UNIVERSITAS INDONESIA

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

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*

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*
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*

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*

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*
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 EagleVM 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 EagleVM 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 EagleVM 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 EagleVM 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 EagleVM 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 EagleVM 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.

EagleVM merupakan salah satu platform code virtualization open-source yang menarik untuk dikaji. EagleVM menyediakan framework untuk melakukan code virtualization pada berbagai platform dan arsitektur prosesor [2]. Penelitian ini akan mengkaji implementasi dan efektivitas EagleVM dalam melindungi perangkat lunak dari rekayasa balik. Dengan menganalisis tingkat kesulitan reverse engineering pada kode yang dilindungi oleh EagleVM, 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 EagleVM pada perangkat lunak, termasuk tahapan-tahapan yang terlibat dan konfigurasi yang diperlukan?

2. Seberapa efektifkah code virtualization menggunakan EagleVM 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 EagleVM 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 EagleVM pada sebuah aplikasi studi kasus, mencakup seluruh tahapan implementasi dan konfigurasi.
2. Menganalisis efektivitas code virtualization menggunakan EagleVM 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 EagleVM 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 EagleVM
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.
4. Penelitian ini hanya berfokus pada teknik code virtualization dan tidak membahas teknik obfuscation lainnya.

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 EagleVM. 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 EagleVM 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 EagleVM 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 EagleVM.

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)

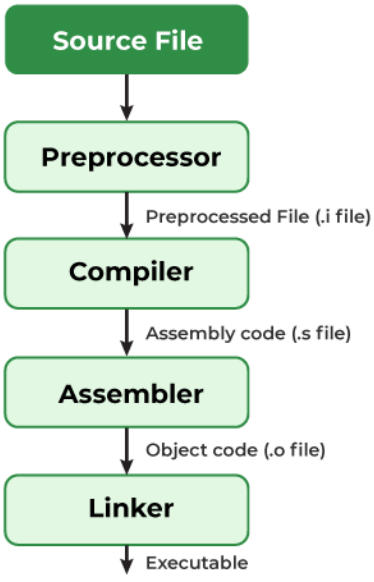
[4]. Proses ini melibatkan beberapa tahapan, yang masing-masing dilakukan oleh program utilitas yang berbeda:

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]

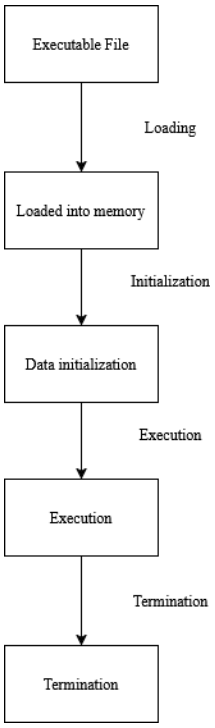
2.1.3.2 Proses Eksekusi

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.

