BAB II
TINJAUAN PUSTAKA

II.1 Debugging

Bug pada perangkat lunak merupakan sebuah kesalahan dalam program atau sistem yang menyebabkan setidaknya satu fungsional atau non-fungsional menjadi terganggu. *Debugging* adalah sebuah proses sistematis memerika program dari *bug*. *Debugging* sendiri diambil dari kata *bug* yang fokus utamanya memang untuk melepas *bug*. Namun, tujuan akhirnya tidak selalu untuk memperbaiki program, beberapa *reverse engineer* melakukannya agar dapat mengambil keuntungan tertentu seperti eksploitasi.

Pada umumnya *debugging* adalah bagaimana menciptakan suatu masalah seperti input diluar ekspektasi atau dengan membuat *test case* sembarangan *(black box testing)* sampai menimbulkan *crash.* Setelah program mencapai keadaan *crash* kemudian *debugger* mengeksaminasi status program dimana terjadi urutan letak masalah atau dikenal sebagai *call stack.* Beberapa *call stack* memiliki informasi yang kurang jelas atau implisit tergantung dari tingkatan kompleksnya sebuah program.

II.2 Macam-Macam Format Debugging

Format *Debugging* merupakan desain data yang tersimpan dalam program untuk mempermudah *debugger* dalam menganalisis. Macam – macam format debugging diantaranya:

1. STAB (symbol table) menyimpan informasi *debug* ke bagian simbolis file objek yang umumnya digunakan oleh *executable* pada sistem operasi UNIX terdahulu dan digunakan hingga sekarang untuk alasan kompabilitas.
2. DWARF (debug with arbitrary record format) menggunakan file objek untuk menyimpan informasi *debug* dengan struktur pohon di mana setiap simpul mewakili tipe data, variabel atau fungsi. DWARF menjadi standar hampir untuk semua *compiler*.

II.3 Instrumentation

Instrumentasi dalam istilah pemograman adalah tindakan menanamkan kode *monitoring* ke dalam sistem perangkat lunak. Instrumentasi dapat dilakukan dengan cara transformasi biner program, manipulasi *link-time, hooks* khusus dalam *virtual machine*(bahasa pemograman tingkat tinggi/interpreter) atau dengan transformasi pada sumber kode. Keluaran yang dihasilkan berupa peristiwa yang terjadi pada program dalam konteks spesifik, tidak meluas atau hanya bergantung pada komponen dan fungsi yang ditinjau.

Kelemahan instrumentasi terletak pada jangkauannya yang mengurucut ke fungsi target sehingga memungkinkan kode yang di-*instrument* lepas dari eksamen atau tidak sampai mencapai poin eksekusi ketika fungsi tersebut memanggil fungsi lainnya. *Instrumentor* juga berperan menggapai memori yang diakuisisi program dan membuat kinerja program menjadi lamban atau bahkan *crash.* Hal tersebut sangat bergantung pada kondisi *environment* atau *resource* yang dimiliki.

Dalam menentukan instrumentasi apa yang cocok untuk digunakan perlu dilihat ketersediaan informasi program. Adapun kategori tersebut dikategorikan berdasarkan *language agnostic* yakni:

1. DLL(*Dynamic Link Library)* Rewriting

   Instrumentasi dijalankan saat setelah siklus kompilasi/*linking* dimana penggunaan hasil *cache* merupakan DLL yang telah dimodifikasi

2. In Place Instrumentation

   Sama dengan DLL *rewriting* akan tetapi DLL tidak dimodifikasi begitu juga dengan target program. Fungsi – fungsi di-*hook* tepat untuk tugas yang diperlukan ketika DLL tersebut dimuat atau di-*load* pertama kali saat *startup* atau setelah panggilan *LoadLibrary*

3. On-the-fly Instrumentation

Mirip dengan *in place instrumentation* tetapi hanya menginstrumentasi sebuah fungsi saat pertama kali fungsi tersebut dieksekusi

4. Intermediate Language Instrumentation

Instrumentasi pada bahasa tingkat menengah adalah memonitor aktivitas eksekusi sebelum bahasa tingkat menengah dikompilasi ke dalam instruksi *executable* oleh *virtual machine* atatu *interpreter*

5. Intermediate Language Instrumentation via Reflection

Java dan .Net keduanya memberikan API refleksi yang memungkinkan penemuan metadata tentang metode. Dengan menggunakan data tersebut dapat dibuat metode baru dengan *on-the-fly instrumentation* yang ada seperti dengan instrumentasi bahasa tingkat menengah disebutkan sebelumnya.

