UNIVERSITAS INDONESIA

IMPLEMENTASI DAN ANALISIS EFEKTIVITAS *CODE VIRTUALIZATION* DALAM MENINGKATKAN KEAMANAN *SOFTWARE* DENGAN MEMPERSULIT *REVERSE ENGINEERING* MENGGUNAKAN VXLANG

SKRIPSI

SENO PAMUNGKASS RAHMAN
2106731586

FAKULTAS TEKNIK
PROGRAM STUDI TEKNIK KOMPUTER
DEPOK
2025

UNIVERSITAS INDONESIA

IMPLEMENTASI DAN ANALISIS EFEKTIVITAS *CODE VIRTUALIZATION* DALAM MENINGKATKAN KEAMANAN *SOFTWARE* DENGAN MEMPERSULIT *REVERSE ENGINEERING* MENGGUNAKAN VXLANG

SKRIPSI
Diajukan sebagai salah satu syarat untuk memperoleh gelar
Sarjana Teknik

SENO PAMUNGKASS RAHMAN
2106731586

FAKULTAS TEKNIK
PROGRAM STUDI TEKNIK KOMPUTER
DEPOK
JANUARI 2025

HALAMAN PERSETUJUAN

Judul	:	Implementasi dan Analisis Efektivitas *Code Virtualization* dalam Meningkatkan Keamanan *Software* dengan Mempersulit *Reverse Engineering* menggunakan VxLang
Penulis	:	Seno Pamungkas Rahman
NPM	:	2106731586

Laporan Skripsi ini telah diperiksa dan disetujui.

Januari 2025

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

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	:	**Januari 2025**

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh	:	
Nama	:	Seno Pamungkas Rahman
NPM	:	2106731586
Program Studi	:	Teknik Komputer
Judul Skripsi	:	Implementasi dan Analisis Efektivitas *Code Virtualization* dalam Meningkatkan Keamanan *Software* dengan Mempersulit *Reverse Engineering* menggunakan VxLang

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., M.Sc.	( )
Penguji	:	Dr. Ruki Harwahyu, ST. MT. MSc.	( )
Penguji	:	I Gde Dharma Nugraha, S.T., M.T., Ph.D	( )

Ditetapkan di	:	Depok
Tanggal	:	Januari 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, 3 Januari 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 *Code Virtualization* dalam Meningkatkan Keamanan *Software* dengan Mempersulit *Reverse Engineering* menggunakan VxLang

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 : Januari 2025
Yang menyatakan

(Seno Pamungkas Rahman)

ABSTRAK

Nama	:	Seno Pamungkas Rahman
Program Studi	:	Teknik Komputer
Judul	:	Implementasi dan Analisis Efektivitas *Code Virtualization* dalam Meningkatkan Keamanan *Software* dengan Mempersulit *Reverse Engineering* menggunakan VxLang
Pembimbing	:	Dr. Ruki Harwahyu, S.T., M.T., M.Sc.

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 Code Virtualization in Improving Software Security by complicating Reverse Engineering
Counsellor	:	Dr. Ruki Harwahyu, S.T., M.T., M.Sc.

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.

Gambar 2.1: Sistem Operasi Windows & Linux

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 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)

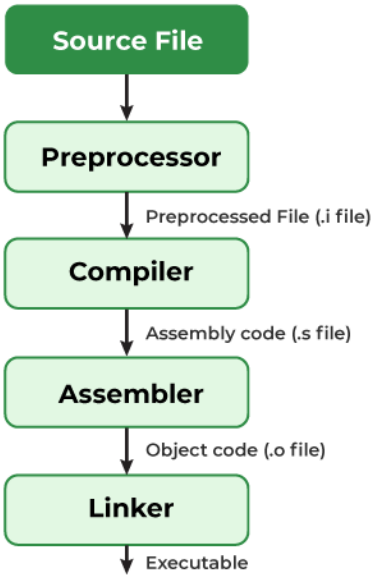
[4]. Proses ini melibatkan beberapa tahapan, yang masing-masing dilakukan oleh program utilitas yang berbeda:

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]

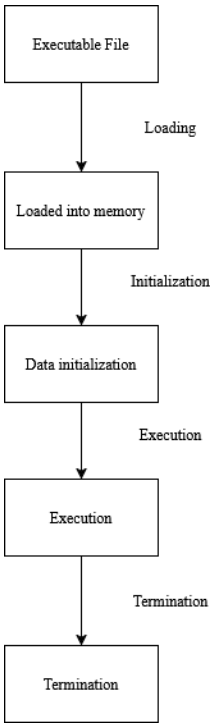
2.1.3.2 Proses Eksekusi

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.

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

- ***if-else statement:*** Menyediakan jalur alternatif jika kondisi *if* tidak terpenuhi.

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

- ***switch statement:*** Memungkinkan percabangan ke banyak kasus.

```
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;
}
```

Loop Instructions:

- ***for loop:*** Mengeksekusi blok kode sejumlah kali tertentu.

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

- ***while loop:*** Mengeksekusi blok kode selama kondisi tertentu terpenuhi.

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

Jump Instructions (Indirect - Function Calls):

- ***Function call:*** Memindahkan kontrol ke fungsi lain dan kembali setelah selesai.

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

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.

```
cmp eax, 10
```

- ***jle, jge, je, jne (conditional jumps):*** Melompat ke label lain jika flags kondisi memenuhi syarat.

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

Loop Instructions (Implementasi dengan Conditional Jumps):

- Menggunakan kombinasi instruksi perbandingan, pengurangan counter, dan conditional jump untuk membuat loop.

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

```
jne loop_start
```

Jump Instructions (Direct):

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

```
jmp target_label
```

Jump Instructions (Indirect - function calls):

- ***call:*** Melompat ke alamat fungsi dan menyimpan alamat return.

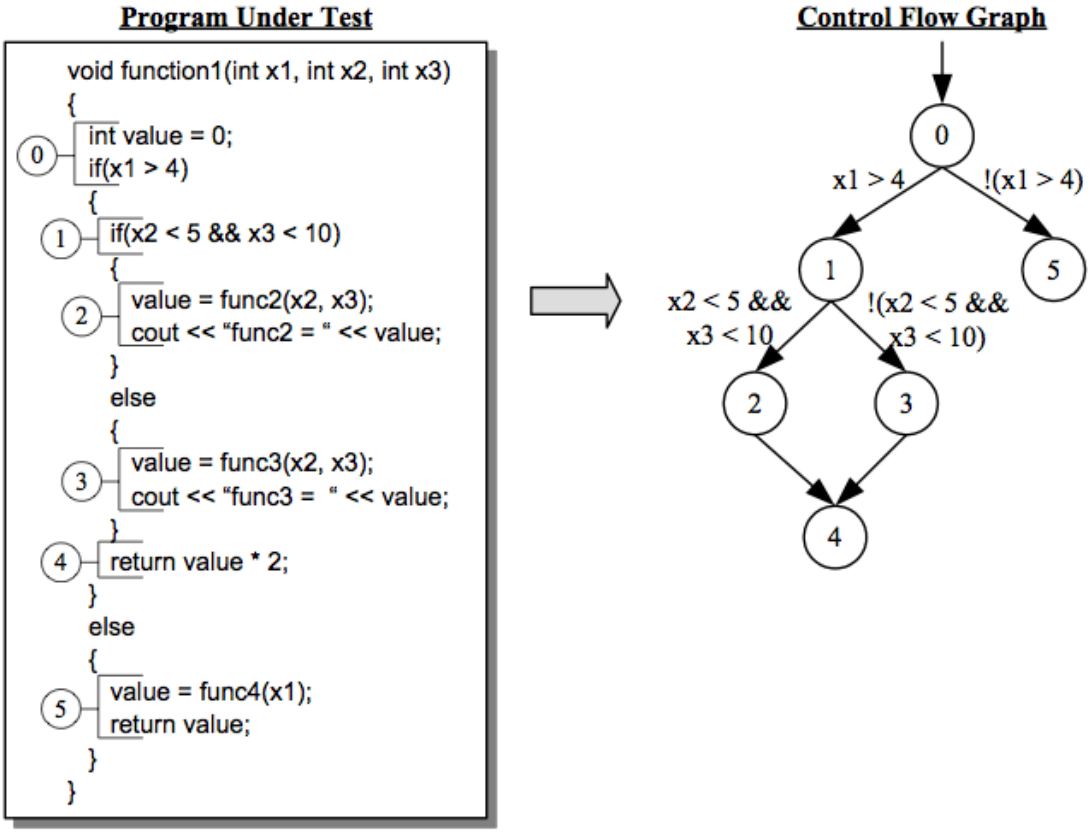
```
call function_address
```

- ***ret:*** Mengembalikan kontrol dari fungsi.

```
ret
```

2.2.2 Control Flow Graph

Graf Alur Kendali (Control Flow Graph atau CFG) adalah representasi grafis dari alur kendali suatu program. CFG memvisualisasikan bagaimana eksekusi program berjalan melalui berbagai blok kode, menggambarkan urutan eksekusi, percabangan, dan perulangan [8]. CFG sangat penting dalam analisis statis kode, reverse engineering, dan optimisasi kode [9]. Dalam CFG, blok dasar (basic blocks) direpresentasikan sebagai simpul (nodes) dan transisi antar blok dasar direpresentasikan sebagai sisi/garis (edges). Contohnya dapat dilihat pada Gambar 2.5.



