090/login"; // URL server
    backend

    // Membuat payload JSON
    nlohmann::json payload;
    payload["username"] = username;
    payload["password"] = password;
    std::string jsonStr = payload.dump(); // Konversi JSON ke string

    curl_global_init(CURL_GLOBAL_ALL); // Inisialisasi global libcurl
    curl = curl_easy_init(); // Inisialisasi handle cURL

    if (curl) {
        struct curl_slist *headers = nullptr; // List untuk custom
        headers
        headers = curl_slist_append(headers, "Content-Type:
        application/json"); // Set header Content-Type

        // Set opsi cURL
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str()); // Set URL
        tujuan
        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers); // Set custom
        headers
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, jsonStr.c_str()); // Set
        body request (data JSON)
        curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, jsonStr.length()); //
        Set ukuran body
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback); //
        Set fungsi callback untuk menerima data respons
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer); // Pointer
        ke string buffer untuk menyimpan respons
        curl_easy_setopt(curl, CURLOPT_TIMEOUT, 5L); // Set timeout
        request 5 detik
        curl_easy_setopt(curl, CURLOPT_FAILONERROR, 1L); // Gagal jika
        HTTP status code >= 400
```

```
    // Lakukan request HTTP POST
    res = curl_easy_perform(curl);

    // Cleanup
    curl_easy_cleanup(curl);
    curl_slist_free_all(headers); // Bebaskan memori headers

    // Cek hasil request cURL
    if (res != CURLE_OK) {
        // std::cerr << "curl_easy_perform() failed: " <<
        curl_easy_strerror(res) << std::endl;
        curl_global_cleanup();
        return false; // Gagal mengirim request
    }

    // Cek jika buffer kosong (meskipun CURLE_OK, mungkin
    tidak ada body respons)
    if (readBuffer.empty()) {
       // std::cerr << "Received empty response from server."
       << std::endl;
        curl_global_cleanup();
        return false;
    }

    // Parse respons JSON
    try {
        nlohmann::json response = nlohmann::json::parse(readBuffer);
        curl_global_cleanup();
        // Kembalikan nilai field "success" dari JSON (default
        false jika tidak ada)
        return response.value("success", false);
    } catch (const nlohmann::json::parse_error &e) {
        // std::cerr << "JSON parse error: " << e.what() <<
        std::endl;
        // std::cerr << "Received data: " << readBuffer <<
        std::endl;
        curl_global_cleanup();
        return false; // Gagal parse JSON
    }
     catch (const nlohmann::json::exception &e) {
       // std::cerr << "JSON generic error: " << e.what() <<
       std::endl;
       curl_global_cleanup();
       return false; // Error JSON lainnya
    }
}

curl_global_cleanup();
return false; // Gagal menginisialisasi cURL
```

```
}
```

Kode 3: Kode Lengkap: Implementasi Fungsi `send_login_request` (`cloud.hpp`)

KONTEKS ASSEMBLY ANALISIS BAB 5

Berikut adalah konteks assembly yang lebih lengkap untuk analisis yang disajikan pada Bab 5.

```
; --- Konteks Lebih Lengkap untuk Analisis Statis
Non-Virtualized (app_imgui) ---
; (Asumsi RDX sudah menunjuk ke input username, RCX sudah
menunjuk ke "seno")
; ... (Beberapa instruksi sebelumnya) ...
1400031d6 e8 ...      CALL          VCRUNTIME140.DLL::memcmp ; Cek
Username
1400031db 49 83 fe 04 CMP           R14, 0x4                 ; Cek
panjang username?
1400031df 74 45       JNZ           LAB_140003226            ; Lompat
jika panjang salah
1400031e1 85 c0       TEST          EAX,EAX                  ; Cek hasil
username memcmp
1400031e3 74 41       JNZ           LAB_140003226            ; Lompat
jika username salah

; --- Lanjut ke cek password jika username OK ---
140003201 48 8d 15 ... LEA          RDX,[s_rahman_140110551] ; Muat
alamat "rahman"
140003208 48 89 f9    MOV           RCX,RDI                  ; Muat
alamat input password ke RCX
14000320b e8 ...      CALL          VCRUNTIME140.DLL::memcmp ; Cek
Password
140003210 48 83 fb 06 CMP           RBX, 0x6                 ; Cek
panjang password?
140003214 74 10       JNZ           LAB_140003226            ; Lompat
jika panjang salah
140003216 85 c0       TEST          EAX,EAX                  ; Cek hasil
password memcmp
140003218 74 0c       JNZ           LAB_140003226            ; Lompat
jika password salah

; --- Blok Sukses (jika password benar) ---
; ... (Kode yang dijalankan jika autentikasi berhasil) ...
; ... (Mungkin ada lompatan ke akhir blok auth) ...

