BAB III
ANALISIS SELF-MODIFYING CODE DAN PERANCANGAN DYNAMIC BINARY INSTRUMENTATION

III.1 Analisis Sistem

Subbab analisis ini membahas mengenai analisis masalah, analisis biner program, analisis kebutuhan nun fungsional, analisis teknik *Self-Modifying Code* dan perancangan *Dynamic Binary Instrumentation.*

III.1.1 Analisis Masalah

Masalah yang menjadi latar belakang pada penelitian ini adalah bagaimana menganalisis *Self-Modifying Code* setelah dikompilasi ke dalam biner program untuk memahami cara kerjanya menggunakan metode *DBI(Dynamic Binary Instrumentation).* Adapun spesifikasi masalah adalah pengaruh *Self-Modifying Code* sebagai usaha anti analisis dalam membuat proses *debugging* menjadi sulit, kendala *Dynamic Binary Instrumentation* dalam menganalisis program serta konfigurasi *environment* yang berbeda-beda.

III.1.2 Analisis Biner Program

Analisis Biner adalah proses menganalisis dan menggambarkan bagaimana program dalam bekerja menjalankan tugasnya. Teknik analisis biner memiliki ruang lingkup yang luas. Selain meliputi biner yang melibatkan sistem operasi, analisis biner juga memodelkan aspek–aspek yang terkait dengan instruksi independen arsitektur. Instruksi yang perlu diketahui untuk menganalisis program *Self-Modifying Code* adalah transfer kontrol tidak kondisional seperti:

Tabel III.1 Transfer Kontrol

Instruksi	Keterangan
JMP	JMP(Jump) adalah transfer kontrol satu arah dan tidak menyimpan alamat kembali di *stack*

CALL	CALL(Call Procedure) mendorong alamat kembali ke *stack* dan transfer kontrol ke prosedur
RET	RET(Return From Procedure) memunculkan alamat kembali dari *stack* dan mengembalikan kontrol ke lokasi tersebut

Tidak ada urutan standar dalam analisis biner karena setiap proses *debugging* adalah seni. Adapun langkah-langkah yang diambil untuk analisis biner *Self-Modifying Code* dengan *Dynamic Binary Instrumentation* adalah sebagai berikut:

1. Memeriksa fungsi *de-obfuscation routine* dengan *disassembly* secara statis. Hal ini dapat dilakukan dengan alat seperti *objdump* maupun *debugger* seperti GDB.
2. Setelah mengetahui fungsi dimana terjadi *Self-Modifying Code*, dilakukan pembangunan instrumentasi untuk memonitoring fungsi tersebut.
3. Selanjutnya program *Self-Modifying Code* dijalankan dengan invokasi *instrumentor* dalam hal ini *DynamoRIO*
4. Setelah hasil analisis berupa keluaran *intermediate reprsesentation* dilanjutkan dengan membuat *script* sederhana atau *patcher* untuk memecahkan *Self-Modifying Code* tersebut
5. Membuat laporan cara kerja dengan teliti apa yang sebenarnya dilakukan oleh kesuluruhan program

III.1.3 Analisis Teknik Self-Modifying

Self-Modifying Code yang dianalisis merupakan perspektif telah menjadi program *executable* artinya tidak ada akses untuk sumber kode. Informasi yang tersedia hanya kode *assembly* pada biner dimana setiap *routine* harus diidentifikasi terlebih dahulu untuk mendapatkan fungsi yang mengolah modifikasi. Terdapat tiga kasus yang akan diuji diantara lain *Self-Checksumming, Self-Decrypting* dan *Self-Extracting.*

1. Self-Checksumming

Program *Self-Modifying* yang memverifikasi keaslian kode selama eksekusi terdapat sisipan produser kode perhitungan *checksum*. Sebuah algoritma *checksum* menghitung *hash* melewati berbagai kode kritis yang tidak boleh diubah. Verifikasi tersebut membandingkan hasil perhitungan *checksum* terhadap nilai yang tertera secara *hard-coded* dalam fungsi program. Kegagalan verifikasi menunjukkan dugaan modifikasi dari program dimana integritas tidak lagi terjamin.

Dimisalkan sebuah program yang dapat dijalankan hanya dengan adanya lisensi yang valid. Program harus dapat melindungi modifikasi kode yang melakukan cek lisensi. *Reverse engineer* mungkin menghapus fungsi pengecekan lisensi tersebut dengan cukup memasukkan instruksi *jump* pada kode dimana cek lisensi berada. Artinya pengecekan dilewati atau tidak dieksekusi sama sekali. Adapun contoh *self-checking checksum* dapat dilihat pada *pseudocode* berikut:

```
Selfcheck(checksum,     nonce,      codeStart,  codeEnd,
      codeSize)   {
      while (iteration  <     2500000)
      {
            checksum[0] +=    nonce;
            checksum[1] ^=    DP;
            checksum[2] +=    *DP;
            checksum[4] ^=    EIP;
            mix(checksum);
            nonce +=    (nonce*nonce)     |     5;
            DP    =     codeStart   +     (nonce      %
      codeSize);
            iteration++;
      }
}
```

Nonce mempunyai nilai acak(*integer)* yang tidak pernah sama setiap kali eksekusi. Bagian *codeStart*, *codeEnd* dan *codeSize* didefinisikan pada seksi header program yang diperoleh ketika kompilasi.

2. Self-Decrypting

Idealnya perangkat lunak yang mendekripsi bagian dirinya hanya berisi *decryptor stub* (informasi yang digunakan untuk mendekripsi) dan kode yang terenkripsi. Ada banyak varian dari metode enkripsi, beberapa diantaranya cendrung statis (dengan kunci dekripsi) dan yang lainnya dengan *polymorphic* atau mengubah kunci dekripsi. Hal ini dilakukan dalam rangka untuk membuat ruang lingkup *debugger* atau emulasi pada *antivirus* berhenti. Adapun klasifikasi enkripsi biner dibedakan berdasarkan bagaimana kode tersebut didekripsi yaitu:

1. Dengan kunci statis (dekripsi *routine* sangat mudah dimengerti)
2. Kunci yang berubah-ubah sewaktu-waktu(*looping* dengan *increment* +1, mudah dimengerti namun sedikit kerja ekstra untuk *reverse engineer)*
3. Tanpa kunci atau hanya operasi bit (contoh instruksi *circular shift* atau penggeseran seperti not, rol, ror, x-bit *swap)*
4. Kunci dengan eksternal dependensi(Berganti terus menerus dan tidak mempunyai pola spesifik seperti pada pengalamatan modul poin entri atau alamat virtual di memori untuk *executable)*
5. Tanpa kunci sama sekali(kode dekripsi didapatkan dengan *brute force)*

Cryptanalysis dapat dilakukan dengan menggunakan metode *Stream Cipher Attack* atau memanfaatkan informasi yang muncul berkali-kali. Adapun kasus *polymorphic self-decryption* dapat dilihat pada *pseudocode assembly* berikut:

```
LEA SI,...
MOV SP,0682h
JUMP:
XOR WORD PTR [SI],SI
XOR WORD PTR [SI],SP
INC SI
DEC SP
JNZ JUMP
```

Berdasarkan kode diatas kunci dekripsi akan berganti dengan *multiple execution* dikarenakan *stack pointer* diinilisiasi sebelum *looping.* Berdasarkan pengamatan dekripsi *routine* tersebut seharusnya kode enkripsi menggunakan

byte,word dan *dword* sebagai kunci atau kode metadata seperti alamat *relative* dan *absolute* pada memori sebagai kuncinya untuk enkripsi yang lebih baik.