Gambar 2.5: Contoh Control Flow Graph [10]

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 dokumentasi asli atau kode sumber [11]. 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 [12]. 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 2.6: 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 [12]. 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

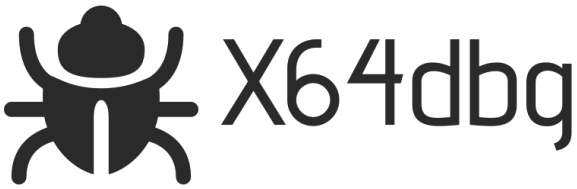
- **IDA Pro (Interactive Disassembler Pro):** Disassembler dan debugger komersial yang sangat populer dan powerful. IDA Pro mendukung berbagai arsitektur prosesor dan sistem operasi, menyediakan antarmuka yang canggih untuk analisis kode, dan mendukung plugin untuk ekstensibilitas [13].

Gambar 2.7: IDA Pro [13]

- **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 [14].

Gambar 2.8: Ghidra [14]

- **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 [15].



Gambar 2.9: x64dbg [15]

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], [20], [21]:*** Mengubah representasi data, misalnya mengenkripsi string atau mengubah nilai numerik dengan operasi matematika. Ini membuat data asli sulit dikenali langsung.
  - ***Instruction Substitution [22], [23]*** Mengganti instruksi yang sederhana dengan instruksi yang lebih kompleks atau setara, tetapi lebih sulit dipahami.
  - ***Mixed boolean arithmetic [24], [25], [26]*** 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 [27]:*** menambahkan kode palsu yang memengaruhi alur kontrol.
  - ***Opaque predicates [28]:*** menyisipkan kode sampah ke dalam pernyataan kondisional.
  - ***Control Flow flattening [29]:*** 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 [30] [31]. Tujuannya adalah untuk mempersulit rekayasa balik bytecode menjadi kode sumber dengan mudah.
Berikut Teknik-teknik obfuscation pada bytecode :

- ***Renaming [30] :*** 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 [30] :*** Meggunakna percabangan yang kompleks, kondisional, dan konstruksi berulang untuk membuat kode yang didekompilasi menjadi non-deterministik dan lebih sulit untuk diikuti

- ***String Encryption [30] :*** 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 [30] :*** 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 [32] :*** 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 [32] :*** Menggunakan inderect calls dan jumps, dan memanipulasi instruksi panggilan dan ret untuk mempersulit mengikuti jalur eksekusi program.

- ***Chunked Control Flow [32] :*** Membagi kode menjadi blok-blok yang sangat kecil dengan instruksi lompatan antar blok untuk mengganggu pembongkaran linear assembly.

- ***Obfuscated Constants [32] :*** Menyembunyikan nilai konstan yang digunakan dalam kode melalui berbagai operasi aritmatika atau logika.

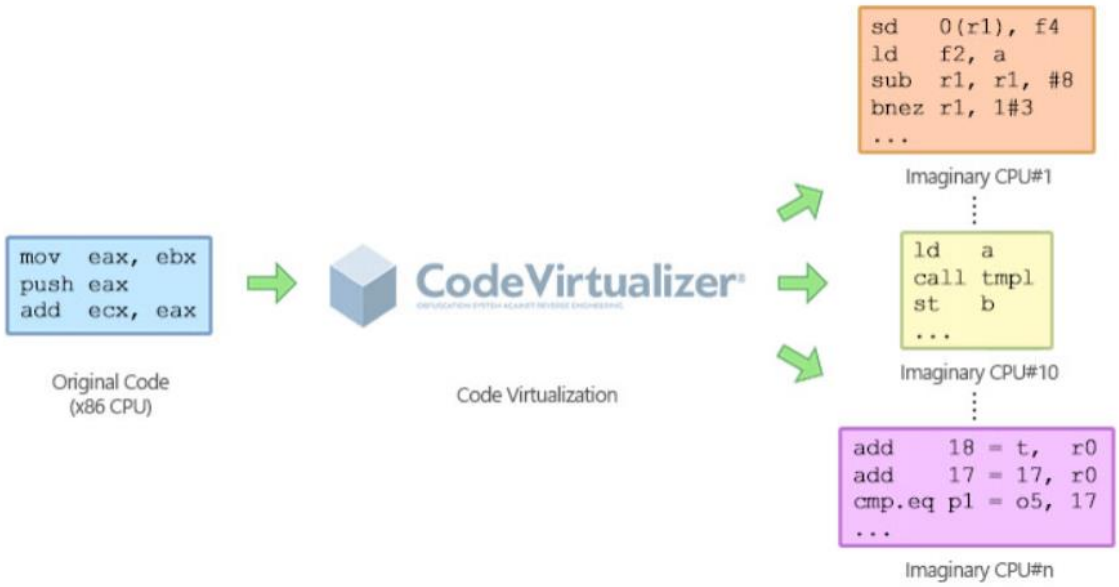
- ***Code Virtualization [1] :*** Menerjemahkan kode mesin menjadi instruksi virtual yang dieksekusi oleh mesin virtual khusus yang tertanam dalam aplikasi.

Gambar 2.10: 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 2.10). Byte-code ini dapat dijalankan secara run-time dengan mesin virtual (i.e. intrepreter) yang tertanam pada aplikasinya [1], [33]. Teknik Code virtualization tidak akan mengembalikan sumber kode aslinya dalam memori sedangkan teknik Code Encryption tetap mengembalikan kodenya aslinya dalam memori saat dilakukan dekripsinya [34].

Set instruksi virtual ini merupakan kunci dari obfuscationnya yang digunakan untuk mapping relasi antara intruksi lokal dan instruksi virtual [33] . 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], [33].



Gambar 2.11: Transformasi ISA x86 menjadi berbagai messin virtual [1]

Arsitektur umum mesin virtual untuk Code Virtualization memiliki komponen-komponen yang mirip dengan design CPU [35], [36].

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).

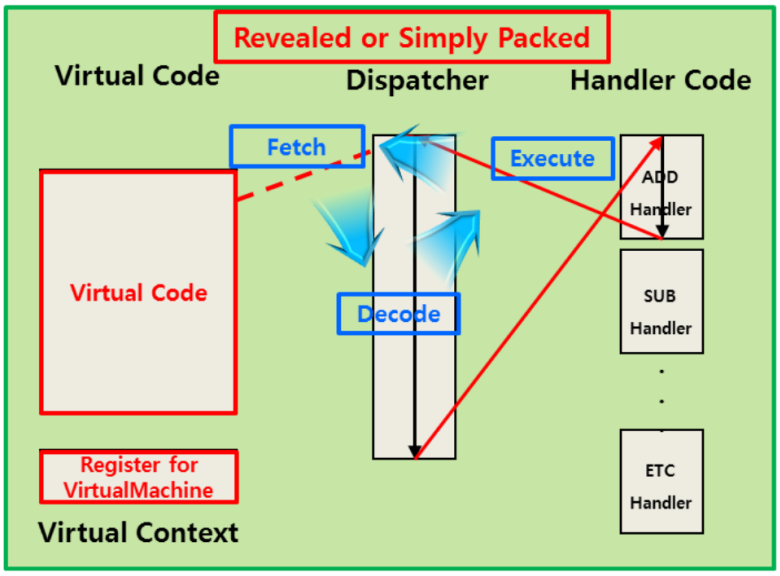
3. ***Decode:*** Perannya adalah untuk mendekode opcode yang diambil dan operan yang sesuai untuk menentukan instruksi ISA mana yang akan dieksekusi.

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

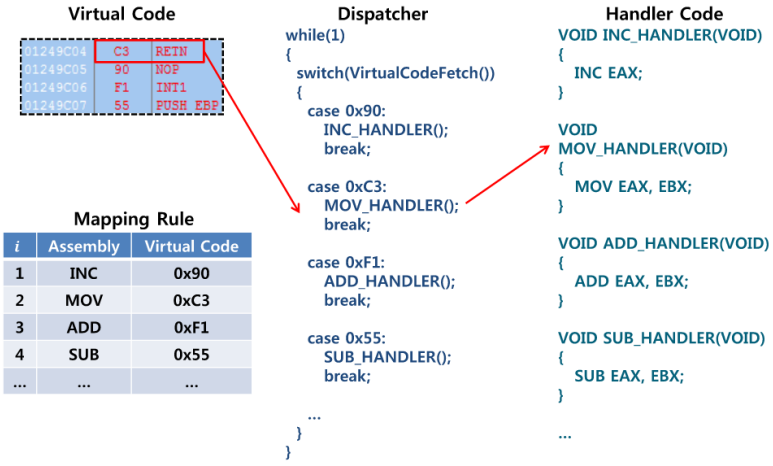
5. ***Handlers:*** Meniru instruksi virtual melalui rangkaian instruksi asli dan memperbarui konteks internal VM, biasanya vpc.