LAB_140003226:                                       ; Label Blok
Gagal
140003226 48 8b 0d ... MOV          RCX, qword ptr [DAT_140162878] ;
Setup MessageBox
```

```
14000322d e8 ...       CALL          FUN_140100180           ; Panggil
fungsi internal?
140003232 48 8d 15 ... LEA
RDX,[u_Authentication_Failed_1401104da] ; Teks Error
140003239 4c 8d 05 ... LEA           R8,[u_Login_140110506]      ; Judul
MessageBox
140003240 48 89 c1     MOV           RCX, RAX                ; Handle
window?
140003243 41 b9 10 ... MOV           R9D, 0x10               ; Tipe
MessageBox (MB_ICONERROR)
140003249 ff 15 ...    CALL          qword ptr [->USER32.DLL::MessageBoxW]
; Tampilkan Pesan
; ... (Kelanjutan setelah gagal) ...
```

Kode 4: Konteks Assembly Lengkap: Analisis Statis Non-Virtualized (*app_imgui*)

```
; --- Konteks Lebih Lengkap untuk Analisis Statis Cloud
(console_cloud) ---
; ... (Kode sebelum check hasil auth) ...
LAB_140001754:                                  ; Awal blok check
        140001754 8a 45 bf  MOV      AL, byte ptr [RBP + local_69] ; Ambil
        hasil
        140001757 a8 01     TEST     AL, 0x1                       ; Cek
        hasil
        140001759 75 02     JNZ      LAB_14000175d                 ;
        Lompat jika sukses
        14000175b eb 31     JMP      LAB_14000178e                 ;
        Lompat ke gagal jika tidak sukses

LAB_14000175d:                                  ; Blok Sukses
        14000175d 48 8b 0d .. MOV      RCX, qword ptr
        [->MSVCP140D.DLL::std::cout]
        140001764 48 8d 15 .. LEA      RDX, [s_Authorized!_140037270] ;
        String "Authorized!"
        14000176b e8 d0 0c .. CALL     FUN_140002440                  ;
        Panggil fungsi print?
        140001770 48 89 45 b0 MOV      qword ptr [RBP + local_78], RAX ;
        Simpan return value print?
        140001774 eb 00     JMP      LAB_140001776                  ;
        Lompat ke akhir blok? (Alamat tidak ditampilkan lengkap)

LAB_14000178e:                                  ; Blok Gagal
        14000178e 48 8b 0d .. MOV      RCX, qword ptr
        [->MSVCP140D.DLL::std::cout]
        140001795 48 8d 15 .. LEA      RDX, [s_Not_authorized_14003727c] ;
        String "Not authorized"
        14000179c e8 9f 0c .. CALL     FUN_140002440                  ;
        Panggil fungsi print?
```

```
        1400017a1 48 89 45 a8 MOV      qword ptr [RBP + local_80], RAX ;
        Simpan return value print?
        1400017a5 eb 00     JMP      LAB_1400017a7                ;
        Lompat ke akhir blok? (Alamat tidak ditampilkan lengkap)
; ... (Akhir dari blok pengecekan) ...
```

Kode 5: Konteks Assembly Lengkap: Analisis Statis Non-Virtualized Cloud (*console_cloud*)

```
; --- Konteks Lebih Lengkap untuk Analisis Dinamis
Non-Virtualized (app_qt) ---
; (Asumsi di dalam fungsi handler tombol login)