3. Self-Extracting

Sederhananya *Self-Extracting* melakukan *unpacking* satu atau lebih segmen yang mengandung data konstan ke dalam blok yang dialokasikan pada memori *executable* dan kemudian memulai proses langsung melompat pada kode sebenarnya. Dengan kata lain sama seperti *self-decrypting* dimana terdapat *decompressor* dan *program loaders. Self-Extracting* dibangun menggunakan *packers* yaitu UPX(*Ultimate Packer for eXecutable).*

Ketika *packing executable* menggunakan UPX, semua seksi seperti .text dan .data dikompresi dengan penamaan UPX0, UPX1 dan seterusnya. Kemudian UPX menambahkan seksi kode diakhir yang mengandung keseluruhan dekompresi saat eksekusi. Berikut ini langkah-langkah yang dilakukan oleh UPX:

1. Eksekusi dimulai dengan OEP(*Original Entry Point)* baru
2. Menyimpan status *register* menggunakan instruksi PUSHAD
3. Semua *packed* atau pemadatan di-*unpack* dalam memori
4. Memuat kembali alamat *import table* orisinil dari file *executable*
5. Memuat kembali status *register* asli menggunakan instruksi POPAD
6. Terakhir *jump* ke OEP untuk memulai eksekusi *executable* sebenarnya

III.1.4 Analisis Binary Translation

Program yang emodifikasi bagian dirinya menimbulkan masalah ketika kode yang telah diterjemahkan berganti menjadi program asing. Jika *binary translators* tidak menyadari akan hal tersebut, *binary translator* akan mengeksekusi kode asal saat sebelum diterjemahkan. Analisis *Binary Translation* dilakukan berkenaan dengan desain keluaran yang akan dihasilkan berupa representasi bahasa pemograman tingkat menengah.

Performansi pada proses translasi biner juga patut diperhatikan. Adapun konsep yang dapat mempengaruhi proses translasi biner menjadi cepat atau bahkan lamban adalah sebagai berikut:

1. Looping Translation

   Proses penerjemahan berjalan pada unit dasar blok. Sebuah blok dasar adalah aliran langsung instruksi dengan instruksi cabang (JMP, call, dll) di *tail* namun bukan di dalam fungsi. Sebuah blok dasar diterjemahkan dapat langsung dijalankan pada CPU. Ketika instruksi *branch* terakhir dijalankan, kemudian ditentukan apakah target *branch* sudah diterjemahkan. Jika iya, *Translator* hanya bisa melompat disana. Jika tidak, *Translator* harus menerjemahkan blok dasar pertama. Panggilan untuk mengirimkan trampolin ditambahkan ke setiap blok dasar yang diterjemahkan.

```
for (;;) {
    if (!(translated_pc = find_translation(pc))
        translated_pc = translate_basic_block(pc);
    execute(translated_pc);
}
```

2. Direct Branch

   Branch langsung adalah jika target dapat ditentukan pada saat penerjemahan. Misalnya, memanggil fungsi tetap merupakan *branch* langsung. Jika hanya ada satu target yang deterministik untuk *branch* maka penerjemahan akan sangat mudah untuk dioptimalkan. Ketika menemukan sebuah *branch* langsung dilakukan pemeriksaan terlebih

dahulu apakah sudah dijalankan. Jika iya, *binary translator* dapat dengan mudah meng-*encode* target *branch*. Jika tidak, bangun trampolin seperti:

```
push $patch_addr
push $target_pc
jmp dbt_find_direct_internal
```

Dimana patch_addr adalah pointer ke target *branch* langsung dalam instruksi *branch* asli. dbt_find_direct_internal adalah fungsi perakitan untuk menyimpan dan memulihkan konteks program. Ketika fungsi tersebut dipanggil selanjutnya adalah menerjemahkan blok dasar yang hilang dengan menerjemahkan translated_pc ke patch_addr. Oleh karena itu ketika instruksi *branch* asli ditemui kedua kalinya, langsung melompat ke target yang benar tanpa *overhead*.

3. Indirect Branch

   Karena *Branch* tidak langsung tidak dapat ditentukan pada saat penerjemahan maka, diselesaikan pada saat *runtime*. Efektivitasnya langsung mempengaruhi kinerja sistem DBT.

4. Return Handling

   Instruksi *ret* adalah bentuk khusus dari percabangan tidak langsung yang sangat mudah diprediksi. Beberapa sistem DBT menggunakan pendekatan "*fast return*" untuk meminimalkan terjadinya *overhead*. Ketika *call* dijalankan, *binary translator* mendorong alamat pengirim yang diterjemahkan, bukan alamat pengirim asli untuk stack. Masalah utama dari pendekatan ini adalah tidak transparan untuk aplikasi pokok, dan fitur tertentu seperti *exception handling* akan gagal jika alamat kembali diubah.

DynamoRIO menerapkan teknik yang disebut "*return cache*". Kembalinya *cache* bermaksud tabel hash terdiri dari pemetaan dari alamat kembali. Setiap *hash bucket* memiliki hanya satu entri. Ketika instruksi panggilan dijalankan, pc yang diterjemahkan dari alamat pengirim ditulis ke *hash table bucket*. Ketika instruksi kembali dijalankan, alamat pengirim di stack diiris dan melompat masuk ke *hash* yang sesuai. Alamat kembali *cache* mungkin dapat terjadi kesalahan.

Untuk menangani kasus ini, sebuah *stub* sederhana ditambahkan setelah instruksi panggilan.

1. Call return pair optimization

   CPU modern memiliki mekanisme prediksi *branch* yang maju untuk meningkatkan efisiensi eksekusi. Salah satu optimasi tersebut disebut "*return stack buffer*". Hal tersebut merupakan *ring buffer* yang mirip dengan alamat kembali *cache*. Ketika instruksi *call* dijalankan, alamat kembali dimasukkan ke *buffer*. Ketika instrruksi ret dijalankan, tumpukan atas muncul dan digunakan sebagai prediksi alamat target. Untuk mengambil keuntungan dari ini, instruksi *call* dan *ret* harus dalam pasangan. Dalam sistem penerjemahan biner tradisional dinamis dua instruksi tersebut biasanya diimplementasikan menggunakan instruksi *jmp* ke berbagai trampolin yang benar-benar hilang optimasinya dari *return stack buffer*.