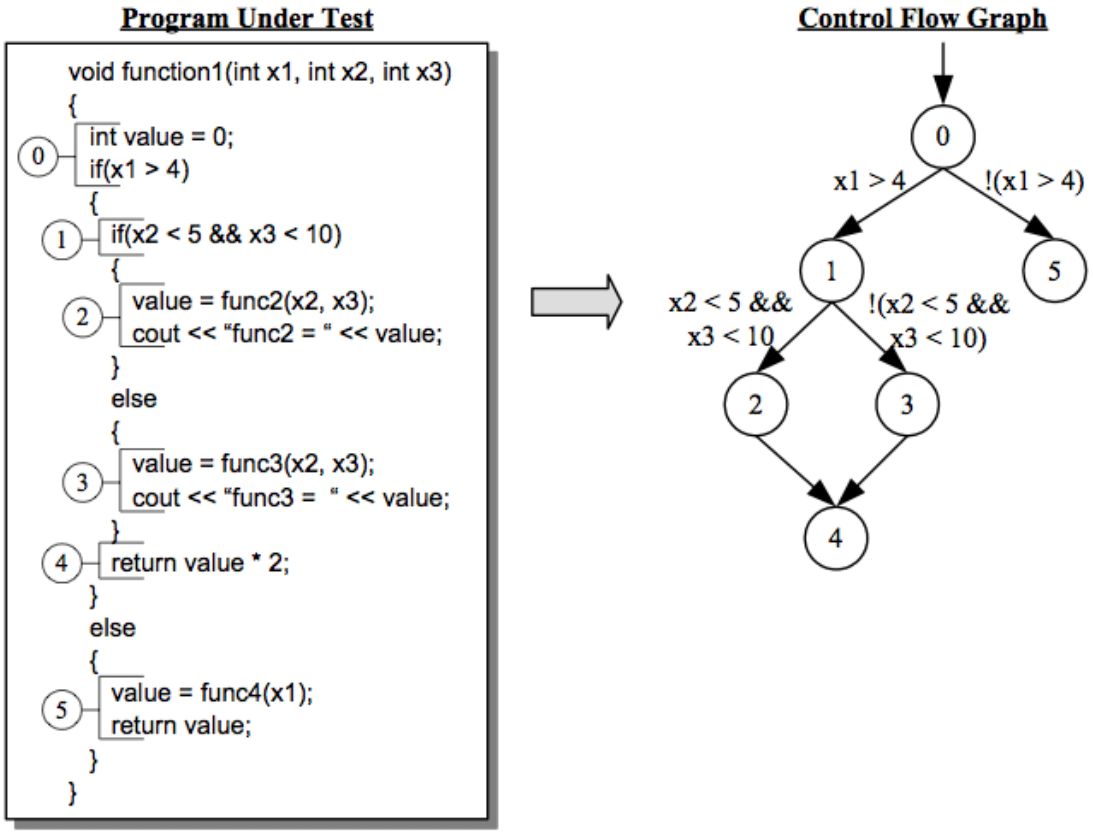
```
call function_address
```

- ***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

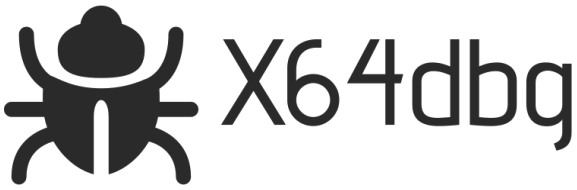
- **IDA Pro (Interactive Disassembler Pro):** Disassembler dan debugger komersial yang sangat populer dan powerful. IDA Pro mendukung berbagai arsitektur prosesor dan sistem operasi, menyediakan antarmuka yang canggih untuk analisis kode, dan mendukung plugin untuk ekstensibilitas [13].

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]–[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]–[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.

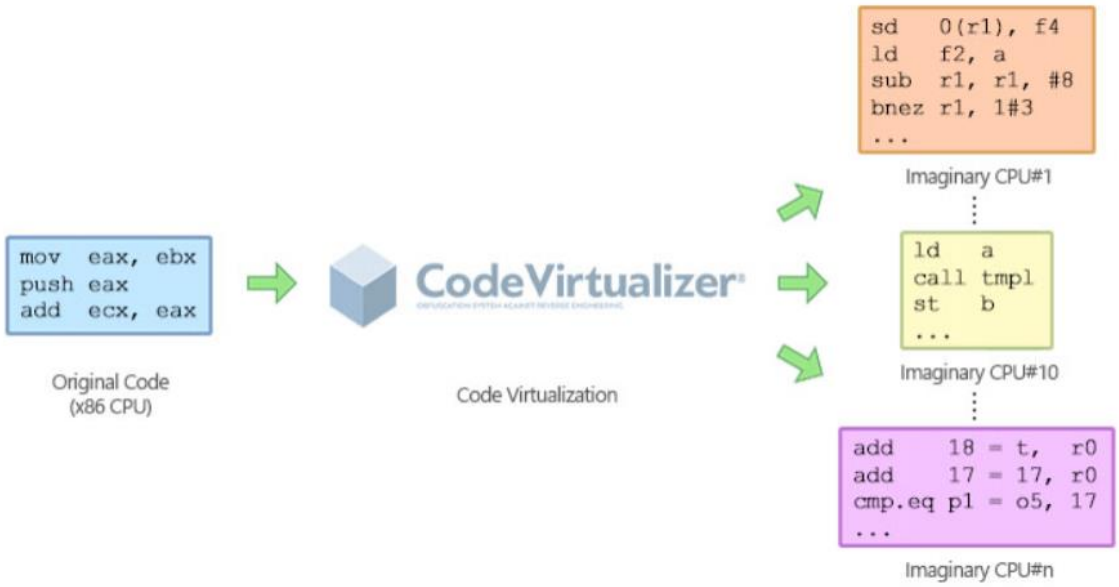
- ***Obfuscated Constants [32] :*** Menyembunyikan nilai konstan yang digunakan dalam kode melalui berbagai operasi aritmatika atau logika.
- ***Code Virtualization [1] :*** Menerjemahkan kode mesin menjadi instruksi virtual yang dieksekusi oleh mesin virtual khusus yang tertanam dalam aplikasi.

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).