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 2.12 dan contoh code virtualization pada intel assembly dapat dilihat pada Gambar 2.13.



Gambar 2.12: Alur eksekusi Code Virtualization [34]



Gambar 2.13: Transformasi kode asli menjadi kode virtual [34]

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

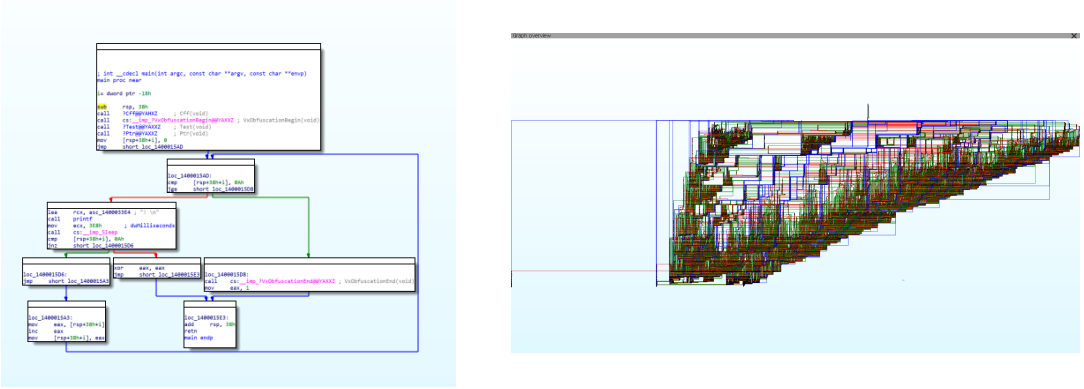
VxLang terdiri dari beberapa komponen utama yang bekerja sama untuk melindungi kode:

- ***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

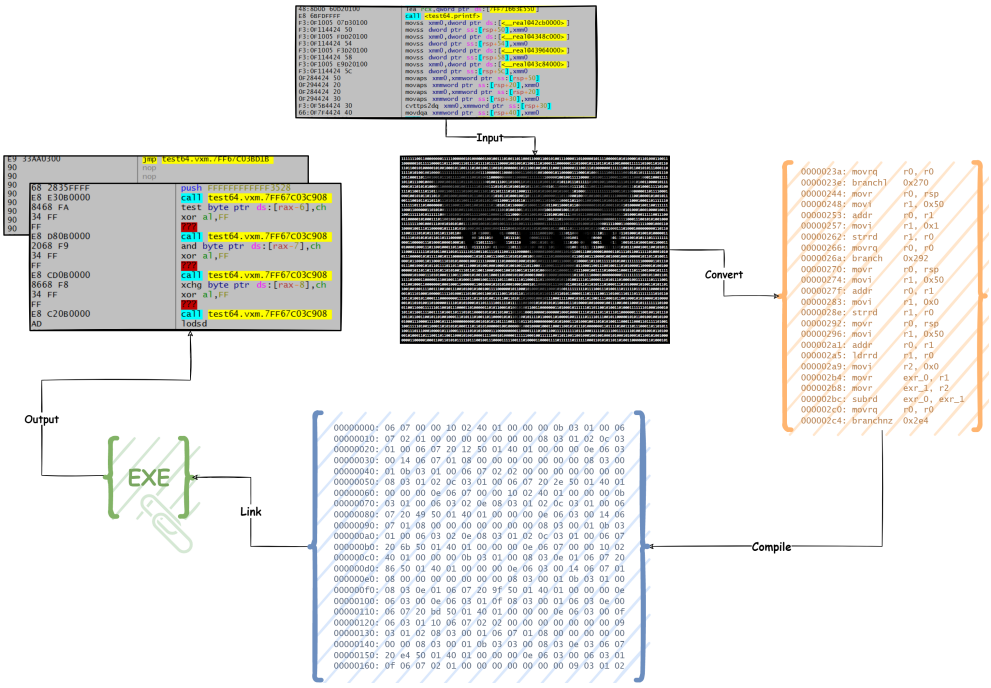
- **Proteksi Biner:** Binary Protector mengambil executable asli dan mengubahnya menjadi format internal VxLang. Proses ini dapat mencakup kompresi untuk mengurangi ukuran file, enkripsi untuk melindungi konten, dan modifikasi struktur file untuk mempersulit analisis statis. Dengan mengubah format file, 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 2.14: 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 2.15: 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.

```
void obfuscation_test(int x) {

  VL_OBFUSCATION_BEGIN;

  printf("Obfuscation Test!")

  VL_OBFUSCATION_END;

}
```

- **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.

```
void code_flattening_test(int x) {

  VL_CODE_FLATTENING_BEGIN;

  printf("Code Flattening Test!")

  VL_CODE_FLATTENING_END;

}
```

- **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.

```
void virtualization_test(int x) {

  VL_VIRTUALIZATION_BEGIN;

  printf("Virtualization Test!")

  VL_VIRTUALIZATION_END;

}
```

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 Konfigurasi Obfuscation VxLang

VxLang menawarkan dua mode penggunaan: Command Line Mode dan Project Mode. Command Line Mode relatif sederhana, sedangkan Project Mode memungkinkan pengembang untuk mengatur proyek mereka melalui file JSON. Dalam penjelasan ini, kita akan fokus pada Project Mode.

Project Mode menggunakan file JSON untuk mengkonfigurasi berbagai aspek dari proses obfuscation dan virtualisasi.

```
{
    "Input" : "",
    "Output" : "",
    "Backup" : false,
    "BaseSectionName" : "",
    "MapFilePath" : "",
    "SignatureMode" : false,
    "Virtualizer" : {
        "EntryPoint" : true
    },
    "Obfuscator" : {
        "EntryPoint" : true,
        "Function": [
            ""
        ]
    },
    "Packer" : {
        "Enable" : true,
        "SymbolData" : false,
        "UseAntiTamper" : {
            "Enable" : true,
            "Message" : "The program exits with abnormal behavior. Please run it again."
        },
        "Extension" : [
        ],
        "RawData" : [
        ]
    }
}
```

Gambar 2.16: VxLang konfigurasi JSON [2]

Berikut adalah penjelasan mengenai pengaturan-pengaturan yang terdapat dalam file JSON konfigurasi:

- **Input :** Menentukan path (jalur) ke file kode sumber yang akan diproses.
- **Output :** Menentukan path ke file keluaran (output) yang telah diobfuskasi.
- **Backup :** Flag (bendera) yang menentukan apakah file kode sumber asli perlu di-backup (dicadangkan).
- **BaseSectionName :** Nama bagian (section) yang menjadi dasar. Jika diatur ke ”.base”, nama bagian VxLang akan ditulis sebagai ”.base0”/”.base1”. Section adalah bagian-bagian dalam file executable yang menyimpan kode, data, dan informasi lainnya.
- **MapFilePath :** Path ke file map. File map berisi informasi mengenai alamat dan simbol dalam program, yang dapat digunakan untuk debugging atau analisis.
- **SignatureMode :** Kemampuan untuk melakukan obfuscation dengan menambahkan tanda tangan (signature) dalam lingkungan di mana SDK tidak dapat ditemukan (misalnya, bahasa native yang berbeda). Signature dapat digunakan untuk memverifikasi integritas kode atau untuk keperluan lisensi.
- **Virtualizer :** Pengaturan untuk fitur virtualisasi.

  - **EntryPoint :** Memvirtualisasi entry point program. Entry point adalah alamat memori tempat eksekusi program dimulai. Memvirtualisasi entry point dapat mempersulit attacker untuk memahami bagaimana program dimulai dan bagaimana alur kontrol program diinisialisasi.

- **Obfuscator :** Pengaturan untuk code flattening atau obfuscation umum.
  - **EntryPoint :** Meng-obfuscasi entry point program.
  - **Function :** Meng-obfuscasi fungsi-fungsi tertentu yang dipilih dalam file map.

- **Packer :** Pengaturan untuk kompresi dan obfuscation biner. Packer adalah program yang mengompresi dan/atau meng-obfuscasi file executable untuk mempersulit reverse engineering.
  - **Enable :** Flag yang menentukan apakah biner perlu dikompresi.

- **SymbolData :** Flag yang menentukan apakah simbol-simbol debugging perlu dicatat (logged). Simbol-simbol debugging berisi informasi yang berguna untuk debugging, tetapi juga dapat membantu reverse engineer dalam memahami kode.