; Cek Username
LEA rax,qword ptr ds:[<"seno">]      ; Alamat string "seno"
mov qword ptr ss:[rbp+48],rax      ; Simpan alamat ke stack
lea rcx,qword ptr ss:[rbp+A8]      ; Alamat QString input username
dari UI
lea rdx,qword ptr ss:[rbp+48]      ; Alamat "seno" di stack
call <app_qt.bool __cdecl operator==(class QString const &, char const
*const &)> ; Panggil QString::operator==
mov cl,al                          ; Simpan hasil bool ke CL
xor eax,eax
test cl,1                          ; Cek hasil (Zero Flag set jika
false)
mov byte ptr ss:[rbp-41],al        ; Simpan hasil bool ke variabel
stack
jne app_qt.7FF7B0724899            ; Lompat jika TIDAK SAMA (ZF=0) ke
blok selanjutnya (cek pwd)
jmp app_qt.7FF7B07248B7            ; Lompat jika SAMA (ZF=1) ke blok
GAGAL (atau logika lain?)
                                   ; --> Perlu verifikasi ulang
                                   logika JNE/JMP ini di debugger

app_qt.7FF7B0724899: ; Label jika username TIDAK SAMA

; Cek Password (Hanya jika username OK - sesuaikan alurnya)
app_qt.7FF7B07248B7: ; (Asumsi label ini adalah jika username SAMA)
LEA rax,qword ptr ds:[<"rahman">]    ; Alamat string "rahman"
mov qword ptr ss:[rbp+40],rax      ; Simpan alamat ke stack
lea rcx,qword ptr ss:[rbp+90]      ; Alamat QString input password
dari UI
lea rdx,qword ptr ss:[rbp+40]      ; Alamat "rahman" di stack
call <app_qt.bool __cdecl operator==(class QString const &, char const
*const &)> ; Panggil QString::operator==
mov byte ptr ss:[rbp-41],al        ; Simpan hasil bool ke variabel
stack
mov al,byte ptr ss:[rbp-41]
test al,1                          ; Cek hasil (Zero Flag set jika
false)
```

```
jne app_qt.7FF7B07248C3           ; Lompat jika TIDAK SAMA (ZF=0) ke
blok SUKSES?
jmp app_qt.7FF7B0724988           ; Lompat jika SAMA (ZF=1) ke blok
GAGAL?
                                    ; --> Logika JNE/JMP ini juga
                                    perlu diverifikasi

app_qt.7FF7B07248C3: ; (Asumsi Blok Sukses)
; ... (Kode yang dijalankan jika autentikasi berhasil) ...
jmp END_OF_AUTH_CHECK_DYN

# app_qt.7FF7B0724988 (Asumsi Blok Gagal)
lea rcx,qword ptr ss:[rbp+4]      ; 'this' pointer untuk QMessageBox?
mov edx,400                       ; Argumen untuk constructor QFlags?
call <app_qt.public: __cdecl QFlags<...>::QFlags<...>(...)> ; Panggil
constructor?
lea rdx,qword ptr ds:[<"Authentication Failed">] ; Alamat string error
; ... (Panggil static method QMessageBox::warning atau
critical?) ...

END_OF_AUTH_CHECK_DYN:
; ... (Kelanjutan setelah blok autentikasi) ...
```

Kode 6: Konteks Assembly Lengkap: Analisis Dinamis Non-Virtualized (*app_qt*)

```
; --- Konteks Lebih Lengkap: Perbandingan Input/Output ---

; --- Non-Virtualized (console) ---
call <console.public: __cdecl std::basic_string<char, struct
std::char_traits<char>, class std::allocator<char>>::basic_string<char,
struct std::char_traits<char>, class std::allocator<char>>(void)> ;
Constructor string?
mov rcx, qword ptr ds:[<class std::basic_ostream<char, struct
std::char_traits<char>> std::cout>] ; Pointer ke cout
lea rdx, qword ptr ds:[<"Enter username: ">] ; Pointer ke string prompt
call <console.class std::basic_ostream<char, struct std::char_traits<char>>
& __cdecl std::operator<<<struct std::char_traits<char>>(class
std::basic_ostream<char, struct std::char_traits<char>> &, char const *)> ;
Panggil cout << prompt
jmp console.7FF68CD5104D ; Lompatan dalam fungsi?
mov rcx, qword ptr ds:[<class std::basic_istream<char, struct
std::char_traits<char>> std::cin>] ; Pointer ke cin
lea rdx, qword ptr ss:[rbp-28] ; Pointer ke buffer string di stack
call <console.class std::basic_istream<char, struct std::char_traits<char>>
& __cdecl std::operator>><char, struct std::char_traits<char>, class
std::allocator<char>>(class std::basic_istream<char, struct
std::char_traits<char>> &, class std::basic_string<char, struct > ;
Panggil cin >> buffer
```

```
jmp console.7FF68CD5105F ; Lompatan dalam fungsi?
mov rcx, qword ptr ds:[<class std::basic_ostream<char, struct
std::char_traits<char>> std::cout>] ; Pointer ke cout
lea rdx, qword ptr ds:[<"Enter password: ">] ; Pointer ke string prompt
call <console.class std::basic_ostream<char, struct std::char_traits<char>>
& __cdecl std::operator<<<struct std::char_traits<char>>(class
std::basic_ostream<char, struct std::char_traits<char>> &, char const *)> ;
Panggil cout << prompt
; ... (Lanjut baca password) ...