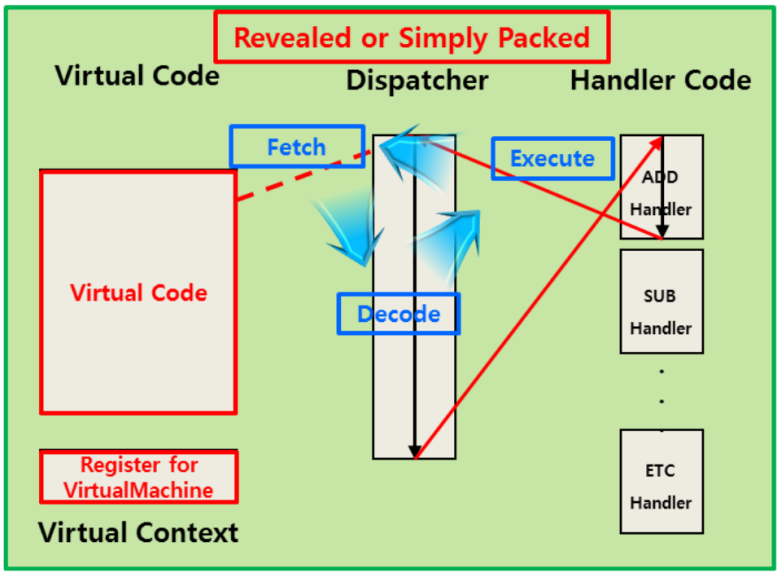
3. ***Decode:*** Perannya adalah untuk mendekode opcode yang diambil dan operan yang sesuai untuk menentukan instruksi ISA mana yang akan dieksekusi.

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

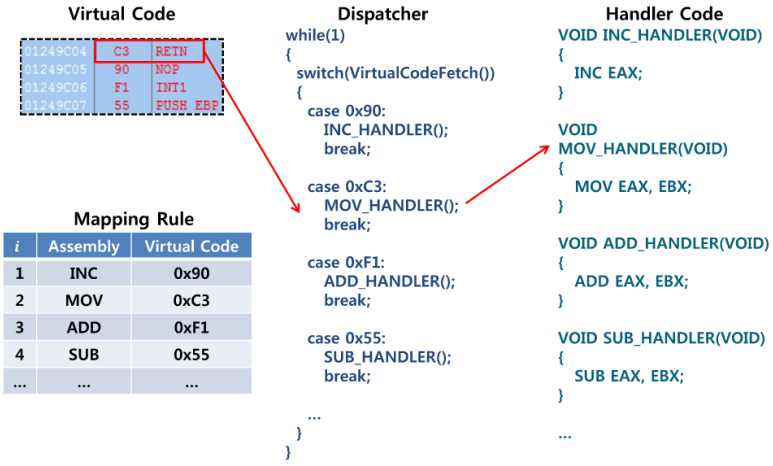
5. ***Handlers:*** Meniru instruksi virtual melalui rangkaian instruksi asli dan memperbarui konteks internal VM, biasanya vpc.

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 EagleVM

EagleVM adalah implementasi code virtualization open-source untuk arsitektur x86-64 yang dirancang khusus untuk tujuan penelitian dan demonstrasi [2]. Proyek ini merupakan hasil kompilasi penelitian dari Code Virtualizer komersial seperti VMProtect, Themida, dan Enigma Protector [1], [37], [38]. Proyek ini berfokus pada transformasi biner-ke-biner (bin2bin), memungkinkan modifikasi langsung pada file executable tanpa memerlukan akses ke kode sumber. EagleVM menyediakan kerangka kerja yang fleksibel untuk mempelajari dan mengembangkan teknik virtualisasi kode, serta menawarkan

kemampuan untuk melindungi bagian-bagian tertentu dari kode program dari analisis reverse-engineering. Meskipun bersifat open-source dan bermanfaat untuk riset, EagleVM tidak didesain untuk penggunaan produksi dan memiliki keterbatasan dalam hal stabilitas dan fitur.

2.5.1.1 Komponen EagleVM

EagleVM memiliki arsitektur modular yang terdiri dari beberapa komponen yang bekerja sama [2]:

- **EagleVM (Core):** Inti dari EagleVM, berisi logika utama untuk memproses biner input, melakukan disassembly, menerjemahkan instruksi x86-64 ke bytecode, dan menghasilkan biner output yang mengandung mesin virtual dan bytecode. Komponen ini juga bertanggung jawab untuk mengelola konteks eksekusi, termasuk pemetaan register dan stack.
- **EagleVM.Stub:** Dynamic Link Library (DLL) yang berperan penting dalam proses penandaan kode yang akan divirtualisasi. DLL ini mengekspor dua fungsi utama, yaitu fnEagleVMBegin dan fnEagleVMEnd. Pengembang menggunakan fungsi-fungsi ini di dalam kode sumber mereka untuk menandai awal dan akhir blok kode yang akan dilindungi dengan virtualisasi. EagleVM kemudian akan mencari panggilan ke fungsi-fungsi ini dalam biner input untuk mengidentifikasi bagian kode yang harus divirtualisasi.
- **EagleVM.Sandbox:** Sebuah aplikasi contoh yang berfungsi sebagai sandbox untuk mendemonstrasikan cara menggunakan EagleVM dan EagleVM.Stub. Proyek ini memberikan contoh konkret tentang bagaimana mengintegrasikan EagleVM ke dalam alur kerja pengembangan perangkat lunak dan bagaimana memanfaatkan fitur-fitur yang disediakan oleh EagleVM.Stub untuk menandai kode yang akan divirtualisasi.