6. Compile Time Instrumentation

Teknik ini digunakan pada waktu kompilasi untuk menyisipkan instruksi yang tepat ke dalam aplikasi selama kompilasi. Kelemahannya membutuhkan usaha pembangunan *linking* yang besar.

7. Source Code Instrumentation

Teknik ini digunakan untuk memodifikasi kode sumber dengan menyisipkan kode instrumentasi yang sesuai dengan dekorator sebelum masing-masing fungsi.

8. Link Time Instrumentation

Teknik ini sangat berguna ketika dibutuhkannya penggantian *memory allocators* seperti fungsi *realloc* dengan *tracing allocators.* Hal tersebut berkaitan dengan pencarian *bug* pada memori seperti *buffer overflow* atau *memory leak*

9. IAT Hooking Instrumentation

Instrumentasi ini melibatkan modifikasi *import address table* untuk fungsi terkait dalam DLL atau *Unix shared object.* Hanya dengan *patch* lokasi *import table* dengan fungsi instrumentasi kemudian memanggil fungsi asli dari *hook.*

II.4 Metode Binary Instrumentation

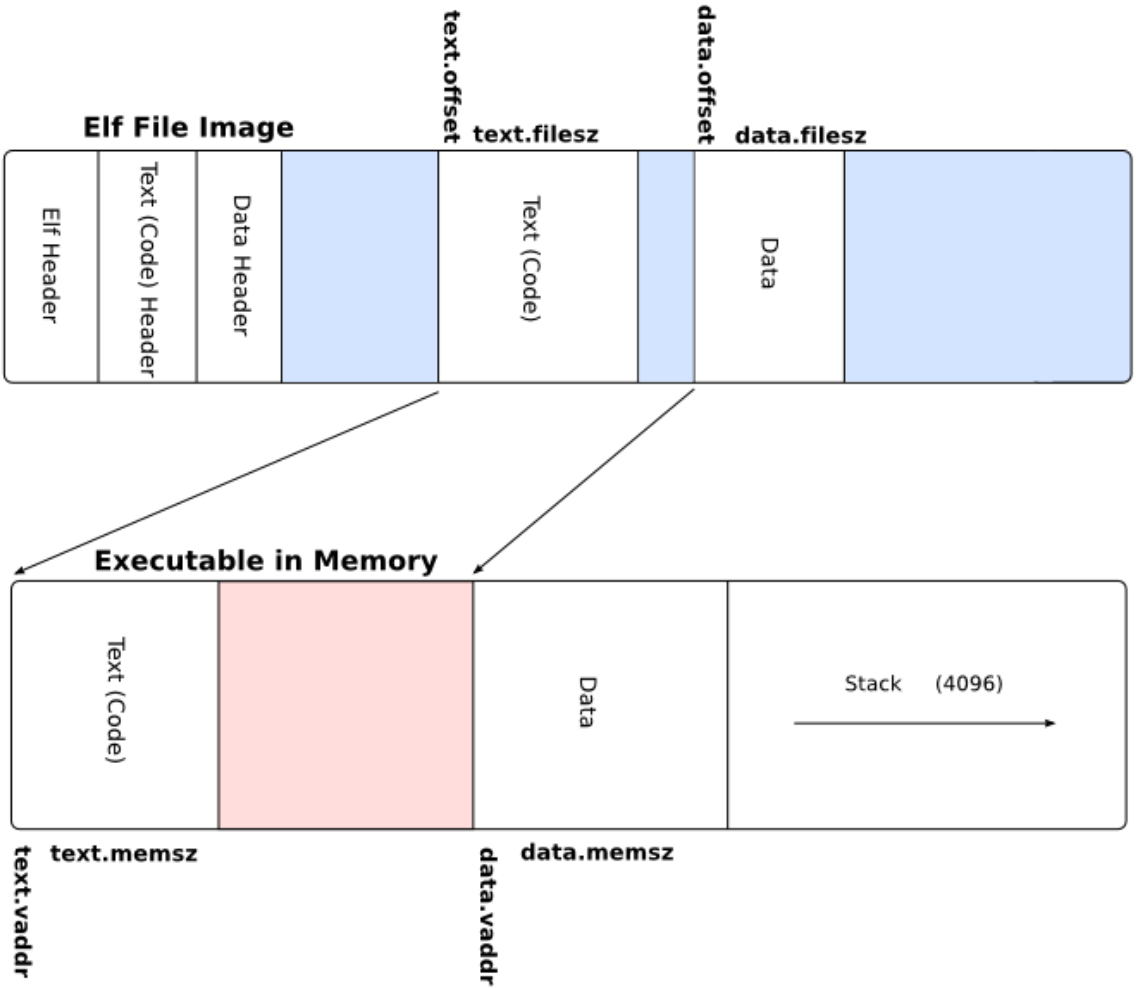
Binary Instrumentation merupakan proses memodifikasi instruksi sebuah program ketika eksekusi. Modifikasi tersebut dapat diaplikasikan pada file *executable,* file objek maupun file kelas pada semua fase seperti waktu kompilasi, *linking* pustaka, *load time* atau *run-time. Dynamic Instrumentation* beroperasi dengan menanamkan kode dinamis ke dalam program tanpa harus rekompilasi. *Dynamic Binary Instrumentation* dapat menganalisis eksekusi program secara *step-by-step* pada konteks memori dan register dan hanya menganalisis kode yang dieksekusi.