; --- Virtualized (console_vm) ---
00007FF74CD48240 | 40 53                    | push rbx
00007FF74CD48242 | 48 83 EC 20              | sub rsp,20
00007FF74CD48246 | E8 15F00400              | call
console_vm.vxm.7FF74CD97260 ; Panggilan internal VM?
00007FF74CD4824B | ??                       | ??? ; Byte tidak
valid/dikenali
00007FF74CD4824C | 3026                     | xor byte ptr ds:[rsi],ah
00007FF74CD4824E | 8F                       | ??? ; Byte tidak
valid/dikenali
00007FF74CD4824F | EB 03                    | jmp short
console_vm.vxm.7FF74CD48254
00007FF74CD48251 | DDD8                     | fstp st(0)
00007FF74CD48253 | 0000                     | add byte ptr ds:[rax],al
00007FF74CD48255 | 0000                     | add byte ptr ds:[rax],al
; ... (Urutan instruksi yang diobfuskasi berlanjut) ...
```

Kode 7: Konteks Assembly Lengkap: Perbandingan Operasi Input/Output

CONTOH KONFIGURASI BUILD CMAKE DAN SKRIP VIRTUALISASI

Bagian ini menampilkan contoh sederhana bagaimana *target build* yang berbeda (asli dan *intermediate* untuk VxLang) dapat didefinisikan dalam `CMakeLists.txt` dan bagaimana *tool* VxLang dijalankan melalui skrip.

```
# Contoh Penyederhanaan Logika CMakeLists.txt untuk Target 'console'
# (Fungsi add_project_executables pada proyek sebenarnya lebih komprehensif
# dan menangani varian GUI serta cloud secara terpadu)

# Variabel untuk direktori output (diasumsikan sudah di-set)
set(OUTPUT_DIR "${CMAKE_BINARY_DIR}/bin/console/${CMAKE_BUILD_TYPE}")

# Target Asli (Non-VM)
add_executable(console src/console/console.cpp
               ${CMAKE_SOURCE_DIR}/public/resources/console.rc) # Contoh RC file
set_target_properties(console PROPERTIES
  RUNTIME_OUTPUT_DIRECTORY "${OUTPUT_DIR}"
  OUTPUT_NAME "console"
)
# Penentuan subsystem untuk console (jika tidak otomatis)
if(MSVC)
  set_target_properties(console PROPERTIES LINK_FLAGS "/SUBSYSTEM:CONSOLE")
endif()

# Target Intermediate (untuk VxLang)
add_executable(console_vm src/console/console.cpp
               ${CMAKE_SOURCE_DIR}/public/resources/console.rc) # Contoh RC file
target_compile_definitions(console_vm PRIVATE USE_VL_MACRO)
target_link_libraries(console_vm PRIVATE vxlib64) # vxlib64 dari link_directories()
set_target_properties(console_vm PROPERTIES
    RUNTIME_OUTPUT_DIRECTORY "${OUTPUT_DIR}"
    OUTPUT_NAME "console_vm"
)
# Penentuan subsystem untuk console_vm (jika tidak otomatis)
if(MSVC)
  set_target_properties(console_vm PROPERTIES LINK_FLAGS "/SUBSYSTEM:CONSOLE")
endif()

# Catatan: Kode di atas adalah ilustrasi. Proyek menggunakan fungsi
# add_project_executables yang lebih generik di CMakeLists.txt utama.
# Fungsi tersebut menangani berbagai target (console, qt, imgui, performance)
# dan variannya (standar, cloud, _vm, _cloud_vm) serta properti GUI/Console.
# Contoh relevan dari add_project_executables:
if(TARGET_VM_LIBS) # Untuk target _vm dan _cloud_vm
  target_compile_definitions(${TARGET_NAME_VM} PRIVATE USE_VL_MACRO ${TARGET_DEFINITIONS})
  target_link_libraries(${TARGET_NAME_VM} PRIVATE ${TARGET_VM_LIBS})
  # ... set_target_properties ...
else() # Untuk target standar dan cloud standar
  # ... tidak ada USE_VL_MACRO dan tidak ada link ke vxlib64 ...
endif()
```

Kode 8: Ilustrasi Konfigurasi CMake untuk Target Asli dan Intermediate VxLang