2.5.1.2 Arsitektur Mesin Virtual EagleVM

Mesin virtual EagleVM didesain dengan mempertimbangkan kesederhanaan dan kemudahan implementasi, sambil tetap memberikan perlindungan yang memadai terhadap reverse engineering. Arsitekturnya menyerupai arsitektur CPU sederhana, dengan komponen-komponen utama sebagai berikut:

- **Konteks x86-64 dan VM:** Saat memasuki mesin virtual, konteks x86-64 asli (nilai register dan stack pointer) disimpan di stack. EagleVM kemudian mengalokasikan ruang di stack untuk konteks mesin virtual, termasuk register virtual dan virtual

call stack. Register R0-R15 x86-64 kemudian dapat digunakan oleh mesin virtual. Dokumentasi EagleVM menyebutkan bahwa register RFLAGS selalu disimpan pertama di stack karena otomatisasi dalam pembuatan VMENTER dan VMEXIT.

- **Register Virtual** EagleVM menggunakan sejumlah register virtual untuk eksekusi bytecode. Beberapa register virtual yang penting meliputi:
  - VIP (Virtual Instruction Pointer): Analog dengan RIP pada x86-64, menunjuk ke instruksi virtual selanjutnya yang akan dieksekusi.
  - VSP (Virtual Stack Pointer): Analog dengan RSP pada x86-64, menunjuk ke puncak stack virtual.
  - VREGS: Penunjuk ke awal area di stack yang menyimpan nilai register x86-64 yang disimpan.
  - VCS (Virtual Call Stack): Digunakan untuk menyimpan alamat kembali (return address) saat panggilan fungsi virtual.
  - VTEMP dan VTEMP2: Register sementara untuk perhitungan.
  - VCSRET: Menyimpan RVA yang digunakan dalam perhitungan VIP. Pemetaan register virtual ke register x86-64 dilakukan secara acak untuk mempersulit analisis.
- **Virtual Call Stack (VCS):** EagleVM menggunakan virtual call stack untuk mendukung panggilan fungsi di dalam kode yang divirtualisasi. VCS menyimpan alamat kembali ke instruksi setelah panggilan fungsi, memungkinkan mesin virtual untuk melanjutkan eksekusi dengan benar setelah fungsi selesai.
- **Handler Instruksi:** Setiap instruksi virtual memiliki handler yang berisi kode x86-64 untuk mengeksekusi instruksi tersebut. Handler inilah yang sebenarnya melakukan operasi yang ditentukan oleh instruksi virtual. EagleVM menyediakan handler untuk berbagai instruksi x86-64, dan pengembang dapat menambahkan handler baru untuk mendukung instruksi lain atau mengimplementasikan perilaku khusus.
- **VMENTER dan VMEXIT:** VMENTER bertanggung jawab untuk memasuki lingkungan mesin virtual, menyimpan konteks x86-64, menginisialisasi konteks VM, dan memulai eksekusi bytecode. VMEXIT bertanggung jawab untuk keluar dari lingkungan mesin virtual, memulihkan konteks x86-64, dan mengembalikan kontrol ke kode asli.

2.5.1.3 Alur Kerja Virtualisasi dengan EagleVM

Proses virtualisasi kode dengan EagleVM melibatkan langkah-langkah berikut:

1. **Penandaan Kode Target:** Pengembang memasukkan panggilan ke fungsi fnEagleVMBegin dan fnEagleVMEnd dari EagleVM.Stub di dalam kode sumber mereka untuk menandai blok kode yang akan divirtualisasi.

2. **Kompilasi dan Linking:** Kode sumber dikompilasi dan di-link untuk menghasilkan file executable.

3. **Pemrosesan oleh EagleVM:** File executable yang dihasilkan kemudian diproses oleh EagleVM. EagleVM akan memparsing file executable, mencari panggilan ke fungsi penanda dari EagleVM.Stub, dan mengekstrak kode yang ditandai untuk divirtualisasi.