Dynamic Binary Instrumentation adalah sebuah metode analisis tingkah laku aplikasi secara runtime pada saat eksekusi melalui penanaman kode instrumentasi. Kode intrumentasi tersebut mengeksekusi sebagai bagian dari instruksi asli. DBI dibagi atas dua berdasarkan cara kerjanya yaitu *light-weight* dan *heavy-weight. Light-weight* DBI beroperasi pada instruksi *stream* secara spesifik berdasarkan arsitektur dan status perlakuan analisis sedangkan *heavy-weight* beroperasi secara abstrak pada *stream* dan status instruksinya [2].

Berbeda dengan *Static Binary Instrumentation* yang memodifikasi file *executable* maupun pustakanya dengan pendekatan sebelum eksekusi program. Instrumentasi biner dinamis memanipulasi saat program di tahap sudah dimuat dalam memori. Instrumentasi biner secara statis mempunyai keunggulan dalam hal performansi karena DBI melakukan penyisipan sebelum eksekusiyang berakibat akses informasi seperti nama variabel menjadi sulit.

II.5 ELF

ELF (Executable and Linking Format) ialah format file *executable* standar UNIX System V seperti Linux yang dibagi beberapa seksi seperti .*text* untuk kode, .*data* untuk variabel *global*, .*rodata* yang umumnya menyimpan *strings* konstan, .*bss*(Block Started by Symbol) *data segment* yang mengandung variabel statis representasi oleh *zero-valued(*pengosongan memori*)* ketika eksekusi dimulai dan .*stab* untuk simbol *debugging.* File ELF juga berisi header yang menggambarkan bagaimana seksi tersebut harus disimpan dalam memori.



Gambar II.1 ELF Diagram

ELF dahulunya disebut *Extensible Linking Format* karena dapat direlokasi dengan mudah mengandalkan *linking* dan *loading* DSO (*dynamically linked shared objects).* Namun, ELF juga mampu mendukung *position independent code* yang menghindari pengalamatan secara absolut atau tidak membutuhkan relokasi sama sekali. Hal tersebut berbeda dengan pustaka statis walaupun ELF juga mendukung sistem *portable.*

II.6 Self-Modifying Code

Self-Modifying Code adalah sebuah teknik pemograman dimana program dapat merubah fungsi dengan sendirinya ketika eksekusi. Teknik tersebut

merupakan cetusan dari *Rice's theorem* dimana untuk setiap properti *non-trivial* dari fungsi parsial, tidak ada metode umum dan efektif yang dapat memutuskan apakah suatu algoritma mengkalkulasi fungsi parsial properti tersebut. Logika *Self-Modifying Code* pada dasarnya menggunakan ekspresi *lambda* kalkulus dalam subtitusi fungsi.