2. CPUID Emulation

   Karena setiap instruksi harus secara eksplisit diharapkan agar bekerja seluruhnya. Informasi CPUID tidak dapat ditinggalkan dari *host* langsung ke target aplikasi. Pertimbangan dukungan instruksi khusus seperti AVX langsung menggunakan CPUID asli yang membuat aplikasi berpikir dapat menggunakan instruksi AVX. Oleh karena itu *binary translator* harus mencegat instruksi CPUID dan menutupi keluar fitur tersebut.

III.1.5 Analisis Sintaksis

Parsing atau *Syntatic Analysis* menggunakan DCFG(Dynamic Control-Flow Graph) yang mana menambahkan data dari sebuah spesifik eksekusi program. Biasanya CFG didefinisikan secara statis dan tidak mengandung informasi tentang jalur eksekusi dari setiap beban kerja tertentu. Adapun kelebihan DCFG antara lain:

1. Sebuah DCFG berisi node awal yang tidak memiliki *node* pendahulu dan pengganti *node*-nya berisi instruksi pertama yang dieksekusi per *thread*. Hal ini juga berisi *node* akhir yang tidak memiliki *node*

penerus dan pendahulunya berisi instruksi terakhir yang dieksekusi per *thread*.

2. Setiap *edge* DCFG ditambah dengan jumlah yang dinamis untuk menunjukkan berapa kali dilalui per *thread* oleh eksekusi program yang diberikan. Kecuali untuk awal dan akhir *node*, hitungan dinamis setiap *node* adalah sama dengan jumlah dari semua *edge* yang masuk, yang juga sama dengan jumlah dari semua *edge* keluar nya.
3. Sebuah DCFG tidak perlu mengandung *node* atau *edge* yang tidak dieksekusi. Sebuah DCFG diperbolehkan mengandung *node* dan *edge* yang tidak dieksekusi atau hanya akan memiliki jumlah nol.
4. Sebuah DCFG berisi *edge* yang mewakili semua jalur kode eksekusi sebenarnya, bahkan untuk *non-control-flow*. Sebagai contoh, sebuah instruksi *floating-point* yang menyebabkan pengecualia selama eksekusi dapat menciptakan keunggulan untuk kode *exception-handling*.

Blok di DCFG dapat dikombinasikan menjadi tingkat yang lebih tinggi untuk konstruksi seperti perulangan, rutinitas dan biner *images*. Data dinamis seperti jumlah iterasi *loop* dapat disimpulkan dari jumlah *edge* dasar. Format DCFG memungkinkan penyimpanan informasi pada proses, termasuk images konstituen, simbol, data debug, dan kontrol.

Jenis dari elemen dideskripsikan menggunakan format JSON yang mempunyai list untuk setiap strukturnya. Hal tersebut nantinya disimpan sementara dalam memori untuk mempermudah penerjemahan biner dalam hal parsing. Adapun spesifikasinya sebagai berikut:

1. Top-level structure

   Nilai dari *Top-level* adalah sebuah objek atau urutan pasangan nilai kunci. Bagian utama dari aliran DCFG ditandai dalam objek tingkat atas yang terdaftar secara singkat dibawah dan diperluas dalam sisa dokumen berikut ini:

1. Versi format
2. Daftar pengidentifikasi (id) digunakan untuk referensi *string* kemudian di-*stream* seperti nama file, jenis *edge* dan *node* khusus.
3. Data proses berisi id, jumlah instruksi per-*thread*, serta daftar *images* dari biner seperti alamat dan ukuran *Load*, daftar simbol, sumber berkas, garis nomor debug data, daftar *node* blok dasar, daftar rutinitas dan *loop* konstituen.

2. Versi

Versi mayor dan minor dari format file terdaftar sebagai dua tag terpisah di objek tingkat atas. Maksudnya adalah bahwa format DCFG akan kemabali dan maju secara kompatibel sebanyak mungkin. Kompatibilitas berarti bahwa versi ke depan tidak harus mengubah *tag* yang telah ditentukan atau menghapus data non-opsional. Kompatibilitas ke depan berarti bahwa penambahan harus dilakukan *parser* untuk format yang sudah ada harus mampu membaca format ke depan jika mengabaikan *tag* yang diketahui.

Jika kompatibilitas rusak, atau jika fitur baru utama ditambahkan, nomor versi utama harus bertambah dan nomor versi minor ulang ke nol. Jika tidak, perubahan format lain dibuat nomor versi minor yang bertambah. Ketika dikonversi ke teks, versi mayor dan versi minor dipisahkan oleh sebuah titik (".") dan versi minor harus ditampilkan sebagai dua digit yang memungkinkan hingga versi 99 tanpa kebingungan karena nilai tempat desimal, misalnya 2.03 mendahului 2.12.

3. Nama File

Daftar nama file yang digunakan untuk menghemat ruang adalah dengan mengizinkan nama file direferensikan nanti oleh pengidentifikasi bilangan bulat, bukan string panjang yang mungkin perlu diulang berkali-kali. Adapun yang harus diperhatikan untuk nama file adalah:

1. Seperti layaknya semua string, karakter khusus harus lolos memenuhi format string JSON.
2. Parsers tidak boleh berasumsi bahwa id mulai dengan satu atau berurutan dalam pemesanan tertentu.
3. Semua nama file yang digunakan dalam aliran DCFG disimpan dalam tabel yang sama, termasuk untuk images dan file sumber.

4. Jenis Edge

Daftar jenis *edge* digunakan untuk menghemat ruang dengan mengizinkan jenis *edge* untuk kemudian direferensikan dengan pengenal bilangan bulat, bukan *string* yang seharusnya perlu diulang berkali-kali. Adapun jenis *string edge* yang telah ditentukan dapat dilihat pada tabel berikut:

Tabel III.2 Nilai String Jenis Edge

Nilai String Jenis Edge	Keterangan
ENTRY	*Edge* pertama dilalui dalam DCFG yang menjadi tanda dari mulainya *node* ke blok dasar pertama eksekusi
EXIT	*Edge* terkahir dilalui dalam DCFG yang menjadi tanda dari berakhirnya *node* ke blok dasar akhir eksekusi
BRANCH	Mewakilkan semua *branch* yang tidak diketahui
CONDITIONAL_BRANCH	Mewakilkan semua jenis *conditional branch* baik *direct* maupun *indirect*
UNCONDITIONAL_BRANCH	Mewakilkan semua jenis *unconditional branch* baik *direct* maupun *indirect*
DIRECT_BRANCH	Sebuah *direct* untuk *conditional branch*
INDIRECT_BRANCH	Sebuah *indirect* untuk *unconditional branch*