- **UseAntiTemper :** Pengaturan untuk fitur anti-tamper. Anti-tamper adalah teknik untuk mencegah atau mendeteksi modifikasi yang tidak sah pada kode.
  - **Enable :** Nilai yang menentukan apakah fitur anti-tamper diaktifkan.
  - **Message :** Pesan deteksi yang akan ditampilkan ketika deteksi oleh fitur deteksi tertentu.

- **extension :** Daftar modul ekstensi VxLang. Modul ekstensi dapat digunakan untuk menambahkan fungsionalitas tambahan ke VxLang.

- **RawData :** Daftar data yang menyertai modul ekstensi. Daftar ini tersedia dalam modul ekstensi.

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 [37], 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 2.17: Qt Logo [37]

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 [38], 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 [39].
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 2.18: OpenSSL Logo [39]

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** dengan kombinasi metode **kuantitatif** dan **kualitatif**. Pendekatan eksperimental dipilih karena penelitian ini bertujuan untuk mengukur dan membandingkan efek dari suatu perlakuan (intervensi), yaitu penerapan *code virtualization* menggunakan VxLang, terhadap variabel dependen (tingkat kesulitan *reverse engineering* dan performa perangkat lunak).

- Metode **kuantitatif** akan digunakan untuk mengukur dampak virtualisasi pada performa perangkat lunak (waktu eksekusi dan ukuran file) serta mencatat keberhasilan/kegagalan upaya *bypass* autentikasi sebagai indikator.
- Metode **kualitatif** akan digunakan untuk menganalisis dan mendeskripsikan tingkat kesulitan dalam melakukan *reverse engineering* (analisis statis dan dinamis) pada kode sebelum dan sesudah virtualisasi, berdasarkan observasi dan interpretasi peneliti.

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

Setiap objek studi akan dibuat dalam versi asli (non-*virtualized*) dan versi *virtualized* oleh VxLang untuk perbandingan.

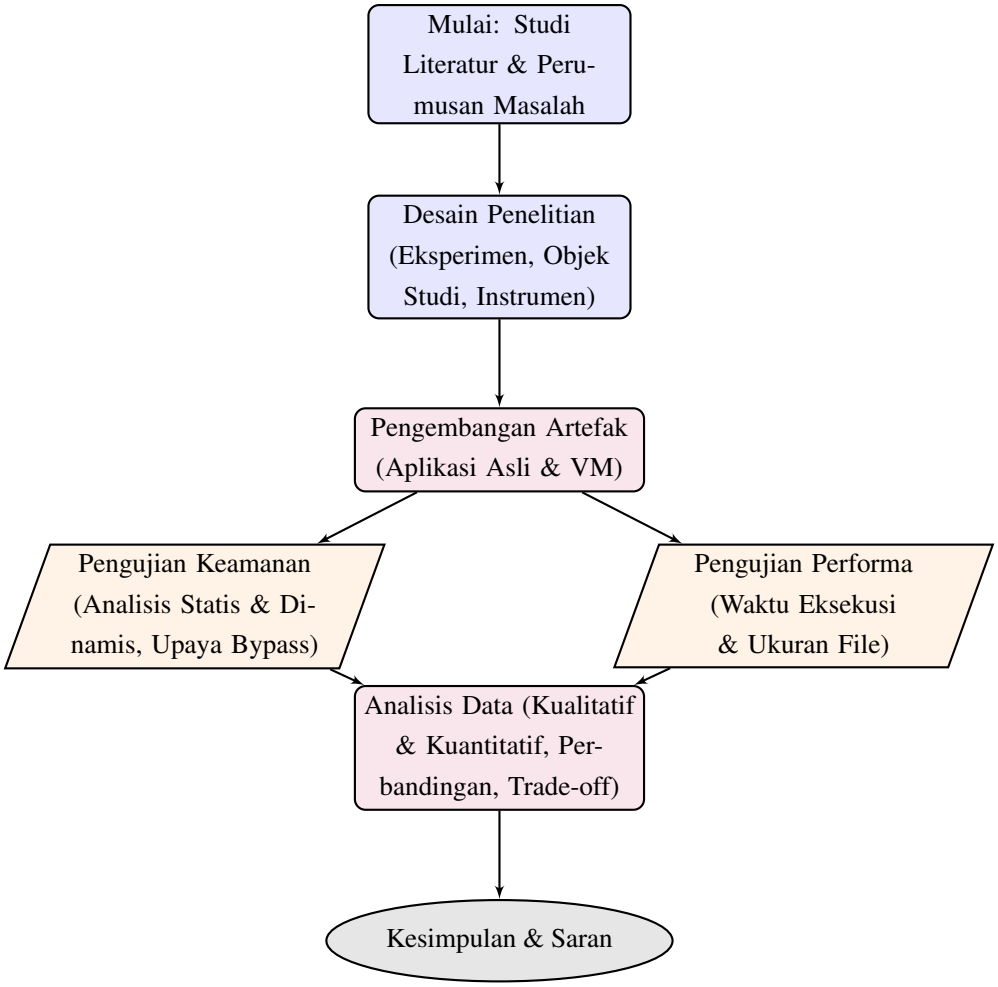
3.4 Instrumen dan Bahan Penelitian

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

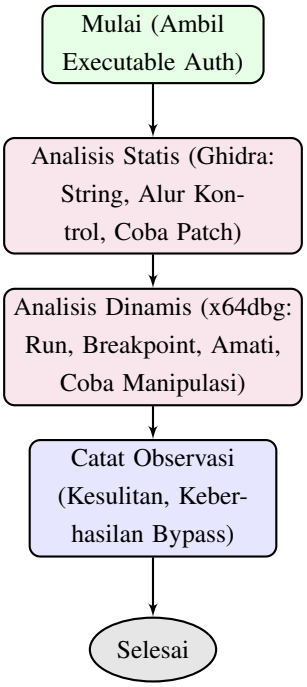
Tahap ini berfokus pada implementasi teknis objek studi. Ini mencakup pengembangan kode sumber untuk aplikasi autentikasi dan *benchmark* performa, integrasi VxLang SDK

ke dalam kode, konfigurasi sistem *build* (CMake) untuk menghasilkan versi asli dan versi *intermediate* (dengan penanda VxLang), serta proses kompilasi. Versi *intermediate* kemudian diproses menggunakan *tool* eksternal VxLang untuk menghasilkan *executable* akhir yang tervirtualisasi. 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 Observasi:** Mencatat temuan kualitatif mengenai tingkat kesulitan pada setiap langkah analisis (pencarian *string*, pemahaman alur, identifikasi logika) dan mencatat keberhasilan atau kegagalan setiap upaya *bypass* (statis dan dinamis).



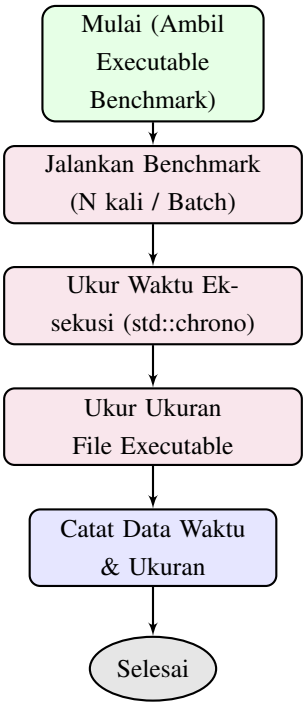
Gambar 3.2: Diagram Alur Prosedur Pengujian Keamanan Autentikasi.

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*.

3.6 Teknik Analisis Data

Data yang telah dikumpulkan akan dianalisis menggunakan teknik berikut:

- **Data Kualitatif (Keamanan):** Analisis deskriptif dan interpretatif berdasarkan catatan observasi untuk membandingkan tingkat kesulitan *reverse engineering* antara kelompok kontrol dan eksperimen.
- **Data Kuantitatif (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:

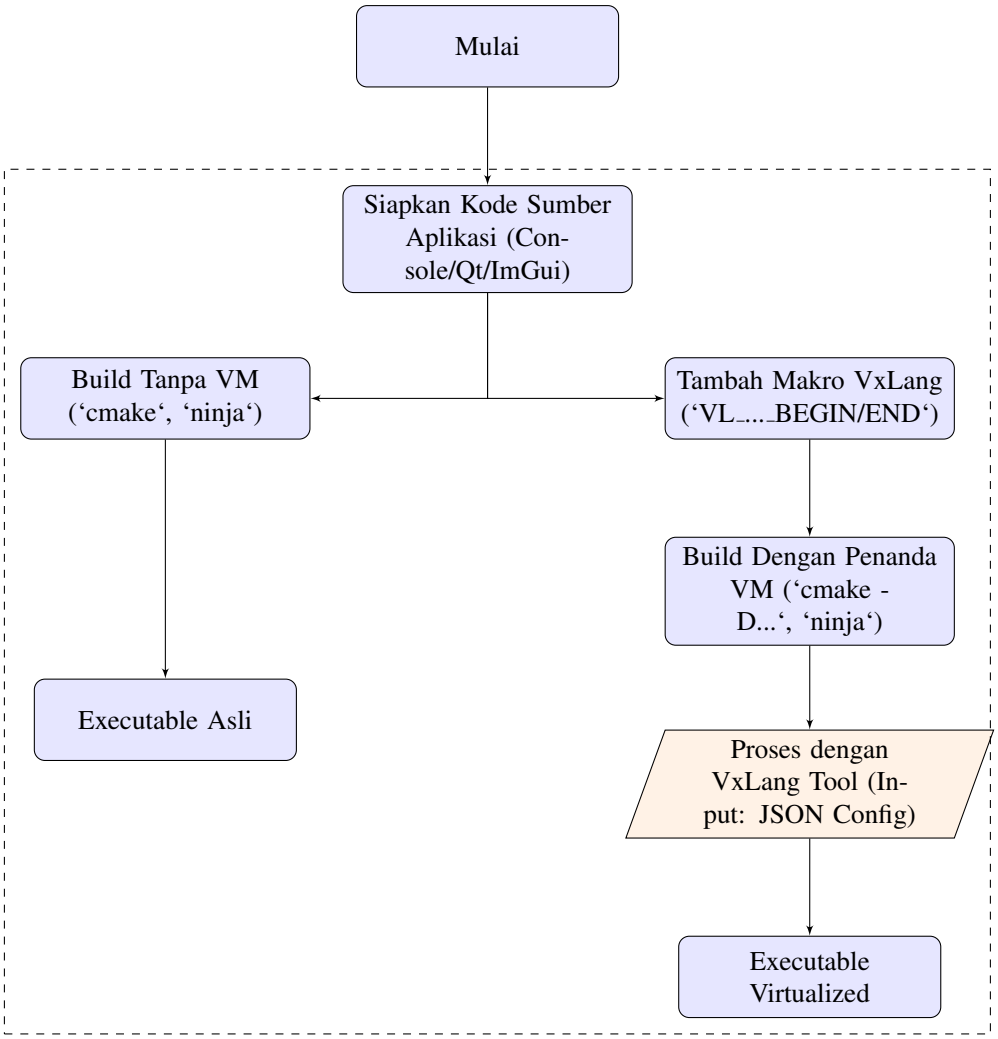
- **Sistem Operasi:** Seluruh proses pengembangan, kompilasi, dan eksekusi pengujian dilakukan pada **Microsoft Windows 11 (64-bit)**. Pemilihan Windows sebagai platform utama didasarkan pada target *executable* PE (Portable Executable) yang umum didukung oleh *tool* proteksi perangkat lunak seperti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

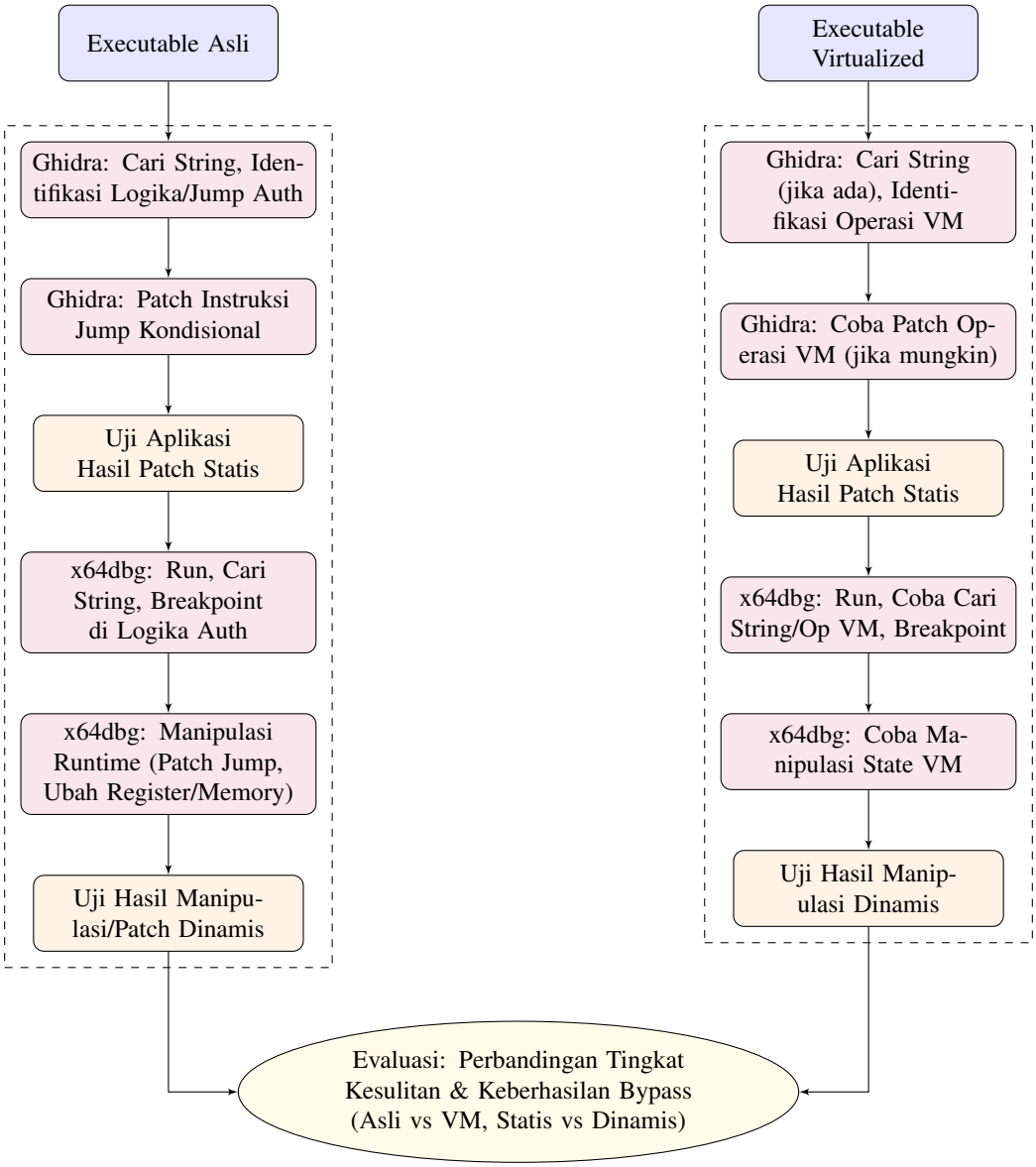
4.2 Implementasi Pengujian Autentikasi

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. 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.



Gambar 4.2: Diagram Alur Analisis Upaya Bypass 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()`.
    - **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`.
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 perbandingan *hardcoded* dalam *slot* `on_button_login_clicked()` menggunakan `QLineEdit::text()`. Menampilkan hasil via `QMessageBox`.
    - **Varian Cloud (`src/app_qt/src/todo_auth_cloud.cpp`):** *Slot* memanggil `send_login_request` dan menampilkan `QMessageBox` berdasarkan respons.
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`.

4.2.2 Integrasi VxLang pada Aplikasi Autentikasi

Integrasi VxLang dilakukan secara selektif pada logika autentikasi:

1. **Penyertaan Header:** `#include "vxlang/vxlib.h"` ditambahkan pada file `.cpp` yang relevan.
2. **Penautan Library:** `vxlib64.lib` ditautkan ke target via `target_link_libraries` di `CMakeLists.txt`.

3. **Penempatan Makro Penanda:** `VL_VIRTUALIZATION_BEGIN` dan `VL_VIRTUALIZATION_END` membungkus blok kode autentikasi. Contoh pada `src/console/console.cpp`:

```
int main(int, char *[]) {
  // ... input username/password ...

  VL_VIRTUALIZATION_BEGIN; // <-- Makro awal

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

  VL_VIRTUALIZATION_END; // <-- Makro akhir

  system("pause");
  return 0;
}
```

   Pada varian *cloud*, makro membungkus pemanggilan `send_login_request` dan penanganan responsnya.

4. **Manajemen Build via CMake:** Opsi CMake `USE_VL_MACRO` mengontrol aktivasi makro.

   - Jika aktif (`-DUSE_VL_MACRO=1`): Definisi `USE_VL_MACRO` ditambahkan, mengaktifkan fungsi makro di `vxlib.h`. Nama output diubah menjadi `*_vm.exe`.
   - Jika tidak aktif: Makro menjadi kosong. Nama output standar.