Self-Modifying Code kerap kali diidentikkan dengan salah satu teknik *obfuscation* namun, program tidak serta merta menjadi kompleks dalam hal analisis melainkan performa program yang membaik dan baris kode yang lebih efisien. Rata – rata program *obfuscated* adalah memberikan *junk* yang tidak berarti untuk mengelabui analis sedangkan *Self-Modifying Code* umumnya mengambil jalan potong untuk instruksi yang lebih sederhana.

Berdasarkan fungsinya *Self-Modifying Code* dibagi atas beberapa tujuan. Adapun contoh diantaranya yang sering digunakan adalah sebagai berikut:

1. Self-Checksumming

Program melakukan pengecekan integritas terhadap variasi segmen kode saat eksekusi untuk memastikan bahwa biner program tidak diubah. Verifikasi berada pada segmen memori dengan status *segmentation fault* atau program dihentikan dengan *signal handler* interupsi tertentu jika terdeteksi ketidaksesuaian. Hal ini bukan mengecek apakah *hash checksum* dari keseluruhan biner berbeda melainkan keutuhan dari instruksi program. Adapun contoh dari *Self-Checksum* dapat dilihat pada *pseudocode assembly* berikut:

```
add (rbx), r15 ;membaca memori
sub 1, ecx ;looping counter secara decrement
add rdi, rax
xor r14, rdi
add rcx, rdx
add rbx, rdi
xor rax, rdx ;modifikasi register jump_target yaitu rdx dan rdi
xor r15, rdi
add rdx, r15
```

```
add rdi, r14 ;modifikasi checksum dengan rdx dan rdi
xor rdx, -8(rsp) ;modifikasi checksum dalam stack
xor r15, r13
add r14, r12
rol r15
xor rdi, rbx ;membuat pseudorandom akses memori
and mask1, ebx
or mask2, rbx
xor rdx, rsp
and mask3, esp
or mask4, rsp
and 0x180, edx ;Modifikasi penunjuk stack dan alamat target_jump
and 0x1, rdi
add rdi, rdx
add rdi, rdi
add rdi, rdx
or mask, rdx ;membuat alamat target_jump
xor rdx, r15 ;Menambahkan alamat target_jump ke dalam checksum
mov rax, rdi
imul rax, rax
or 0x5, rax
pushfq
add (rsp), rbx
jmp *rdx ;jump ke 1 dari 4 blok
```

2. Self-Decrypting

Enkripsi dapat diimplementasikan pada biner *executable* dan didekripsi pada saat program dijalankan. Biasanya menggunakan instruksi aritmetik XOR(*exclusive-or)* maupun *polymorphic* dengan menambahkan operasi bit instruksi ADD(*addition)* dan SUB(*subtraction)*. Contoh *Self-Decrypt* dapat dilihat pada *pseudocode assembly* berikut:

```
main:
   pop rsi          ; pop alamat data
   mov cl, 38        ; set jumlah loop ke ukuran dari data yang
```

```
dienkripsi
decode:
   mov al, byte [rsi]    ; Muat karakter dari kunci ke al
   xor byte [rsi+38], al      ; xor kunci dengan kode yang
dienkripsi
   inc rsi         ; increment alamat dekode loop
   jmp rsi         ; jump ke kode yang didekripsi
```

3. Self-Extracting

Biner *executable* dapat diperkecil ukuran datanya dengan teknik kompresi yang kemudian diekstrak ke memori untuk mendapatkan fungsi program sebenarnya. Algoritma kompresi yang digunakan sama dengan metode untuk kompresi file seperti Huffman/Lempel-Ziv, Lempel-Ziv-Welch, Lempel-Ziv-Markov dan PAQ akan tetapi *header* atau *magic* program tetap sebagaimana *executable* semestinya. Berikut contoh *self-extract* dalam bahasa pemograman *assembly:*

```
inject_loop:
    cmpb $0x20, (%rax, %rsi, 1)
    je inject_finished
    movb (%rax, %rsi, 1), %r10b
    xor $0x3, %r10b
    movb %r10b, (%rcx, %rsi, 1)
    inc %rsi
    jmp inject_loop

inject_finished:
    inc %rsi
    movb $0xc3, (%rcx, %rsi, 1)
    push %rdi
    push %rcx
    ret
```