DIRECT_CONDITIONAL_BRANCH	Sebuah *direct* untuk *conditional branch*
INDIRECT_CONDITIONAL_BRANCH	Sebuah *indirect* untuk *conditional branch*
DIRECT_UNCONDITIONAL_BRANCH	Sebuah *direct* untuk *unconditional branch*
INDIRECT_UNCODITIONAL_BRANCH	Sebuah *indirect* untuk *unconditional branch*
REP	Sebuah *Edge* yang mengindikasikan sebuah instruksi dengan prefix "REP" atau *repeated.*
FALL_THROUGH	Sebuah *Edge* yang mengindikasikan instruksi pada alamat selanjutnya telah dieksekusi.
CALL	Mewakilkan semua jenis *call* yang tidak diketahui
DIRECT_CALL	Sebuah *direct call* yang mempunyai satu alamat statis
INDIRECT_CALL	Sebuah *indirect call* yang mempunyai satu alamat dinamis
RETURN	Sebuah *return* biasanya dipanggil dari *routine*
CALL_BYPASS	Mengindikasikan *control flow* dalam sebuah *routine* tanpa diikuti *call*
SYSTEM_CALL	Sebuah *call* khusus ke sistem *routine*
SYSTEM_RETURN	Sebuah *return* dari sistem *routine*
SYSTEM_CALL_BYPASS	Sama seperti CALL_BYPASS akan tetapi untuk sistem *routine*
CONTEXT_CHANGE	Sebuah *Edge* yang dibuat dari semua instruksi *non-control-flow*
CONTEXT_CHANGE_RETURN	Sebuah *return* dari *handler context-change*
CONTEXT_CHANGE_BYPASS	Sama seperti CALL_BYPASS akan tetapi untuk *handler context-change*
EXCLUDED_CODE_BYPASS	Sebuah *Edge* yang digunakan untuk memlihara *control flow* yang tidak termasuk wilayah kode dari CFG

UNKNOWN	Mewakilkan semua jenis *Edge* yang tidak diketahui

5. Node Khusus

Daftar node khusus digunakan untuk menghemat ruang dengan mengizinkan node yang akan kemudian direferensikan oleh pengidentifikasi bilangan. Berikut yang harus diperhatikan adalah:

1. Parser tidak boleh berasumsi bahwa id dimulai dengan satu atau berurutan dalam urutan tertentu.
2. Parser tidak boleh berasumsi bahwa id tetap atau sama antara DCFG. Misalnya *node* "START" mungkin memiliki id=1 dalam satu DCFG dan id=3 di lain.

6. Data Proses

Semua data untuk proses disimpan dalam objek yang muncul di kolom PROCESS_DATA dari tabel PROSES. Kunci dalam objek dapat dilihat pada table sebagai berikut:

Tabel III.3 Proses Objek

Proses Objek	Nilai
INSTR_COUNT	Integer: jumlah instruksi untuk proses yang melewati seluruh *thread*
INSTR_COUNT_PER_THREAD	Array: Setiap *integer* dari setiap jumlah insruksi untuk proses per-*thread*
IMAGES	Tabel *image*
EDGES	Tabel edge

7. Images

Semua *images* untuk suatu proses yang disimpan dalam tabel muncul setelah kunci IMAGES dalam proses data objek. Adapun klasifikasinya sebagai berikut:

Tabel III.4 Images

Heading	Nilai
IMAGE_ID	*Integer* dapat dimulai dengan nol
LOAD_ADDR	Sebuah *integer* yang mengandung alamat *base* yang dimuat oleh sistem operasi
SIZE	Sebuah *integer* yang mengandung ukuran *image*
IMAGE_DATA	Sebuah objek JSON yang mengandung data *image*

8. Simbol

Semua simbol untuk *images* yang diberikan, disimpan dalam tabel yang muncul setelah kunci SYMBOLS dalam data objek *images*. Adapun klasifikasinya sebagai berikut:

Tabel III.5 Simbol Images

Heading	Nilai
NAME	*String* yang mengandung nama simbol
ADDR_OFFSET	*Integer* yang mengandung alamat *base* dari relativitas simbol ke alamat *load* yang mengandung *image*
SIZE	Sebuah *integer* yang mengandung ukuran simbol dalam *bytes*

9. Informasi Sumber

Semua informasi sumber debug *images* yang diberikan, disimpan dalam tabel yang muncul setelah kunci SOURCE_DATA dalam data objek *images simbol*. Adapun klasifikasinya sebagai berikut:

Tabel III.6 Informasi Sumber

Heading	Nilai
FILE_NAME_ID	*Integer* id dari nama file tabel yang mempresentasikan file sumber kode dari data *debug*
LINE_NUM	Sebuah *integer* yang mempresentasikan garis nomor file referensi sumber kode
ADDR_OFFSET	Sebuah *integer* yang mengandung alamat *base* dari relativitas data *debug* yang mengandung *image*
SIZE	Sebuah *integer* yang mengandung ukuran data dalam *bytes*
NUM_INSTR	Sebuah *integer* yang mengandung angka dari instruksi oleh data *debug*

10. Blok Dasar

Semua blok dasar untuk *images* yang diberikan, disimpan dalam tabel yang muncul setelah kunci BASIC_BLOCKS dalam *images* objek data informasi sumber. Sebuah blok dasar hanya memiliki satu poin entri dan satu titik keluar. Dasar blok didefinisikan secara dinamis, sehingga dapat bervariasi dari *run-to-run* pada beban kerja yang sama, hal ini bergantung pada *edge* dalam CFG yang dilalui. Variasi tersebut dapat mencakup tidak hanya blok dasar yang disertakan tetapi juga awal dan akhir alamat masing-masing. Adapun klasifikasinya dapat dilihat pada tabel berikut:

Tabel III.7 Blok Dasar

Heading	Nilai
NODE_ID	Sebuah *integer* id dan harus unik menyeluruh pada proses, tidak hanya *image*
ADDR_OFFSET	Sebuah *integer* yang mengandung instruksi pertama dalam relativitas blok dasar ke alamat *load* dari *image*
SIZE	Sebuah *integer* yang mengandung ukuran blok dala *bytes*
NUM_INSTRS	Sebuah *integer* yang mengandung angka instruksi dalam blok
LAST_INSTR_OFFSET	Sebuah *integer* yang mengandung alamat instruksi akhir dalam relativitas blok dasar ke alamat pertama instruksi
COUNT	Sebuah *integer* yang mengandung jumlah angka dari waktu blok yang telah dieksekusi menyeluruh pada *thread*

11. Routines

Semua informasi rutin untuk *images* yang diberikan, disimpan dalam tabel yang muncul setelah kunci ROUTINES dalam data objek *images* blok dasar. Rutinitas harus mengikuti aturan berikut:

1. Sebuah *routine* dapat memiliki hanya satu entri (routine dapat memiliki beberapa poin exit)
2. Setiap node dalam *routines* harus dicapai dari entri *node.* Hal ini berarti bahwa setiap *node* harus menjadi penerus langsung (target *edge*) dari entri *node* atau node lain dalam *routines*. *Edge* antar *routine* (*call*, *returns*, *context changes*, dll) tidak diperbolehkan dalam traversal tersebut. Namun, setiap *node* yang dicapai dari entri *node* tidak diperlukan untuk berada di *routines* yang sama. Hal tersebut memungkinkan untuk pembagian routines ketika ada beberapa node masuk, melompat antara *routines* dan kontrol aliran non-standar lainnya.