4. **Disassembly dan Analisis Alur Kontrol:** EagleVM melakukan disassembly pada kode yang akan divirtualisasi dan membangun representasi alur kontrol program dalam bentuk basic blocks. Analisis ini penting untuk memastikan bahwa alur eksekusi kode yang divirtualisasi tetap sama dengan kode aslinya.

5. **Transformasi ke Bytecode dan Pembuatan VM:** EagleVM menerjemahkan instruksi x86-64 asli ke dalam bytecode khusus untuk mesin virtualnya. EagleVM juga menghasilkan kode untuk mesin virtual yang akan mengeksekusi bytecode tersebut.

6. **Pembuatan Biner Output:** EagleVM menghasilkan biner output yang telah dimodifikasi, mengandung bytecode dan mesin virtual.

7. **Eksekusi:** Saat program dijalankan, mesin virtual yang tertanam dalam biner akan mengambil dan mengeksekusi bytecode, mereplikasi fungsionalitas dari kode asli.

BAB 3
METODE PENELITIAN

3.1 Desain Penelitian

Penelitian ini menggunakan pendekatan eksperimental untuk mengevaluasi efektivitas dan dampak code virtualization menggunakan EagleVM. Dua percobaan utama akan dilakukan:

1. **Pengujian Autentikasi - Analisis Statis dan Dinamis:** Percobaan ini bertujuan untuk menganalisis kompleksitas control flow program sebelum dan sesudah di-obfuscate menggunakan EagleVM. Analisis dilakukan melalui dua pendekatan: analisis statis menggunakan Ghidra dan analisis dinamis menggunakan x64Dbg. Selain itu, percobaan ini juga akan mencakup upaya untuk melewati pengujian autentikasi dengan memanipulasi control flow menggunakan Ghidra dan x64dbg.
2. **Pengujian Performa - Waktu Eksekusi dan Ukuran File:** Percobaan ini bertujuan untuk mengukur performa waktu eksekusi program dengan algoritma sorting sebelum dan sesudah di-obfuscate, serta membandingkan ukuran file executable. Waktu eksekusi diukur menggunakan library chrono::high_resolution_clock.

3.2 Alat dan Bahan

Perangkat lunak dan perangkat keras yang digunakan dalam penelitian ini adalah:

- **Perangkat Lunak:**
  - **Sistem Operasi:** Windows 11 akan digunakan sebagai lingkungan operasi utama untuk melakukan semua percobaan.
  - **Kompiler:** Kompiler MSVC akan digunakan untuk mengkompilasi kode sumber.EagleVM
  - **Alat Virtualisasi Kode:** EagleVM akan digunakan sebagai alat utama untuk mengimplementasikan virtualisasi kode.
  - **Alat Analisis Statis:** Ghidra, sebuah framework reverse engineering yang dikembangkan oleh NSA, akan digunakan untuk melakukan analisis statis.

  - **Alat Analisis Dinamis:** x64dbg, sebuah debugger sumber terbuka untuk Windows, akan digunakan untuk analisis dinamis
  - **Pustaka C++:** Pustaka chrono akan digunakan untuk mengukur waktu eksekusi.

- **Perangkat Lunak:**
  - **Prosesor:** Intel Core i7-12700H
  - **RAM:** 16 GB

3.3 Prosedur Penelitian

3.3.1 Pengujian Autentikasi: Analisis Statis dan Dinamis

Pengujian autentikasi bertujuan untuk mengevaluasi efektivitas virtualisasi kode dalam menghambat upaya reverse engineering dengan menganalisis aspek statis dan dinamis dari perangkat lunak sebelum dan sesudah obfuscation, serta mencoba untuk melewati mekanisme autentikasi.

Berikut persiapan yang akan dilakukan :

1. **Pengaturan Lingkungan:** Lingkungan perangkat lunak diatur untuk memastikan bahwa semua alat (Windows 11, MSVC, EagleVM, Ghidra, dan x64dbg) terinstal dan terkonfigurasi dengan benar. Kode sumber aplikasi studi kasus disiapkan untuk kompilasi.

2. **Kompilasi Aplikasi Asli:** Aplikasi studi kasus dikompilasi tanpa menerapkan teknik obfuscation apa pun menggunakan kompiler MSVC. Hasilnya adalah file executable asli.