4. Self-Repairing

Self-Error-Correcting atau *Self-Healing* code merupakan upaya untuk memperbaiki perangkat lunak dari *error* dengan sendirinya agar tidak pernah *crash.* Hal ini merupakan topik lanjutan dari SMC karena menggunakan kecerdasan buatan dalam implementasinya. Konsep dari *Self-Repairing* adalah menyalin instruksi *clean* untuk memperbaiki jika ada satu atau lebih instruksinya rusak. Adapun contohnya dapat dilihat pada *pseudocode assembly* berikut yang merupakan *safepoint* pada *hardisk* atau S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology)

```
disk:
      movb  cl, (si)     ; Jumlah sektor yang akan dibaca
      push  si           ; Simpan si
      mov   si, #LOADOFF+ext_rw ;
      movb  2(si), cl    ; Mengisi blok yang akan ditransfer
      mov   4(si), bx    ; alamat buffer
      mov   8(si), ax    ; Memulai angka blok
      mov   10(si), dx
      movb  dl, device(bp)     ; dl = device yang akan dibaca
      movb  ah, #0x42
      int   0x13
      pop   si           ; Mengeluarkan si ke poin alamat array
      !jmp  rdeval
rdeval:
      jc    error        ; Jump ke disk yang error dibaca
      movb  al, cl
      addb  bh, al
      addb  bh, al
      add   1(si), ax    ; Pembaharuan alamat oleh sector baca
      adcb  3(si), ah
      subb  (si), al     ;  Decrement jumlah sektor dengan sector
baca
      jnz   load         ;
      add   si, #4             ; memperbaiki alamat selanjutnya
      cmpb  ah, (si)     ;  Selesai  ketika  tidak  ada  sector  lagi
```

```
yang dibaca
      jnz   load        ; Baca instruksi selanjutnya
```

5. Self-Testing

Self-Testing Code bertujuan me-*refactoring* kode agar dapat menguji *testcase* dengan sendirinya. Pendekatan ini mirip dengan *built-in test* akan tetapi eksekusi bersifat jika peristiwa tertentu terjadi seperti kegagalan fungsi progam (*bug)* dalam menjalankan tugasnya. Adapun contohnya dapat dilihat pada *pseudocode* berikut:

```
      MOV R1, 00000000 ; R1 = 0
      MOV R2, 00000000 ; R2 = 0
      MOV R6, 00000000 ; R6 = 0
      MOV R7, signature  ; deklarasi signature
      MOV R8, 11111111 ; 4-bit increment dengan 1
loop: UMUL R4,R3,R2,R1 ; multiplikasi
      ADD R5,R4,R3 ; tambah words
      ADD R6,R6,R5
      ADC R6,R6,0
      ADD R2,R2,R8 ; increment R2
      CMP R2, 11111110
      BNE loop
      MOV R2, 00000000 ; R2 = 0
      ADD R1,R1,R8 ; increment R1
      CMP R1, 11111110
      BNE loop
      CMP R6,R7 ; cek signature
      BNE test_fail_routine
      JMP test_pass_routine
```

II.7 Anti Debugging

Anti-debugging adalah salah satu upaya untuk menghambat *reverse engineering* sehingga analis tidak dapat menggunakan *debugger* untuk menganalisis program. Teknik ini umum digunakan untuk mencegah pembajakan

perangkat lunak dan infeksi dari program lain seperti *malware (malicious software).* Pada dasarnya semua sistem proteksi *software* primitif dibagi atas dua ide yaitu *checks* dan *landmines. Checks* menghasilkan nilai ubahan atau jalan eksekusi kode berbasis status item yang dicek sedangkan *landmines* seperti *crash, hang, scramble* yang menginterupsi alat analisis atau *debugger* itu sendiri.

Beberapa bahasa pemograman telah menyediakan fungsi *built-in* berupa API (*Application Programming Interface)* untuk pendeteksian *debugger* dan *virtual machine.* Selanjutnya *programmer* dapat memutuskan apakah *debugger* diizinkan untuk menganalisis atau tidak. *Anti-debugging* juga dapat diciptakan oleh *compiler* yang fungsinya menghapus *readable strings (metadata)* termasuk informasi *debug* atau dengan memberikan *obfuscation* agar instruksi program menjadi kompleks.