3. Setiap *node* dalam *routines* harus dalam *images* yang dikandung.
4. Setiap simpul dari suatu *images* (kecuali awal dan akhir) akan muncul di tepat satu *routine*.

Dengan demikian alamat dan ukuran *routines* tidak akan selalu sesuai dengan yang ada di tabel simbol. *Routines* juga dapat dimasukkan dengan *edge* selain *calls* dan dapat keluar selain *returns*. Adapun spesifikasinya dapat dilihat pada tabel berikut:

Tabel III.8 Routines

Heading	Nilai
ENTRY_NODE_ID	Sebuah *integer* id dari blok dasar
EXIT_NODE_IDS	*Array* dari id blok dasar yang mengindikasikan satu atau lebih *edges* yang meninggalkan *routine*
NODES	Tabel *node*

12. Loops

Semua informasi *loop* untuk *routine* disimpan dalam tabel yang muncul di bawah kolom LOOPS dalam tabel *routine*. *Loop* didefinisikan dengan algoritma standar *graph-traversal*. Kendala tersebut dapat diberlakukan sebagai berikut:

1. Sebuah *loop* hanya dapat memiliki satu *head node*
2. Sebuah *loop* dapat memiliki beberapa *back-edge*
3. Semua *node* dalam satu *loop* berada di *routines* yang dikandung

Tabel III.9 Head Loop

Heading	Nilai
LOOP_HEAD_NODE_ID	Sebuah *integer* dari blok dasar pada *head* di *loop*
LOOP_BACK_EDGE_SOURCE_NODE_IDS	Sebuah id *integer* blok dasar dari *nodes* yang mengandung *branches* kembali pada *head* di *loop*
LOOP_NODE_IDS	*Array* dari id blok dasar untuk semua *nodes* dalam *loop*
PARENT_LOOP_HEAD_NODE_ID	Sebuah id *integer* blok dasar dari *head* di *node* yang mengandung *loop* dalam *routine*

13. Edges

Semua informasi *edge* untuk proses tertentu disimpan dalam tabel yang muncul setelah tag EDGE dalam objek proses. *Edge* ditempatkan di luar data *images* karena *edge* sering keluar diantara *images* yang berbeda. Adapun spesifikasinya dapat dilihat pada tabel berikut:

Tabel III.10 Edges

Heading	Nilai
EDGE_ID	Sebuah id yang bernilai *integer*
SOURCE_NODE_ID	Sebuah *node* id khusus atau id blok dasar dari *control flow*
TARGET_NODE_ID	Sebuah *node* ide khusus atau id blok dasar ke *control flow*
EDGE_TYPE_ID	Sebuah ide *integer* yang merupakan jenis *edge*
COUNT_PER_THREAD	*Array* yang berisi *integer* dimana setiap *integer* berisi hitungan eksekusi untuk *edge* per-*thread*

III.1.6 Spesifikasi Intermediate Representation

Representasi menggunakan standar *universal* sebuah SSA (*Static Single Assignment*) dari LLVM karena ekspresif dalam *human readable assembly language.* Adapun spesifikasinya dapat diklasifikasikan sebagai berikut:

1. Identifiers

Identifier dibagi atas dua tipe dasar yaitu *global* dan *local.* Identifier *global (functions,* variabel *global)* dimulai dengan karakter '@'. Identifier *local (register names, types)* dimulai dengan dengan karakter '%'. Terdapat tiga format untuk identifier tergantung tujuan masing -masing:

1. Nilai yang mempunyai nama direpresentasikan sebagai string dengan *prefix*-nya. Contoh %foo dan @DivisionByZero. Identifier yang memerlukan karakter lain dalam penamaanya dikurung dengan tanda kutip dua. Karakter khusus di-*escape* menggunakan "\xx" dimana xx adalah kode ASCII untuk karakter dalam *hexademical. Prefix* "\01" dapat digunakan dalam variabel *global* untuk penekanan *mangling.*
2. Nilai yang tidak mempunyai nama direpresentasikan sebagai *unsigned numeric* dengan *prefix*-nya. Contoh %12 dan @2.
3. Konstan dibagi atas dua jenis yaitu *simple* dan dan *complex.* Konstan *simple* direpresentasikan dengan *Boolean, Integer, Floating Point, Null Pointer* dan *Token* sedangkan *complex* berupa Struktur, *Array, Vector, Zero Initialization* dan simpul *metadata.*

Prefix digunakan karena beberapa kata kunci seperti *add, bitcast, ret* untuk tipe data primitf seperti *void* dan *i32* tidak boleh konflik dengan variabel nama. Komentar dipisah dengan penandaan sebuah titik koma ';' terus sampai akhir baris.

2. Linkage Types

Semua variabel *global* dan fungsi mempunyai setidaknya satu dari tipe *linkage.* Hal ini dapat dilihat seperti pada tabel berikut:

Tabel III.11 Linkage Types

No	Linkage Type	Keterangan
1	private	Nilai *global* dengan *linkage* “private“ hanya dapat diakses langsung oleh objek – objek dalam modulnya
2	internal	Sama dengan “private“ akan tetapi nilai dimunculkan sebagai simbol lokal (STB_LOCAL pada ELF) dalam file objek
3	available_externally	Global dengan tanda “available_externally” mengizinkan *inlining* dan optimasi lainny sebagai pengetahuan definisi *global* diluar modul tersebut.
4	linkonce	Global dengan *linkage* “linkonce“ menandakan penggabungan dengan *global* lainnya yang mempunyai nama sama ketika *linkage* terjadi
5	weak	Secara semantik sama dengan *linkage* linkonce kecuali referensinya bisa jadi tidak dihapus
6	common	Sebagian besar mirip dengan “weak” tetapi *linkage* “common” digunakan untuk definisi sementara pada C seperti “int X;” pada skala global
7	appending	Global “appending” hanya diterapkan pada variabel *pointer* ke tipe array
8	extern_weak	Semantik *linkage* ini mengikuti model file objek pada ELF yang mana akan selalu *null* jika tidak *linked*
9	linkonce_odr	*Linkage* ini untuk mengindikasikan bahwa *global* akan selalu digabungkan bersama *global* yang setara lainnya
10	external	*Linkage* “external“ digunakan jika tidak ada satupun identifier yang dipakai, artinya dapat digunakan dalam menyelesaikan simbol eksternal

3. Visibility Styles

Seluruh variabel *global* dan fungsi pada ELF setidaknya mempunyai *visibility styles.* Hal tersebut dapat dilihat dalam tabel berikut:

Tabel III.12 Visibility Styles

No	Visibility Style	Keterangan
1	default	Deklarasi tampak untuk modul lainnya dalam *shared libraries*
2	hidden	Mengindikasikan bahwa simbol tidak akan ditempatkan dalam tabel simbol dinamis
3	protected	Menyatakan bahwa simbol akan ditempatkan dalam tabel simbol dinamis tetapi tidak dapat di-*override* oleh modul lain

Sebuah simbol dengan *linkage* internal atau private harus mempunyai "default" *visibility.* Hal ini dikarenakan setiap pustaka saling berkaitan satu sama lain

4. Comdats

Comdat adalah sebuah objek yang mengandung paket fungsi dan data. Pada *Intermediate Representation* berisi akses untuk COFF dan file objek ELF. *Comdat* memiliki seleksi jenis untuk menyediakan *input* dalam bagaimana *linker* harus memilih antara kunci dalam dua file objek berbeda.