4.2.3 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

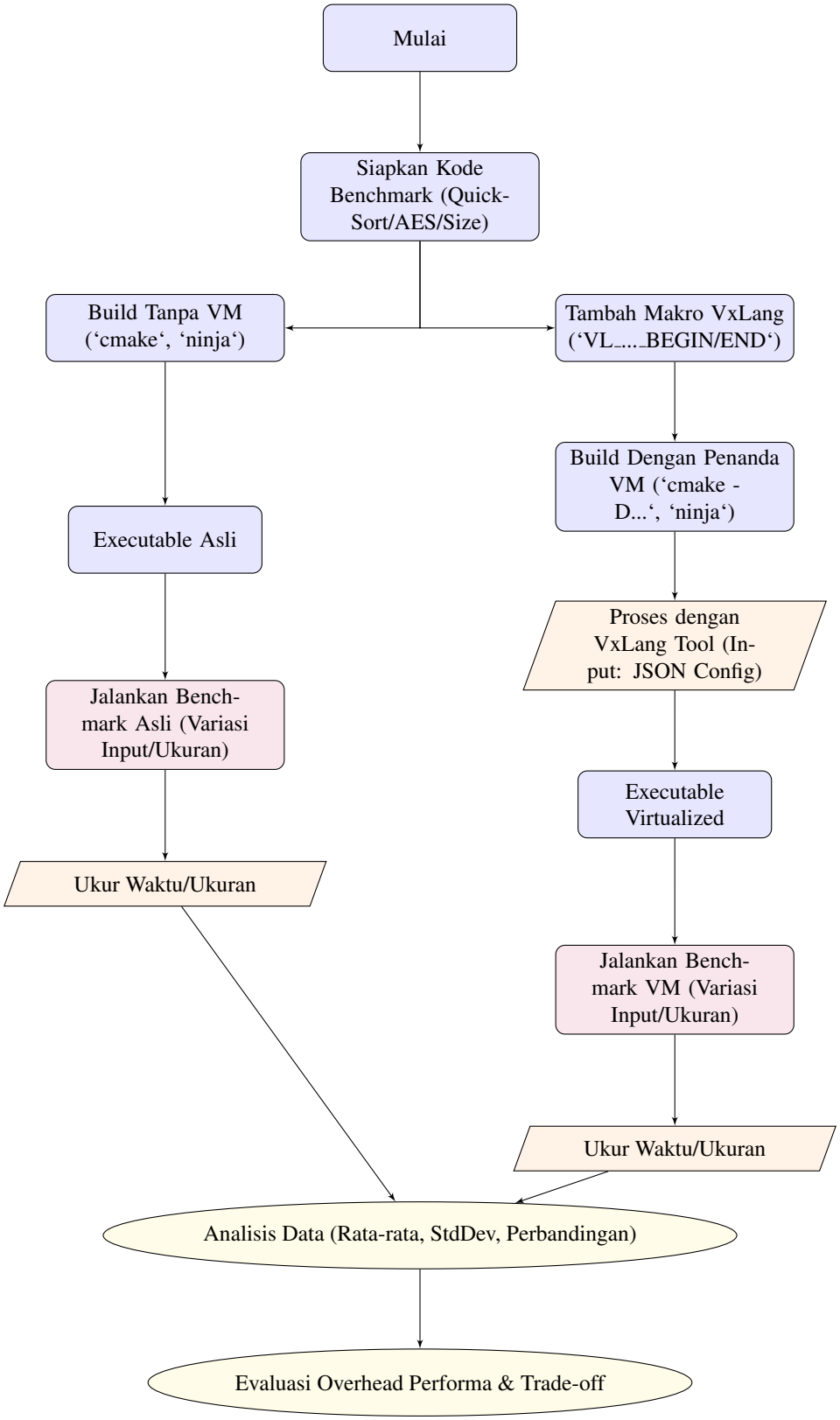
Pengujian ini mengukur dampak kuantitatif VxLang pada kecepatan eksekusi dan ukuran *executable*. Diagram alur untuk persiapan dan pelaksanaan pengujian performa disajikan pada Gambar 4.3.

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`.

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`.



Gambar 4.3: Diagram Alur Persiapan dan Pengujian Performa.

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 5.3.
- **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.

4.4 Proses Kompilasi dan Virtualisasi

Alur kerja untuk menghasilkan *executable* asli dan yang tervirtualisasi adalah:

1. **Konfigurasi CMake:** Menjalankan `cmake -G Ninja -B build [-DUSE_VL_MACRO=1] ...` Opsi `-DUSE_VL_MACRO=1` ditambahkan jika membangun untuk virtualisasi.
2. **Kompilasi Awal:** Menjalankan `ninja -C build`. Menghasilkan `*.exe` (jika `USE_VL_MACRO` nonaktif) atau `*_vm.exe` (jika `USE_VL_MACRO` aktif, berisi penanda).
3. **Proses Virtualisasi Eksternal:** *Executable* `*_vm.exe` dari langkah 2 diproses oleh **tool eksternal VxLang** (tidak ada dalam repositori ini). Tool ini, dikonfigurasi via file JSON, membaca penanda, menerjemahkan kode menjadi *bytecode*, dan menyematkan *interpreter* VM ke *executable* output.
4. **Hasil Akhir:** Direktori `bin` berisi *executable* asli (`*.exe`) dan *executable* yang telah divirtualisasi (`*_vm.exe`), siap untuk dianalisis.

Implementasi yang sistematis ini memastikan bahwa artefak yang dihasilkan cocok untuk perbandingan dan analisis yang valid sesuai dengan tujuan penelitian.

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:

```
1400032cc 48 8d 15 ... LEA          RDX,[DAT_14011054c]       = 73h  s
1400031d3 48 89 d9     MOV          RCX,RBX
1400031d6 e8 ...       CALL         VCRUNTIME140.DLL::memcmp
1400031db 49 83 fe 04  CMP          R14,0x4
1400031df 74 45        JNZ          LAB_140003226
1400031e1 85 c0        TEST         EAX,EAX
1400031e3 74 41        JNZ          LAB_140003226

140003201 48 8d 15 ... LEA          RDX,[s_rahman_140110551]   =
↪  "rahman"
140003208 48 89 f9     MOV          RCX,RDI
```

```
14000320b e8 ...       CALL          VCRUNTIME140.DLL::memcmp
140003210 48 83 fb 06  CMP           RBX,0x6
140003214 74 10        JNZ           LAB_140003226
140003216 85 c0        TEST          EAX,EAX
140003218 74 0c        JNZ           LAB_140003226

LAB_140003226:
140003226 48 8b 0d ... MOV           RCX,qword ptr [DAT_140162878]
14000322d e8 ...       CALL          FUN_140100180
140003232 48 8d 15 ... LEA
↪  RDX,[u_Authentication_Failed_1401104da] = u"Authentication
↪  Failed"
140003239 4c 8d 05 ... LEA           R8,[u_Login_140110506]      =
↪  u"Login"
140003240 48 89 c1     MOV           RCX,RAX
140003243 41 b9 10 00 00 00 MOV           R9D,0x10
140003249 ff 15 ...    CALL          qword ptr
↪  [->USER32.DLL::MessageBoxW]
```

Dalam konteks kode di atas, kita dapat melihat bahwa program memuat string ”rahman” (pada alamat ‘140003201‘) dan kemungkinan string lain (pada alamat ‘1400032cc‘) ke dalam register. Kemudian, fungsi ‘memcmp‘ dipanggil, yang umumnya digunakan untuk membandingkan blok memori. Setelah pemanggilan ‘memcmp‘, terdapat operasi ‘TEST EAX,EAX‘ yang diikuti oleh instruksi ‘JNZ LAB_140003226‘. Instruksi ‘JNZ‘ (Jump if Not Zero) akan melompat ke label ‘LAB_140003226‘ jika hasil dari ‘memcmp‘ tidak nol, yang dalam konteks perbandingan string sering kali menandakan bahwa string yang dibandingkan tidak sama. Pada label ‘LAB_140003226‘, kita melihat string ”Authentication Failed” dimuat dan ditampilkan melalui fungsi ‘MessageBoxW‘.

Berdasarkan analisis ini, dapat disimpulkan bahwa blok kode antara alamat ‘140003201‘ dan ‘140003218‘ kemungkinan besar merupakan bagian dari logika autentikasi yang membandingkan *input* pengguna dengan nilai yang diharapkan (”rahman” dan string lainnya). Jika perbandingan gagal (hasil ‘memcmp‘ tidak nol), program akan melompat ke ‘LAB_140003226‘ dan menampilkan pesan ”Authentication Failed”.

Untuk memvalidasi potensi celah keamanan, instruksi ‘JNZ LAB_140003226‘ pada alamat ‘140003216‘ 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.