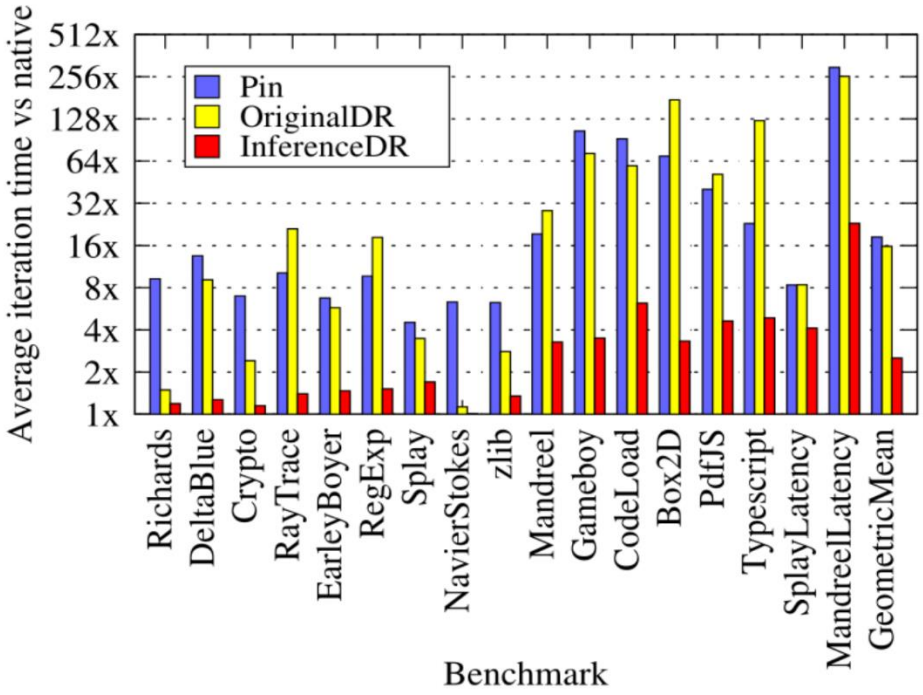
II.8 DynamoRIO

DynamoRIO adalah salah satu *framework* atau alat *Dynamic Binary Instrumentation* yang tersedia *multiplatform* dan sumber terbuka. Alat tersebut merupakan kombinasi dari *Dynamo,* sebuah mesin optimasi dinamis yang dikembangkan oleh peniliti dari HP dan RIO yaitu *runtime instropection and optimization engine* yang dikembangkan oleh MIT. *DynamoRIO* mendukung analisis aplikasi pada sistem operasi seperti Windows, Linux dan OSX yang mempunyai arsitektur x86, AMD64 dan ARM.

Salah satu kelebihan *DynamoRIO* adalah analisis sudut pandangnya yang menanamkan kode instrumentasi ketika *binary fragment* sedang dimasukkan ke dalam *fragment cache.* Hal tersebut sangat berguna untuk mencegat akses memori di dalam aplikasi. Ketika fragmen sedang dibuat, pustaka analisis *DynamoRIO* berperan masuk ke instruksi dalam fragmen yang dihasilkan. Menggunakan pengetahuan tersebut memungkinkan analis merancang pustaka analisis yang dapat mencegat memori dalam membaca dan menulis ketika aplikasi dieksekusi.

Performansi merupakan kendala besar dalam instrumentasi biner dinamis. PIN yang merupakan perangkat lunak DBI dari Intel kurang unggul dalam hal

performansi karena menggunakan rekompilasi JIT(*Just-in-Time*) terhadap seksi kode yang memerlukan kerja ekstra. Namun untuk mengatasi logika kompleks *DynamoRIO* dirasa tidak mumpuni karena ketersediaan pada kode *cache* terbatas. Berikut ini dapat dilihat *benchmark* dalam analisis perangkat lunak *chromium* yang menggunakan beberapa pustaka seperti kriptografi hingga multimedia



Gambar II.2 DynamoRIO Benchmark

PIN berulang kali melakukan iterasi untuk memastikan instruksi yang kompleks benar-benar valid. Sedangkan *Inference DynamoRIO* yang merupakan modifikasi khusus untuk target program hanya memerlukan informasi sederhana dari arus instruksi berjalannya sebuah program karena informasi lainnya sudah terdapat pada pustaka analisis yang dibuat.