Syntax:

$<Name> = comdat JenisSeleksi

Jenis seleksi setidaknya berupa satu parameter sebagai berikut:

Tabel III.13 COMDAT

No	Jenis Seleksi	Keterangan
1	any	*Linker* memilih kunci COMDAT yang mana saja secara acak.
2	exactmatch	*Linker* memilih kunci COMDAT yang mana saja tetapi seksi harus mengandung data yang sama
3	largest	*Linker* memilih seksi yang mengandung kunci COMDAT paling besar
4	noduplicates	*Linker* memilih kunci COMDAT yang paling unik pada seksi
5	samesize	*Linker* memilih kunci COMDAT yang mana saja tetapi seksi harus mengandung banyaknya data yang sama

5. Atribut Parameter

Setiap parameter sebuah tipe fungsi memiliki sebuah atribut parameter yang terkait satu sama lain. Atribut parameter digunakan untuk komunikasi informasi tambahan tentang hasil atau parameter sebuah fungsi. Atribut parameter dianggap sebagai bagian dari fungsi, bukan tipe fungsi, dengan kata lain fungsi – fungsi dengan atribut parameter berbeda dapat memiliki tipe fungsi yang sama. Atribut parameter didefinisikan sebagai berikut:

Tabel III.14 Atribut Parameter

No	Parameter	Keterangan
1	zeroext	Mengindikasikan ke kode generator bahwa parameter atau nilai kembali harus *zero-extended* untuk jangkauannya dimana diperlukan oleh *caller* untuk parameter dan *callee* untuk nilai kembali
2	signext	Mengindikasikan ke kode generator bahwa parameter atau nilai kembali harus *sign-extended* untuk jangkauannya dimana diperlukan oleh *caller* untuk parameter dan *callee* untuk nilai kembali

3	inreg	Mengindikasikan bahwa parameter atau nilai kembali harus dianggap spesifik dalam targetnya dengan menempatkannya dalam *register* sebagai penentang ke memori
4	byval	Mengindikasikan bahwa penunjuk parameter harus melewati nilai untuk menuju ke fungsi
5	inalloca	Atribut *inalloca* membolehkan *caller* untuk mengambil alamat pernyataan *stack* yang sedang berjalan
6	sret	Mengindikasikan parameter *pointer* menspesifikasikan alamat sebuah struktur yang mana nilai kembali merupakan fungsi dalam sumber program
7	align	Mengindikasikan bahwa nilai *pointer* diasumsikan oleh *optimizer* untuk mempunyai spesifikasi *alignment*
8	noalias	Mengindikasikan bahwa objek diakses dengan nilai – nilai *pointer* berdasarkan pernyataan atau nilai kembali
9	nocapture	Mengindikasikan bahwa *callee* tidak membuat salinan *pointer* yang dipakai lebih lama dari *callee* itu sendiri
10	nest	Mengindikasikan bahwa parameter *pointer* dapat dipotong menggunakan *trampoline instrincs*
11	returned	Mengindikasikan bahwa fungsi selalu mengembalikan pernyataan sebagai nilai kembali
12	nonnull	Mengindikasikan bahwa parameter atau *pointer* kembali tidak *null*
13	dereferenceable	Mengindikasikan bahwa parameter atau *pointer* kembali *dereferenceable*
14	Dereferenceable_or_null	Mengindikasikan bahwa parameter atau nilai kembali bukan *non-null* atau *non-dereferenceable* pada saat yang sama

6. Atribut Fungsi

Atribut fungsi adalah set komunikasi informasi tambahan tentang sebuah fungsi yang berupa kata kunci sederhana mengikuti spesifikasi tipe. Atribut dapat

berisi lebih dari satu dan diseperasikan dengan spasi. Atribut fungsi didefinisikan sebagai berikut:

Tabel III.15 Atribut Fungsi

No	Fungsi	Keterangan
1	alignstack	Mengindikasikan *backend* harus menyelaraskan *stack pointer* secara paksa
2	alwaysinline	Mengindikasikan *inliner* harus berusaha untuk *inline* fungsi kedalam *callers* kapanpun itu memungkinkan
3	builtin	Mengindikasikan fungsi *callee* pada sebuah panggilan harus dapat diakui sebagai fungsi *built-in*
4	cold	Mengindikasikan bahwa fungsi jarang dipanggil
5	convergent	Mengindikasikan bahwa *callee* bergantung pada pola eksekusi *thread* konvergen dibawah model eksekusi paralel tertentu
6	inlinehint	Mengindikasikan bahwa sumber kode mengandung petunjuk tersembunyi yang mengartikan fungsi tersebut diinginkan
7	jumptable	Mengindikasikan bahwa fungsi harus ditambahkan ke tabel instruksi *jump* pada saat generasi kode
8	minsize	Menunjukkan bahwa optimasi dan kode generator melewati pilihan yang menjaga ukuran kode fungsi sekecil mungkin
9	naked	Menonaktifkan emisi prolog atau epilog untuk fungsi
10	nobuiltin	Mengindikasikan bahwa fungsi *callee* tidak diakui sebagai fungsi *built-in*
11	noduplicate	Mengindikasikan bahwa pangilan – panggilan ke fungsi tidak dapat duplikasi
12	noimpicitfloat	Menonaktifkan instruksi *floating point* secara implisit
13	noinline	Mengindikasikan bahwa *inliner* tidak boleh sama sekali *inline* pada fungsi.
14	nonlazybind	Menekankan simbol *lazy binding* untuk fungsi

15	noredzone	Mengindikasikan bahwa generator kode tidak harus menggunakan sebuah *red zone* bahkan pada target spesifik ABI(*Application Binary Interface)* sekalipun
16	noreturn	Mengindikasikan bahwa fungsi tidak pernah kembali dengan normal
17	norecurse	Mengindikasikan bahwa fungsi tidak memanggil dirinya secara langsung maupun tidak langsung
18	nounwind	Mengindikasikan bahwa fungsi tidak pernah meraih eksepsi
19	optnone	Mengindikasikan bahwa fungsi tidak dioptimasi
20	optsize	Menunjukkan bahwa optimasi melewati pilihan yang menjaga ukuran kode tetap kecil
21	readnone	Mengindikasikasikan bahwa fungsi mengkalkulasi hasilnya berdasarkan pada argumen tanpa *dereferencing pointer* apapun
22	readonly	Mengindikasikan bahwa fungsi tidak menulis pernyataan pada *pointer* manapun
23	argmemonly	Mengindikasikan bahwa memori akses hanya membuka dan menyimpan didalam fungsi dari objek yang ditunjukkan oleh argumen *pointer-typed*
24	returns_twice	Mengindikasikan bahwa fungsi dapat kembali dua kali
25	safestack	Mengindikasikan bahwa proteksi *SafeStack* aktif
26	sanitize_address	Mengindikasikan bahwa pengecekan *AddressSanitizer* aktif
27	sanitize_memory	Mengindikasikan bahwa pengecekan *MemorySanitizer* aktif
28	sanitize_thread	Mengindikasikan bahwa pengecekan *ThreadSanitizer* aktif
29	ssp	Mengindikasikan bahwa fungsi mengeluarkan *stack smashing protector*
30	sspreq	Mengindikasikan bahwa fungsi harus selalu mengeluarkan *stack smashing protector*
31	sspstrong	Mengindikasikan bahwa fungsi selalu mengeluarkan *stack smashing protector*