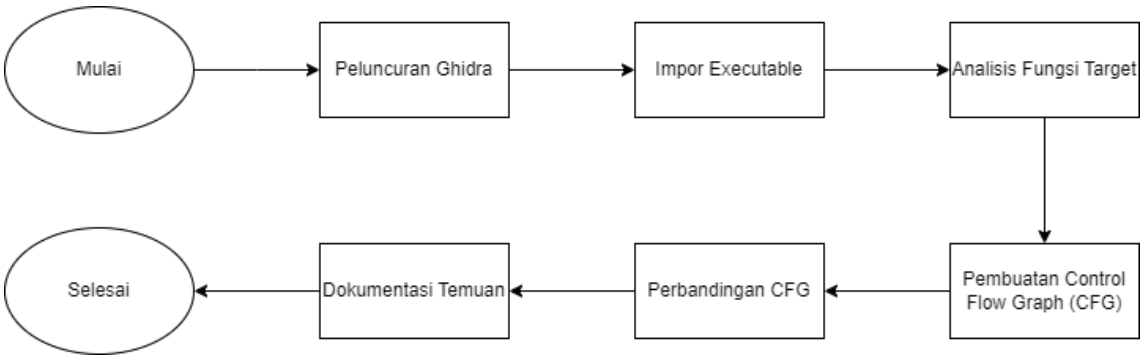
3. **Kompilasi Aplikasi yang di-Obfuscate:** Kode sumber aplikasi studi kasus dimodifikasi untuk menyertakan penanda fnEagleVMBegin dan fnEagleVMEnd dari EagleVM.Stub, untuk menunjukkan blok kode yang akan divirtualisasikan. Kode sumber yang dimodifikasi dikompilasi dengan kompiler MSVC, dan selanjutnya, executable diproses dengan EagleVM untuk menghasilkan versi yang di-obfuscate yang menyertakan mesin virtual dan bytecode yang sesuai.

Gambar 3.1: Flow Diagram Persiapan Penelitian

3.3.1.1 Analisis Statis (Ghidra)

Fase ini bertujuan untuk menganalisis struktur dan kompleksitas kode tanpa menjalankannya.

1. **Peluncuran Ghidra:** Framework Ghidra diluncurkan, dan proyek baru dibuat untuk menampung analisis.
2. **Mengimpor Executable:** Baik file executable asli maupun yang di-obfuscate diimpor ke dalam proyek Ghidra.
3. **Analisis Fungsi:** Sebuah fungsi target dalam aplikasi dipilih untuk analisis terfokus. Alat analisis Ghidra digunakan untuk membongkar fungsi target dan menghasilkan Control Flow Graph (CFG) untuk kedua versi executable.
4. **Perbandingan CFG:** CFG dari versi asli dan yang di-obfuscate dibandingkan dengan mendokumentasikan jumlah basic block, edge, dan kompleksitas siklomatik. Kompleksitas siklomatik yang lebih tinggi berkorelasi dengan peningkatan kesulitan dalam pemahaman kode dan reverse engineering. Dokumentasi untuk setiap versi akan mencakup yang berikut:
    - **Jumlah Basic Block:** Ini menunjukkan jumlah total urutan kode non-percabangan dalam alur kontrol fungsi.
    - **Jumlah Edge:** Ini mewakili alur eksekusi dari satu blok ke blok lainnya.
    - **Kompleksitas Siklomatik:** Metrik ini mengukur jumlah jalur independen dalam alur kontrol kode. Ini dihitung berdasarkan rumus: Edge - Node + 2
5. **Mendokumentasikan Temuan:** Catatan dan observasi terperinci mengenai analisis statis kode dipelihara.



Gambar 3.2: Flow Diagram Analisis Statis

Upaya Manipulasi Control Flow:

1. **Identifikasi Mekanisme Autentikasi:** Menggunakan Ghidra, identifikasi bagian kode yang menangani proses autentikasi. Ini melibatkan pencarian fungsi yang memvalidasi kredensial atau memeriksa lisensi.

2. **Analisis Control Flow:** Analisis control flow dari fungsi autentikasi untuk memahami bagaimana keputusan autentikasi dibuat.

3. **Modifikasi Control Flow:** Menggunakan Ghidra, modifikasi kode assembly untuk mengubah alur kendali agar melewati pemeriksaan autentikasi. Ini bisa melibatkan perubahan conditional jump atau patching instruksi.

4. **Penyimpanan Perubahan:** Setelah modifikasi dilakukan, simpan perubahan pada file executable yang telah di-patch.

5. **Pengujian:** Jalankan file executable yang telah dimodifikasi dan verifikasi apakah modifikasi control flow berhasil melewati mekanisme autentikasi.