II.9 Profiling

Profiling dalam informasi teknologi adalah aktivitas mengumpulkan informasi penggunaan *resources* dan waktu yang diperlukan untuk menjalankan suatu aplikasi. *Profiling* umumnya bertujuan untuk menemukan apa yang menyebabkan performansi aplikasi menjadi lamban. Keluaran yang dihasilkan

dapat berupa penggunaan instruksi, panggilan fungsi diikuti waktu eksekusi hingga aplikasi berhenti dalam bentuk teks maupun visualisasi grafik. *Profiling* dapat dilakukan dengan kode instrumentasi yang ditanam pada biner program atau pada level kode sumber.

Peristiwa yang terjadi pada program dapat diukur dengan statistik atau *benchmarks.* Dengan memberikan kerja berat pada aplikasi, *benchmarking* lebih memberikan informasi maksimal. Variasi kondisi membuat *benchmarking* dan *profiling* menjadi dua hal yang berbeda. Aktivitas *profiling* pada dasarnya tidak dibebani tugas yang berat melainkan tingkah laku pada status normal.

II.10 Tracing

Tracing dalam istilah pemograman merupakan salah satu teknik *debugging* yang digunakan untuk memahami proses berjalannya sebuah program. Tracing sering kali dikomparasikan dengan *logging* karena berupa catatan seperti file apa saja yang dibuka, pustaka apa saja yang dipanggil dan sebagainya. Terdapat beberapa level dalam mekanisme *trace* yaitu *Verbose, Debug, Info, Warning, Error, Fatal* dan *Silent. Buffer* yang ditampilkan juga melingkupi bagian *Main, System, Radio, Events* bahkan *Crash.*

Keluaran *trace* paling popular adalah *backtrace* atau *stacktrace* dimana beberapa *stack frame* yang menjadi masalah pada program dikembalikan sampai ke pangkal fungsi utama aplikasi. *Stacktrace* pada bahasa pemograman tingkat tinggi di-*handle* oleh interpreter dan berisi poin baris kode yang menginterpretasikan referensi masalah sedangkan, bahasa tingkat rendah memerlukan simbol *debug* atau jika tidak ada, keluaran berupa bahasa mesin atau *assembly.*

II.11 Binary Analysis

Analisis biner adalah sebuah pendekatan membongkar program dengan *disassembly* atau dekompilasi untuk tujuan *reverse engineering.* Pekerjaan ini dilakukan ketika analis tidak memiliki sumber kode. Aktivitas analisis biner meliputi *hex editing* hingga membuat *flow- graph.* Beberapa alat analisis biner

mendukung terjemahan yang sederhana ke dalam *byte code* atau emulasi sebagai *binary translation.* Hal tersebut berkenaan bahasa mesin atau *assembly* sulit dimengerti terutama pada aplikasi berskala besar yang mempunyai jutaan instruksi.

Biner program juga dapat direpresentasikan dalam bahasa tingkat menengah namun, bahasa tersebut sangat independen tergantung *compiler* yang mengkompilasinya. Alat analisis biner berperan langsung dalam dekompilasi ke bahasa independen tersebut. Analisis biner juga dapat dilakukan secara dinamis dengan mengeksekusi program pada *real* dan *virtual processor*. Terminologi analisis biner sangat dekat dengan *runtime analysis* karena keterbatasan analisis statis yang tidak mumpuni untuk perangkat lunak berproteksi.

II.12 Program Lifecycle Phase

Fase umum dari hidupnya sebuah perangkat lunak dari sumber kode hingga menjadi biner yang dijalankan meliputi:

II.12.1 Edit Time

Edit time adalah fase dimana ketika kode program sedang diubah artinya status program bisa jadi belum ditahap konsisten. Perubahan biasanya dilakukan oleh *programmer* namun, dapat pula dilakukan oleh *generator*, alat desain atau dengan sistem *meta-programming.* Contohnya seperti pengecekan *syntax* atau analisis sumber kode maupun merubah bahasa pemograman satu ke yang lainnya.