32	thunk	Mengindikasikan bahwa fungsi akan mendelegasikan beberapa fungsi lain dengan panggilan *tail*
33	uwtable	Mengindikasikan bahwa ABI(*Application Binary Interface*) menjadi target

7. Atomic Memory Ordering Constraints

Instruksi *atomic* berperan mengurutkan parameter yang menentukan petunjuk *atomic* lain pada alamat yang sama dengan sinkronisasi. Kendala pengurutan didefinisikan sebagai berikut:

Tabel III.16 Ordering Constraints

No	Ordering	Keterangan
1	unordered	Set dari nilai – nilai yang dapat dibaca ketika sebelum maupun sesudah urutan parsial
2	monotonic	Sama seperti *unordered* akan tetapi *monotonic* memiliki pengecekan modifikasi harus kompatibel sebelum dan sesudah urutan
3	acquire	Sama seperti *monotonic* tetapi *acquire* menambahkan sinkronisasi dengan *edge*
4	release	Sama seperti *acquire* dengan penambahan jika operasi menulis sebuah nilai yangmana sub-sekuensi dibaca oleh sebuah operasi *acquire*
5	acq_rel	Bertindak sebagai sebuah operasi *acquire* dan *release* pada alamatnya
6	seq_cst	Sama seperti *acq_rel* dengan pengecekan bahwa *acquire* untuk sebuah operasi membaca sedangkan *release* merupakan sebuah operasi untuk menulis

Jika sebuah operasi *atomic* ditandai dengan *singlethread* maka, operasi hanya sinkronisasi dengan atau partisipasi dalam modifikasi jumlah pengurutan *seq_cst* pada operasi–operasi berjalan lainnya di *thread* yang sama. Contohnya pada peristiwa *signal handlers.*

8. Constant Expressions

Ekpresi konstan digunakan untuk memungkinkan ekspresi yang melibatkan konstanta sebagai konstan lainnya. Berikut ini adalah sintaks untuk ekspresi kontan:

Tabel III.17 Constant Expressions

No	Ekspresi	Keterangan
1	trunc	*Truncate* sebuah konstan ke tipe lain, ukuran bit CST harus lebih besar dari ukuran bit TYPE, hasilnya harus *integer*
2	zext	*Zero extend* sebuah konstan ke tipe lain, ukuran bit CST harus lebih kecil dari ukuran bit TYPE, hasilnya harus *integer*
3	sext	*Sign extend* sebuah konstan ke tipe lain, ukuran bit CST harus lebih kecil dari ukuran bit TYPE, hasilnya harus *integer*
4	fptrunc	*Truncate* sebuah *floating point* ke tipe *floating point* lain, ukuran CST harus lebih besar dari TYPE, hasilnya harus *floating point*
5	fpext	*Extend* sebuah *floating point* ke tipe *floating point* lain, ukuran CST harus lebih kecil dari TYPE, hasilnya harus *floating point*
6	fptoui	Konversi sebuah konstan *floating point* ke konstan *unsigned integer* , TYPE harus sebuah *scalar* atau tipe vektor *integer* sedangkan CST harus harus *scalar* atau tipe vektor *floating point*
7	fptosi	Konversi sebuah konstan *floating point* ke konstan *signed integer*, TYPE harus sebuah *scalar* atau tipe vektor *integer* sedangkan CST harus *scalar* atatu tipe vektor *floating point*
8	uitofp	Konversi sebuah konstan *unsigned integer* ke konstan *floating point*, TYPE harus sebuah *scalar* atau tipe vektor *floating point* sedangkan CST harus *scalar* atau tipe vektor *integer*
9	sitofp	Konversi sebuah konstan *signed integer* ke konstan *floating point*, TYPE harus sebuah *scalar* atau tipe vektor *floating point* sedangkan CST harus *scalar* atau tipe vektor *integer*
10	ptrtoint	Konversi sebuah konstan *pointer typed* ke konstan *integer*, TYPE harus

		sebuah bertipe *integer* sedangkan CST harus bertipe *pointer*
11	inttoptr	Konversi sebuah konstan *integer* ke konstan *pointer*, TYPE harus bertipe *pointer* sedangkan CST harus bertipe *integer*
12	bitcast	Konversi sebuah konstan dan CST ke TYPE lain
13	addrspacecast	Konversi sebuah *pointer* konstan atau vektor konstan dari *pointer, CST* ke TYPE lain dalam *address space* berbeda
14	getelementptr	Menjalankan kode operasi *getelementptr* pada konstan
15	select	Menjalankan kode operasi *select* pada konstan
16	icmp	Menjalankan kode operasi *icmp* pada konstan
17	fcmp	Menjalankan kode operasi *fcmp* pada konstan
18	extractelement	Menjalankan kode operasi *extractelement* pada konstan
19	insertelement	Menjalankan kode operasi *insertelement* pada konstan
20	shufflevector	Menjalankan kode operasi *shufflevector* pada konstan
21	extractvalue	Menjalankan kode operasi *extractvalue* pada konstan
22	insertvalue	Menjalankan kode operasi *insertvalue* pada konstan
23	OPCODE	Menjalankan semua kode operasi baik *binary* atau *bitwise*

III.1.7 Analisis Kebutuhan Non Fungsional

Analisis kebutuhan non fungsional merupakan analisis yang dibutuhkan untuk menentukan spesifikasi kebutuhan sistem. Spesifikasi ini juga meliputi elemen atau komponen-komponen apa saja yang dibutuhkan untuk sistem yang akan dibangun sampai sistem tersebut dapat diimplementasikan.