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:

```
LAB_140001754                                   XREF[1]:     140001752(j)
        140001754 8a 45 bf                      MOV          AL,byte ptr
        ↪  [RBP + local_69]
        140001757 a8 01                         TEST         AL,0x1
        140001759 75 02                         JNZ          LAB_14000175d
        14000175b eb 31                         JMP          LAB_14000178e


                                LAB_14000175d
                                ↪  XREF[1]:     140001759(j)
        14000175d 48 8b 0d                      MOV          RCX,qword ptr
        ↪  [->MSVCP140D.DLL::std::cout]          = 0003ec5e
                                    44 cd 03 00
        140001764 48 8d 15                      LEA
        ↪  RDX,[s_Authorized!_140037270]                     =
        ↪  "Authorized!"
```

```
                                05 5b 03 00
14000176b e8 d0 0c 00 00              CALL
↪  FUN_140002440
↪  longlong * FUN_140002440(longlon
140001770 48 89 45 b0                 MOV          qword ptr [RBP
↪  + local_78],RAX
140001774 eb 00                       JMP          LAB_140001776



                          LAB_14000178e
                          ↪  XREF[1]:     14000175b(j)
14000178e 48 8b 0d                    MOV          RCX,qword ptr
↪  [->MSVCP140D.DLL::std::cout]          = 0003ec5e
                                13 cd 03 00
140001795 48 8d 15                    LEA
↪  RDX,[s_Not_authorized_14003727c]                  = "Not
↪  authorized"
                               e0 5a 03 00
14000179c e8 9f 0c 00 00              CALL
↪  FUN_140002440
↪  longlong * FUN_140002440(longlon
1400017a1 48 89 45 a8                 MOV          qword ptr
↪  [RBP + local_80],RAX
1400017a5 eb 00                       JMP          LAB_1400017a7
```

Pada kode di atas, label ‘LAB_140001754‘ kemungkinan merupakan awal dari blok logika autentikasi. Instruksi ‘TEST AL, 0x1‘ pada alamat ‘140001757‘ memeriksa suatu kondisi (kemungkinan hasil dari komunikasi dengan server). Jika kondisi ini tidak terpenuhi (hasil test tidak nol), program akan melompat ke ‘LAB_14000175d‘ dan mencetak ”Authorized!”. Jika kondisi terpenuhi (hasil test adalah nol), program akan melompat ke ‘LAB_14000178e‘ dan mencetak ”Not authorized”.

Sama seperti pada kasus *hard-coded* sebelumnya, instruksi ‘JNZ LAB_14000175d‘ pada alamat ‘140001759‘ 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 perlindun-

gan 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. Beberapa observasi signifikan yang ditemukan adalah:

- **Hilangnya String yang Terdefinisi:** *String-string* yang terkait dengan autentikasi, seperti ”Authentication Failed”, tidak ditemukan secara eksplisit dalam bagian ‘.data‘ dari *executable*. Hal ini menyulitkan identifikasi awal dari lokasi kode yang relevan dengan autentikasi.
- **Pengurangan Jumlah Data yang Terdefinisi:** Jumlah data yang terdefinisi dalam *executable* virtualized berkurang secara drastis (dari sekitar 900 menjadi 130 pada salah satu kasus). Ini mengindikasikan bahwa banyak data dan string mungkin telah dienkripsi atau disembunyikan melalui teknik virtualisasi.
- **Operasi yang Tidak Diketahui:** Banyak operasi *assembly* yang tidak dikenali atau dikategorikan sebagai ’?’ oleh *disassembler* Ghidra. Hal ini menyulitkan pemahaman alur kontrol program dan fungsi-fungsi yang terlibat dalam proses autentikasi.
- **Abstraksi Logika Autentikasi:** Mekanisme virtualisasi VxLang tampaknya telah mengabstraksi logika autentikasi yang sebenarnya. Operasi perbandingan *string* dan instruksi kondisional yang jelas terlihat pada aplikasi non-virtualized tidak lagi dapat diidentifikasi dengan mudah.

Ketidakhadiran *string* autentikasi yang jelas dan banyaknya operasi yang tidak diketahui menyiratkan bahwa VxLang menggunakan teknik virtualisasi kode yang efektif dalam menyembunyikan implementasi internal program. Hal ini mempersulit analisis statis tradisional yang mengandalkan identifikasi *string* dan pola *assembly* yang umum.

Akibatnya, identifikasi dan manipulasi logika autentikasi melalui analisis statis menjadi sangat sulit pada aplikasi yang divirtualisasi. Meskipun mungkin ada cara lain untuk menganalisis kode yang divirtualisasi, pendekatan yang bergantung pada pencarian *string* dan *patching* instruksi kondisional secara langsung menjadi tidak efektif dalam kasus ini.

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:

```
LEA rax,qword ptr ds:[<"seno">]
mov qword ptr ss:[rbp+48],rax
lea rcx,qword ptr ss:[rbp+A8]
lea rdx,qword ptr ss:[rbp+48]
call <app_qt.bool __cdecl operator==(class QString const &, char
 ↪  const *const &)>
mov cl,al
xor eax,eax
test cl,1
mov byte ptr ss:[rbp-41],al
jne app_qt.7FF7B0724899
jmp app_qt.7FF7B07248B7

LEA rax,qword ptr ds:[<"rahman">]
mov qword ptr ss:[rbp+40],rax
lea rcx,qword ptr ss:[rbp+90]
lea rdx,qword ptr ss:[rbp+40]
call <app_qt.bool __cdecl operator==(class QString const &, char
 ↪  const *const &)>
mov byte ptr ss:[rbp-41],al
mov al,byte ptr ss:[rbp-41]
test al,1
jne app_qt.7FF7B07248C3
jmp app_qt.7FF7B0724988

# app_qt.7FF7B0724988
lea rcx,qword ptr ss:[rbp+4]
mov edx,400
```

```
call <app_qt.public: __cdecl QFlags<enum
↪  QMessageBox::StandardButton>::QFlags<enum
↪  QMessageBox::StandardButton>(enum QMessageBox::StandardButton)>
lea rdx,qword ptr ds:[<"Authentication Failed">]
```

Kode di atas menunjukkan bahwa program membandingkan *input* dengan string '"seno"' dan '"rahman"' menggunakan operator '=='. Instruksi 'jne' (Jump if Not Equal) pada alamat 'app_qt.7FF7B0724899' dan 'app_qt.7FF7B07248C3' menentukan alur eksekusi berdasarkan hasil perbandingan. Jika *input* tidak sesuai dengan username atau password yang diharapkan, program akan melompat ke alamat yang pada akhirnya menampilkan pesan "Authentication Failed".

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 x64dbg menunjukkan tantangan yang serupa dengan analisis statis. Ketika mencari string seperti "Authentication Failed", *debugger* sering kali tidak dapat menemukannya dalam memori proses. Hal ini mengindikasikan bahwa string tersebut mungkin tidak disimpan dalam format yang jelas atau mungkin dienkripsi dan didekripsi hanya pada saat digunakan.

Selain itu, x64dbg juga menampilkan banyak operasi dengan kategori ??? yang tidak dapat di-*disassemble*. Hal ini mempersulit pemahaman alur eksekusi dan fungsi-fungsi yang terlibat dalam proses autentikasi.

Lebih lanjut, melacak alur eksekusi saat memasukkan username dan password menjadi sangat sulit. Berikut adalah perbandingan *section* kode saat menerima *input* pada aplikasi konsol non-virtualized dan virtualized:

Non-Virtualized:

```
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)>
mov rcx,qword ptr ds:[<class std::basic_ostream<char, struct
↪  std::char_traits<char>> std::cout>]
lea rdx,qword ptr ds:[<"Enter username: ">]
```

```
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 *)>
jmp console.7FF68CD5104D
mov rcx,qword ptr ds:[<class std::basic_istream<char, struct
↪ std::char_traits<char>> std::cin>]
lea rdx,qword ptr ss:[rbp-28]
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 >
jmp console.7FF68CD5105F
mov rcx,qword ptr ds:[<class std::basic_ostream<char, struct
↪ std::char_traits<char>> std::cout>]
lea rdx,qword ptr ds:[<"Enter password: ">]
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 *)>
```

Virtualized:

```
00007FF74CD48244 | 3F                    | ???
00007FF74CD48245 | E9 1EEF0400            | jmp
↪ console_vm.vxm.7FF74CD97168
00007FF74CD4824A | DBDD                  | fcmovnu st(0),st(5)
00007FF74CD4824C | 3026                  | xor byte ptr ds:[rsi],ah
00007FF74CD4824E | 8F                    | ???
00007FF74CD4824F | EB 03                 | jmp
↪ console_vm.vxm.7FF74CD48254
00007FF74CD48251 | DDD8                  | fstp st(0)
```

Pada kode yang telah divirtualisasi, terlihat banyak operasi yang dikategorikan sebagai ???. Lebih lanjut, dalam beberapa kasus, register RIP (*Instruction Pointer*) tampak ”*stuck*” pada alamat tertentu di dalam *debugger* saat mencoba memasukkan username dan password. Hal ini sangat menyulitkan untuk melacak alur eksekusi dan memahami apa yang sebenarnya terjadi di balik layar. Perilaku ini mengindikasikan bahwa VxLang menggunakan teknik virtualisasi yang kompleks, yang secara signifikan menghambat analisis dinamis menggunakan metode tradisional seperti mencari string dan memantau alur eksekusi secara langsung.