II.12.2 Compile Time

Compile time adalah fase setelah sumber kode diterjemahkan kedalam bahasa mesin oleh *compiler.* Pada tahap kompilasi terjadi pengecekan atas konsistensi kode yang diubah dan menghasilkan sebuah file *executable.* Untuk bahasa pemograman tingkat tinggi yang menggunakan interpreter, kompilasi berupa merubah sumber kode menjadi *bytecode.*

II.12.3 Link Time

Link time adalah fase menghubungkan pustaka dengan program yang menginvokasinya. Koneksi tersebut dapat diikat secara statis maupun dinamis

atau bahkan ketika eksekusi. *Linking* kerap kali dikaitkan dengan kompilasi walaupun sebenarnya tahap ini terpisah. *Linking* juga merupakan penggabungan beberapa file objek menjadi satu buah *executable.*

II.12.4 Load Time

Load time adalah fase ketika file *executable* dipanggil dan ditempatkan dalam memori yang aktif sebagai bagian dari mulainya eksekusi. Sederhanya adalah tahap dimana program pertama kali dijalankan yang mana sudah tidak lagi berupa file melainkan menjadi kesatuan atau pengalamatan *interface* pada kernel.

II.12.5 Run Time

Run time adalah fase dimana program sedang berjalan normal maupun tidak normal seperti *hang.* Di tahap ini program berada di status sudah mengakuisisi memori dan dapat mulai bekerja secara *multi threading* atau menciptakan proses *child.*

II.13 Binary Translation

Binary Translation adalah proses menerjemahkan kode mesin (biner) dari satu set instruksi ke instruksi lainnya. Sistem yang menggunakan hal tersebut direferensikan sebagai *emulator*. Kebanyakan *Binary Translation* mempunyai fungsi alat migrasi untuk transisi arsitektur lama ke yang baru. Penerjamahan biner dapat dilakukan secara statis (seluruh biner ke target platform) maupun dinamis dengan menerjemahkan kode saat eksekusi secara parsial berdasarkan fungsi yang diinginkan [3].

Beberapa *Dynamic Binary Translation* menggunakan “fast return” atau *indirect branch/call prediction* dalam pendekatan mengindari *overhead.* Pada *processors* terbaru prediksi menggunakan dua level adaptif *predictor*. Instruksi tersebut berkontribusi lebih dari satu bit ke *history buffer*. Prosesor tanpa mekanisme tersebut secara sederhana memprediksi sebuah *indirect jump* ke lokasi yang sama terakhir kalinya. Contoh sederhana implementasi dari *Binary Translation* dapat ditemukan pada mesin virtualisasi seperti Qemu atau VMware.

II.14 Intermediate Representation

IR atau sering disebut bahasa tingkat menengah merupakan representasi program diantara sumber kode dan biner (*assembly).* Hal tersebut sangat independen tergantung pada *compiler* maupun *interpreter*-nya. Representasi dapat digambarkan melalui *graph* seperti AST (*Abstract Syntax Tree)* maupun biner dalam *bytecode.* B*ytecode* mempunyai tujuan *retargetable* untuk merekonstruksi kembali kedalam biner. Pada dasarnya IR memiliki contoh padanan menampilkan algortitma sebagai berikut:

```
if (x + 2 > 5)
  y = 2;
else
  y = 3;
x++;
```

```
    t = x + 2
    if t > 5 goto l1
    y = 3
    goto l2
l1: y = 2
l2: x = x + 1
```

II.15 Reverse Engineer

Reverse engineer dalam dunia komputer adalah orang yang mempunyai tugas merekayasa hasil objek baik perangkat keras maupun lunak untuk mengubah atau mengembalikkan informasi sumbernya. Pada biner aktivitasnya meliputi membaca kode mesin *assembly* untuk program *executable* dan membaca *hexademical* untuk objek lainnya seperti gambar atau video yang kemudian direplikasi atau *patching* menjadi sesuai keinginan *reverse engineer.*

Terdapat bermacam-macam teknik *reverse engineering* untuk perangkat lunak diantaranya melakukan analisis secara statis(*disassembly)* langsung pada biner program hingga memasukkanya dalam ruang lingkup *debugger*. Beberapa *reverse engineer* melakukan instrumentasi atau menyisipkan program bantuan untuk melakukan analisis secara otomatis dan bermain di area memori. Untuk *reverse engineer* yang tidak biasa dengan hal tersebut biasanya menggunakan *intermediate representation* sebagai acuan dalam mengolah data yang menjadi optimasi.