3.3.1.2 Analisis Dinamis (x64dbg)

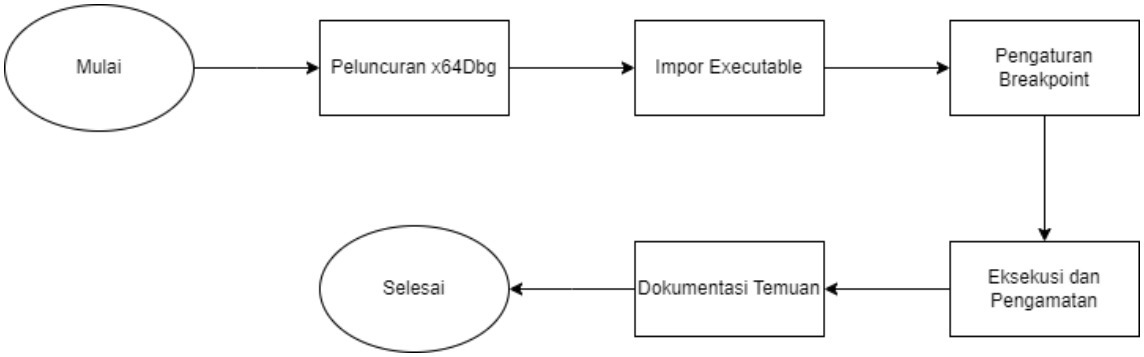
Analisis dinamis melibatkan pengamatan perilaku program saat dijalankan.

1. **Peluncuran x64Dbg:** Debugger x64dbg diluncurkan, dan baik executable asli maupun yang di-obfuscate dimuat.

2. **Pengaturan Breakpoint:** Breakpoint ditempatkan di awal fungsi target dalam setiap versi aplikasi untuk mengontrol eksekusi selama analisis.

3. **Menjalankan dan Mengamati:** Aplikasi dijalankan, dan eksekusi dilacak. Ini termasuk mengamati alur eksekusi, perilaku program, dan mencatat setiap perbedaan dalam jalur eksekusi.

4. **Mendokumentasikan Temuan:** Catatan terperinci dibuat tentang perbedaan yang diamati dalam alur eksekusi dan kesulitan yang dihadapi dalam memahami perilaku versi yang di-obfuscate.



Gambar 3.3: Flow Diagram Analisis Dinamis

3.3.2 Pengujian Performa: Waktu Eksekusi dan Ukuran File

Pengujian kinerja ini bertujuan untuk mengevaluasi dampak virtualisasi kode terhadap waktu eksekusi dan ukuran file aplikasi. Untuk mendapatkan pengukuran yang komprehensif, kita akan menggunakan algoritma pengurutan (sorting) sebagai benchmark.

Berikut persiapan yang akan dilakukan :

1. **Pemilihan Algoritma Pengurutan:** Algoritma pengurutan yang akan digunakan sebagai benchmark adalah algoritma pengurutan cepat (Quicksort). Quicksort dipilih karena efisiensinya yang baik dalam banyak kasus, serta kompleksitasnya yang cukup untuk memberikan gambaran yang baik tentang performa.

2. **Pengembangan Benchmark:** Benchmark akan diimplementasikan dengan membuat fungsi yang melakukan pengurutan pada array data. Array data akan diisi dengan angka acak untuk memastikan pengujian yang representatif. Ukuran array data akan divariasikan (misalnya, 1000, 10000, 100000 elemen) untuk mengamati performa pada berbagai skala data.

3. **Integrasi Benchmark:** Fungsi benchmark diintegrasikan ke dalam kedua versi aplikasi (asli dan yang di-obfuscate). Hal ini memungkinkan kita untuk mengukur waktu eksekusi dari algoritma pengurutan pada kedua versi aplikasi secara terpisah.

3.3.2.1 Pengukuran Waktu Eksekusi

1. **Eksekusi Benchmark:** Fungsi benchmark (algoritma pengurutan) akan dieksekusi beberapa kali (misalnya, N=10) pada setiap ukuran array untuk mendapatkan data yang cukup.

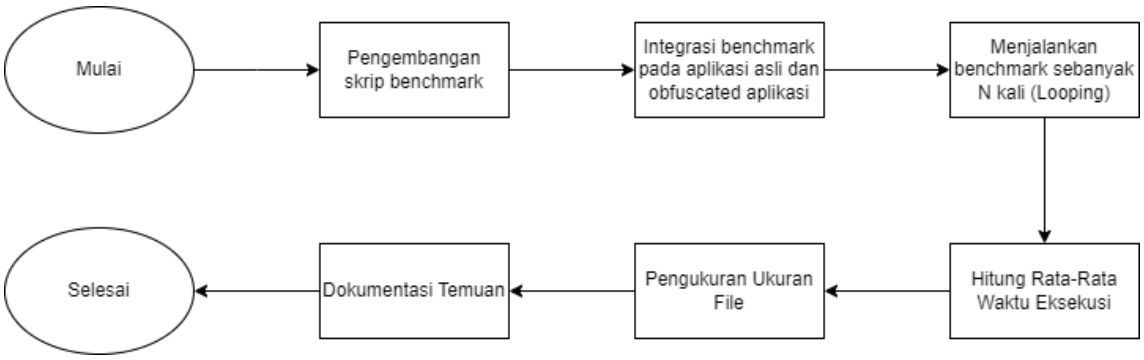
2. **Pengukuran Waktu:** Pustaka chrono::high_resolution_clock dari C++ akan digunakan untuk mengukur secara akurat waktu yang dibutuhkan untuk setiap eksekusi benchmark.