5.2 Analisis Performa *Overhead* VxLang

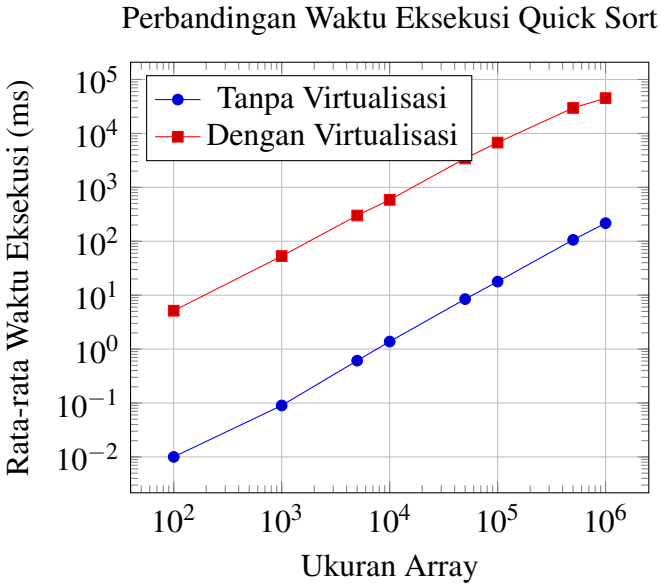
Hasil dari percobaan performa overhead dan perubahan ukuran file setelah penerapan virtualisasi kode menggunakan VxLang. Percobaan ini dilakukan pada algoritma Quick Sort dan enkripsi AES-CBC-256.

5.2.1 Hasil Pengujian Performa *Quick Sort*

Tabel 5.1 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 5.1: Hasil Pengujian Waktu Eksekusi Quick Sort (ms)

Ukuran Array	**Tanpa Virtualisasi**		**Virtualisasi**	
	Rata-rata Waktu (ms)	**Standar Deviasi (ms)**	**Rata-rata Waktu (ms)**	**Standar Deviasi (ms)**
100	0.01	0.00	5.15	0.42
1000	0.09	0.00	53.08	5.52
5000	0.61	0.09	300.64	26.13
10000	1.38	0.22	585.70	79.88
50000	8.45	0.73	3,434.32	592.92
100000	17.86	1.42	6,771.09	553.09
500000	106.16	5.61	29,698.73	3,518.63
1000000	216.59	9.49	45,186.90	6,198.38



Gambar 5.1: Perbandingan Waktu Eksekusi Algoritma Quick Sort antara Versi Tanpa dan Dengan Virtualisasi VxLang.

Berdasarkan Tabel 5.1 ,terlihat adanya peningkatan waktu eksekusi yang signifikan pada algoritma Quick Sort setelah divirtualisasi menggunakan VxLang. Peningkatan ini terlihat semakin besar seiring dengan bertambahnya ukuran array. Sebagai contoh, untuk array berukuran 100, waktu eksekusi rata-rata meningkat dari 0.01 ms menjadi 5.15 ms, yang menunjukkan overhead sekitar 51400%. Untuk array yang lebih besar seperti 1.000.000 elemen, waktu eksekusi meningkat dari 216.59 ms menjadi 45186.90 ms, dengan overhead sekitar 20860%. Peningkatan standar deviasi juga menunjukkan bahwa waktu eksekusi menjadi lebih bervariasi setelah virtualisasi. Hal ini mengindikasikan adanya overhead yang substansial 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 5.2 menyajikan hasil benchmarking enkripsi AES-CBC-256 dengan 1.000.000 blok data, di mana setiap blok berukuran 1024 bytes, sebelum dan sesudah virtualisasi menggunakan VxLang.

Tabel 5.2: Hasil Pengujian Performa Enkripsi AES-CBC-256

Metrik	**Tanpa Virtualisasi**	**Virtualisasi**
Total Waktu Enkripsi (ms)	2,722.96	13,193.51
Total Waktu Dekripsi (ms)	2,055.01	12,529.90
Rata-rata Waktu per Blok Enkripsi (ms)	0.00272	0.01319
Rata-rata Waktu per Blok Dekripsi (ms)	0.00206	0.01253
Throughput Enkripsi (MB/s)	358.64	74.02
Throughput Dekripsi (MB/s)	475.21	77.94
Throughput Gabungan (MB/s)	416.92	75.98

Hasil pengujian enkripsi AES-CBC-256 menunjukkan overhead performa yang signifikan setelah penerapan VxLang. Total waktu enkripsi meningkat dari 2,722.96 ms menjadi 13,193.51 ms, yang merupakan peningkatan sekitar 384%. Total waktu dekripsi juga mengalami peningkatan yang serupa, dari 2,055.01 ms menjadi 12,529.90 ms (sekitar 510%). Peningkatan ini juga tercermin pada penurunan throughput (kecepatan pemrosesan data). Throughput enkripsi menurun dari 358.64 MB/s menjadi 74.02 MB/s, dan throughput dekripsi menurun dari 475.21 MB/s menjadi 77.94 MB/s. Penurunan throughput gabungan juga signifikan, dari 416.92 MB/s menjadi 75.98 MB/s. 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 5.3 menyajikan ukuran file executable (dalam KB) untuk berbagai program sebelum dan sesudah virtualisasi menggunakan VxLang.

Tabel 5.3: Hasil Pengujian Ukuran File (KB)

Program	**Ukuran File**	
	Tanpa Virtualisasi	**Virtualiasi**
quick_sort	119	1,951
encryption	131	1,834
size	97,802	112,716
console	105	1,942
app_imgui	1,773	2,753
app_qt	145	1,954

Hasil pengukuran ukuran file menunjukkan bahwa penerapan virtualisasi kode menggunakan VxLang secara umum meningkatkan ukuran file executable. Peningkatan ukuran file sangat signifikan untuk program-program kecil seperti quick_sort, encryption, dan console, di mana ukurannya meningkat lebih dari sepuluh kali lipat. Untuk aplikasi GUI yang lebih besar seperti app_imgui dan app_qt, peningkatan ukuran file juga terlihat, meskipun persentasenya mungkin lebih kecil dibandingkan dengan program yang lebih kecil. Peningkatan ukuran file size relatif lebih kecil, namun tetap signifikan. Peningkatan ukuran ini kemungkinan disebabkan oleh penambahan interpreter mesin virtual VxLang dan bytecode yang dihasilkan ke dalam executable yang dilindungi.

BAB 6
KESIMPULAN DAN SARAN

6.1 Kesimpulan

6.2 Saran

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] M. M. Aung and K. T. Win, "Simplifying control flow graphs for reducing complexity in control flow testing," *International Journal of Computer Trends and Technology (IJCTT)*, vol. 67, no. 8, pp. 7–12, 2019.

[9] Rountev, A. Volgin, O. Reddoch, and Miriam, "Static control-flow analysis for reverse engineering of uml sequence diagrams," *Static control-flow analysis for reverse engineering of UML sequence diagrams*, vol. 31, no. 1, pp. 96–102, 2005.

[10] Gomes and L.
bibinitperiod E. Schmitz, "A method for automatic generation of test cases to bpel processes.," *A Method for automatic generation of test cases to BPEL Processes.*,

[11] M. Hasbi, E. K. Budiardjo, and W. C. Wibowo, "Reverse engineering in software product line - a systematic literature review," in *2018 2nd International Conference on Computer Science and Artificial Intelligence*, Shenzhen, 2018, pp. 174–179. DOI: https://doi.org/10.1145/3297156.3297203.

[12] 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.

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

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

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

[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] K. Fukushima, S. Kiyomoto, T. Tanaka, and K. Sakurai, "Analysis of program obfuscation schemes with variable encoding technique," *Analysis of program obfuscation schemes with variable encoding technique*, vol. 91, no. 1, pp. 316–329, 2008.

[21] A. Kovacheva, "Efficient code obfuscation for android," in *Advances in Information Technology*, Bangkok, 2013.

[22] C. LeDoux, M. Sharkey, B. Primeaux, and C. Miles, "Instruction embedding for improved obfuscation," in *Annual Southeast Regional Conference*, Tuscaloosa, 2012.

[23] S. Darwish, S. Guirguis, and M. Zalat, "Stealthy code obfuscation technique for software security," in *International Conference on Computer Engineering & Systems*, Cairo, 2010.

[24] 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.

[25] M. Schloegel et al., "Hardening code obfuscation against automated attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, 2022.

[26] Y. Zhou, A. Main, Y. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *International Workshop on Information Security Applications*, Jeju Island, 2007.

[27] 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.

[28] 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.

[29] 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.

[30] P. Parrend, *Bytecode obfuscation*, https : / / owasp . org / www - community / controls / Bytecode_obfuscation, Mar. 2018. Accessed: Dec. 25, 2024.

[31] 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.

[32] 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.

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

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

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

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

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

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

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

LAMPIRAN