```
@echo off
setlocal enabledelayedexpansion
```

```
set build_type=Release
set vxlang_executable=vxlang\vxlang.exe  REM Path ke tool VxLang

rem Contoh untuk target 'console'
set target_name_base=console
set target_intermediate_path=bin\%target_name_base%\%build_type%\%target_name_base%_vm.exe
set target_virtualized_output=bin\%target_name_base%\%build_type%\%target_name_base%_vm.vxm.exe

echo.
echo Virtualizing %target_name_base% ...

if exist "%target_virtualized_output%" (
  echo %target_virtualized_output% already exists. Skipping.
) else (
  if exist "%target_intermediate_path%" (
    echo Processing %target_intermediate_path% with %vxlang_executable% ...
    %vxlang_executable% "%target_intermediate_path%"
    if exist "%target_virtualized_output%" (
      echo Successfully virtualized: %target_virtualized_output%
    ) else (
      echo Failed to create %target_virtualized_output%
    )
  ) else (
    echo Intermediate file %target_intermediate_path% does not exist. Build it first.
  )
)
rem Proses serupa diulang untuk target lain seperti app_qt_vm.exe, Lilith_Client_vm.exe,
dll.
endlocal
```

Kode 9: Contoh Penyederhanaan Skrip `virtualize.bat` untuk Satu Target

PENEMPATAN MAKRO VXLANG PADA KODE BENCHMARK

```
// Potongan relevan dari src/performance/quick_sort.cpp

int partition(std::vector<int> &arr, int low, int high) {
  // VL_VIRTUALIZATION_BEGIN; // Opsional, jika ingin virtualisasi partition
  int pivot = arr[high];
  int i = low - 1;

  for (int j = low; j < high; j++) {
    if (arr[j] <= pivot) {
      i++;
      std::swap(arr[i], arr[j]);
    }
  }
  std::swap(arr[i + 1], arr[high]);
  // VL_VIRTUALIZATION_END; // Opsional
  return i + 1;
}

// Fungsi utama QuickSort yang divirtualisasi
void quickSort(std::vector<int> &arr, int low, int high) {
  VL_VIRTUALIZATION_BEGIN; // Makro VxLang Awal
  if (low < high) {
    // Pemanggilan partition terjadi di dalam blok tervirtualisasi
    int pivot_index = partition(arr, low, high);

    // Panggilan rekursif juga berada di dalam blok virtualisasi
    quickSort(arr, low, pivot_index - 1);
    quickSort(arr, pivot_index + 1, high);
  }
  VL_VIRTUALIZATION_END; // Makro VxLang Akhir
}
```

Kode 10: Penempatan Makro VxLang pada Fungsi `quickSort`

```
// Potongan relevan dari src/performance/encryption.cpp
// Fungsi untuk mengukur waktu enkripsi batch
double measureBatchEncryptionTime(AESCipher &aes,
                                  const std::vector<std::vector<unsigned char>> &blocks,
                                  std::vector<std::vector<unsigned char>> &encrypted) {
    auto start = std::chrono::high_resolution_clock::now();

    VL_VIRTUALIZATION_BEGIN; // Makro VxLang Awal
    encrypted.resize(blocks.size()); // Alokasi di luar loop utama enkripsi
    for (size_t i = 0; i < blocks.size(); ++i) {
        // Panggilan aes.encrypt() terjadi di dalam blok virtualisasi
        encrypted[i] = aes.encrypt(blocks[i]);
    }
    VL_VIRTUALIZATION_END; // Makro VxLang Akhir

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::micro> duration = end - start;
    return duration.count();
}
// Fungsi measureBatchDecryptionTime memiliki struktur serupa
```

Kode 11: Penempatan Makro VxLang pada Loop Utama Benchmark Enkripsi AES

CONTOH PENERAPAN MAKRO VXLANG PADA LILITH RAT

```
// Potongan dari src/Lilith/client/client.cpp
// Fungsi Client::ProcessPacketType(PacketType _PacketType)