3. **Pencatatan Data:** Waktu eksekusi setiap run (dalam milidetik atau mikrosekon) dicatat untuk setiap ukuran array pada kedua versi aplikasi.

4. **Perhitungan Rata-Rata:** Waktu eksekusi rata-rata dihitung untuk setiap versi dan setiap ukuran array dengan menjumlahkan semua waktu eksekusi dan membaginya dengan jumlah run (N).

3.3.2.2 Pengukuran Ukuran File

1. **Perbandingan Ukuran File:** Ukuran file dari executable asli dan yang di-obfuscate dibandingkan untuk menentukan dampak virtualisasi kode pada ukuran aplikasi.



Gambar 3.4: Flow Diagram Analisis Performa

3.4 Teknik Analisis Data

Data yang dikumpulkan dari kedua rangkaian percobaan akan dianalisis menggunakan teknik deskriptif dan komparatif.

3.4.1 Analisis Data Pengujian Autentikasi

- **Analisis Data Statis:** Metrik kompleksitas yang diperoleh dari analisis statis (yaitu, jumlah basic block, edge, dan kompleksitas siklomatik) akan dievaluasi. Perbedaan akan digunakan untuk menilai efektivitas virtualisasi kode.

- **Analisis Data Dinamis:** Hasil dari analisis dinamis, termasuk alur eksekusi yang diamati dan kesulitan dalam analisis kode, akan dianalisis.

- **Evaluasi Efektivitas Keamanan:** Hasil dari analisis dinamis, termasuk alur eksekusi yang diamati dan kesulitan dalam analisis kode, akan dianalisis.

3.4.2 Analisis Data Pengujian Autentikasi

- **Perbandingan Waktu Eksekusi:** Waktu eksekusi rata-rata antara aplikasi asli dan yang di-obfuscate dibandingkan untuk menilai potensi overhead kinerja yang diperkenalkan oleh EagleVM.
- **Perbandingan Ukuran File:** Ukuran file dari kedua versi dibandingkan untuk menilai dampak ukuran file dari virtualisasi kode.
- **Analisis Trade-off:** Analisis komprehensif dilakukan untuk mengevaluasi trade-off antara keamanan dan kinerja, berdasarkan temuan.

BAB 4
HASIL PENELITIAN

@todo
bab 4

4.1 oof

BAB 5
KESIMPULAN

5.1 Saran

DAFTAR REFERENSI

[1] Oreans, *Code virtualizer*, https://www.oreans.com/CodeVirtualizer.php, May 2006. (visited on 11/11/2024).

[2] *Eaglevm documentation*, https://notpidgey.github.io/posts/eaglevm/, Dec. 2021. (visited on 11/11/2024).

[3] geeksforgeeks, *Software and its types*, https://www.geeksforgeeks.org/software-and-its-types/, Aug. 2023. (visited on 11/04/2024).

[4] geeksforgeeks, *Compiling a c program: Behind the scenes*, https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/, Oct. 2024. (visited on 11/10/2024).

[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. (visited on 11/15/2024).

[6] codeacademy, *Control flow*, https://www.codecademy.com/resources/docs/general/control-flow, Jun. 2023. (visited on 11/27/2024).

[7] USC Viterbi School of Engineering, *Cs356 unit 5 x86 control flow*, https://ee.usc.edu/~redekopp/cs356/slides/CS356Unit5_x86_Control. (visited on 11/12/2024).

[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. (visited on 12/18/2024).

[13] Hex-Rays, *Ida pro*, https://hex-rays.com/ida-pro, May 1991. (visited on 11/22/2024).

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

[15] D. Ogilvie, *X64dbg*, https://x64dbg.com, May 2014. (visited on 11/11/2024).

[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, T. Blazytko, M. Contag, *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. (visited on 12/25/2024).

[31] Yakov, *Using llvm to obfuscate your code during compilation*, https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation, Jun. 2020. (visited on 12/20/2024).

[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. (visited on 11/22/2024).

[37] Oreans, *Themida*, https://www.oreans.com/Themida.php. (visited on 11/23/2024).

[38] VMProtect Software, *Vmprotect*, https://vmpsoft.com/vmprotect/overview. (visited on 11/23/2024).

LAMPIRAN