// ...
case PacketType::Instruction: {
  std::string msg;
  if (!GetString(msg)) { // GetString sendiri mungkin tidak divirtualisasi
    return false;
  }
  // Bagian inti pemrosesan perintah dan pengiriman respons divirtualisasi
  VL_VIRTUALIZATION_BEGIN;
  // Panggil fungsi General::processCommand yang mungkin juga sebagian/seluruhnya
  divirtualisasi
  std::string response = General::processCommand(msg);
  // SendString sendiri mungkin tidak divirtualisasi agar interaksi jaringan tetap native
  SendString(response, PacketType::Instruction);
  VL_VIRTUALIZATION_END;
  break;
}
// ...
case PacketType::CMDCommand: {
  std::string msg;
  if (!GetString(msg)) return false;

  if (CMD::cmdptr != NULL) {
    // Hanya interaksi dengan objek CMD yang divirtualisasi
    VL_VIRTUALIZATION_BEGIN;
    CMD::cmdptr->writeCMD(msg);
    VL_VIRTUALIZATION_END;
  } else {
    // Jalur error handling mungkin tidak perlu divirtualisasi
    SendString("Initiate a CMD session first.", PacketType::Warning);
  }
  break;
}
// ...
// Contoh lain: di dalam Client::ClientThread()
// Loop utama pemrosesan paket dapat divirtualisasi
/*
void Client::ClientThread() {
  VL_VIRTUALIZATION_BEGIN; // Melingkupi seluruh loop atau bagian kritisnya
  PacketType packetTypeReceived;
  while (true) { // Kondisi loop mungkin lebih kompleks
    if (!clientptr->GetPacketType(packetTypeReceived))
      break;
    if (!clientptr->ProcessPacketType(packetTypeReceived))
      break;
  }
  // ... (logika setelah loop) ...
  VL_VIRTUALIZATION_END;
  // ... (cleanup koneksi) ...
}
*/
```

Kode 12: Contoh Ilustratif Penempatan Makro VxLang pada Fungsi Inti Lilith RAT Client

ISI PARSIAL HEADER VXLANG SDK (`VXLIB.H`)

```
#pragma once

// Bagian ini mengontrol aktivasi makro VxLang berdasarkan definisi USE_VL_MACRO
#ifdef USE_VL_MACRO
  // Jika USE_VL_MACRO didefinisikan (misalnya, saat build _vm target),
  // makro akan memanggil fungsi eksternal dari vxlib64.lib.
  // Fungsi-fungsi ini adalah titik masuk ke runtime VxLang.
  extern "C" void VxVirtualizationBegin();
  extern "C" void VxVirtualizationEnd();

  // Contoh untuk makro lainnya jika ada dan digunakan dalam proyek
  // extern "C" void VxCodeFlatteningBegin();
  // extern "C" void VxCodeFlatteningEnd();
  // extern "C" void VxObfuscationBegin();
  // extern "C" void VxObfuscationEnd();

  // Definisi Makro yang Aktif
  #define VL_VIRTUALIZATION_BEGIN            VxVirtualizationBegin()
  #define VL_VIRTUALIZATION_END              VxVirtualizationEnd()

  // #define VL_CODE_FLATTENING_BEGIN        VxCodeFlatteningBegin()
  // #define VL_CODE_FLATTENING_END          VxCodeFlatteningEnd()
  // #define VL_OBFUSCATION_BEGIN            VxObfuscationBegin()
  // #define VL_OBFUSCATION_END              VxObfuscationEnd()
  // ... dan seterusnya untuk makro lain yang aktif ...

#else
  // Jika USE_VL_MACRO TIDAK didefinisikan (misalnya, saat build target asli),
  // makro-makro ini menjadi kosong (tidak menghasilkan kode apa pun).
  // Ini memungkinkan kode sumber yang sama digunakan untuk build asli dan virtualized.
  #define VL_VIRTUALIZATION_BEGIN
  #define VL_VIRTUALIZATION_END

  // #define VL_CODE_FLATTENING_BEGIN
  // #define VL_CODE_FLATTENING_END
  // #define VL_OBFUSCATION_BEGIN
  // #define VL_OBFUSCATION_END
  // ... dan seterusnya untuk makro lain yang tidak aktif ...

#endif // USE_VL_MACRO
```

Kode 13: Mekanisme Kondisional Makro VxLang dalam `vxlib.h`