III.1.7.1. Analisis Perangkat Lunak

Dalam analisis *Self-Modifying Code* dibutuhkan sistem operasi GNU/Linux 64 bit dengan hak akses untuk membaca dan menulis di memori pada tabel /proc/{pid}/maps. Linux tersebut tidak memiliki keamanan pada *kernel* seperti SELinux dan PAX. Karena proses memerlukan perubahan segmen kode atau data

eksekusi maka, target program tidak boleh memiliki proteksi seperti RELRO(read-only relocations), stack canary atau SSP(*Stack Smashing Protector*) dan NX(*No Execute).*

III.1.7.2. Analisis Perangkat Keras

Perangkat keras yang dibutuhkan untuk *Self-Modifying Code* adalah komputer dengan dukungan arsitektur *Von Neumann* dimana program dan data disimpan dalam memori sehingga program dapat memodifikasi dirinya. Transfer data dari memori utama ke penyimpanan sementara juga menggunakan struktur data antrian atau dikenal dengan PIQ*(prefetch input queue).*

III.1.8 Analisis Dynamic Binary Instrumentation

Desain DBI yang akan dibangun harus mendukung sistem manajemen proses secara separasi. Hal ini berkaitan dengan transfer kontrol terhadap target program. Adapun langkah-langkah yang diperlukan adalah membangun komponen seperti berikut:

1. Binary Reader

   Dalam hal ini biner yaitu ELF mengandung tabel simbol yang berisi informasi yang nantinya digunakan untuk identifikasi fungsi *boundaries* atau cakupan seperti seksi .text pada kode asli maupun anotasi yang dibuat oleh *compiler*

2. Process Handler

   Modul ini mengatasi semua proses berkenaan dengan *task* yang berkaitan seperti memulai atau melampirkan ke proses yang akan diinstrumentasi, menangani event proses seperti panggilan ke fork() dan exec(), serta mempertahankan pemetaan alamat beban untuk *shared library* yang dimuat atau dibongkar. Hal tersebut juga menyediakan mekanisme kode *monitoring* yang diperlukan saat *startup* untuk memetakan memori bersama ke alamat target ruang pada memori

3. Function Discovery

   Komponen ini mengidentifikasi batas fungsi asli dengan membaca informasi tabel simbol yang hadir dalam biner sasaran. Hal tersebut

memelihara fungsi kamus atau repositori dari semua informasi mengenai rentang alamat, ukuran dan atribut lainnya dari kedua fungsi yaitu fungsi asli dan fungsi yang diinstrumentasi

4. Code Patcher

   Kode *patcher* menyediakan kemampuan *thread-safe* untuk secara dinamis menambal data dan kode dalam proses target. Kode patcher digunakan untuk memasukkan instruksi *break* di fungsi titik masuk dan kemudian menggantikan instruksi tersebut dengan instruksi *branch* untuk fungsi yang diinstrumentasi. Hal ini juga digunakan untuk *patch* data seperti tabel *switch* dalam target proses

5. Decoder

   Decoder menyediakan fungsionalitas untuk membaca fungsi kode asli dan didekode menjadi IR(*intermediate representation)*. IR terdiri dari instruksi dan blok yang memungkinkan operasi seperti penyisipan, penghapusan dan duplikasi

6. Encoder

   Encoder ini digunakan untuk mengkonversi fragmen IR yang mewakili fungsi kode yang diinstrumentasi ke biner bit yang sesuai. Bit yang di-*encode* ditulis ke dalam *shared memory*. *Encoder* juga menyediakan *templatizer* dan *scheduler* untuk memadatkan instruksi-instruksi ke dalam arsitektur 64-bit secara bundel.

7. Instrumentor

   Instrumentor berperan mengingat alamat fungsi asli beserta ukurannya. *Instrumentor* mendekodekan fungsi, menyisipkan kode penyelidikan dan menghasilkan salinan kode yang diinstrumentasi dalam *shared memory*

8. Unwind Handler

   Modul ini mengelola semua aspek data *unwind*. Informasi *unwind* dibaca, diperbarui maupun ditulis. Kode dihasilkan dan disisipkan ke dalam proses target untuk mendaftar hasil modifikasi informasi *unwind* ke sistem *run-time*.

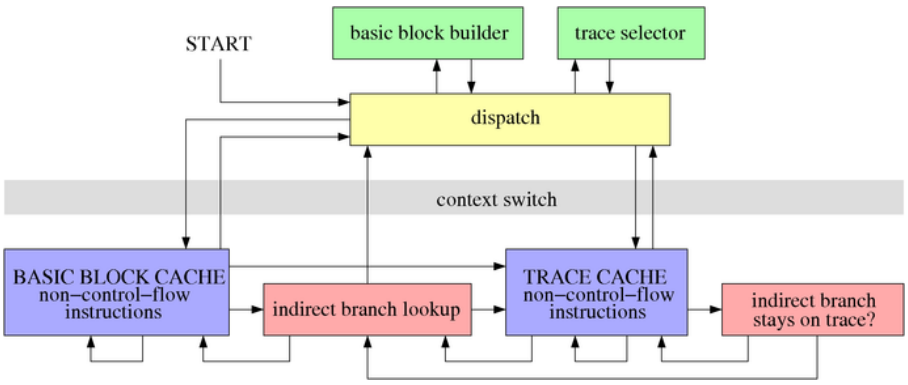
9. Counter Manager

   Modul ini mempertahankan *counter* yang akan digunakan oleh kode *probe* yang dimasukkan. Hal ini menyediakan set *counter* yang dapat berupa statis dialokasikan oleh *instrumentor* atau dialokasikan secara dinamis selama pelaksanaan target program (dengan kode pemeriksaan dimasukkan ke sasaran).

10. Profile Writer

    Hasil pengukuran yang akan diambil dari *counter* ditulis ke dalam data profil file yang dapat digunakan kembali ke oleh *compiler* yang dipandu.

Teknik yang dilakukan *Dynamic Binary Instrumentation* adalah observasi langsung pada *hit* yang menjadi target instrumentasi. *Introspection*-nya menggunakan *virtual machine* dimana tingkah laku program direferensikan pada tahap *runtime.* Adapun tahap-tahap yang dilakukan adalah sebagai berikut:



Gambar III.1 Cara Kerja DynamoRIO

1. Alat DBI dijalankan dengan *Input* target lokasi biner program dimana didefinisikan sebagai *parameter* yang diikuti dengan fungsi yang akan dianalisis atau sudah terletak pada modul pustaka yang dibuat
2. Instrumentasi kemudian mengolah kondisi fungsi target sesuai dengan dekripsi dari modul pustaka dalam memori
3. Pada tahap *runtime,* fungsi yang tidak dinstrumentasi berjalan tanpa pencegatan dari DBI, dalam hal ini ditangani oleh *dispatcher*
4. *Buffer* proses jalannya target program disimpan dalam *cache* yang dicatat oleh *listener* sebagai tindak lanjut untuk hasil keluaran
5. *Output* ditampilkan sesuai definisi dalam pustaka modul yaitu dapat berupa *stdout* maupun file yang merupakan hasil seleksi dari *tracer*

III.1.9 Analisis Aplikasi Sejenis

Selain *DynamoRIO* terdapat alat lainnya yang menyediakan *built-in* analisis *Self-Modifying Code* yaitu *Valgrind*. Terdapat dua opsi pada *Valgrind* yakni mendeteksi *Self-Modifying Code* pada *stack (--smc-check=stack)* atau keseluruhan kode (*--sm-check=all). Valgrind* memberikan catatan nasihat terhadap *routine* yang mempunyai masalah khususnya memori yang digunakan oleh program *Self-Modifying Code.* Berikut ini contoh hasil keluarannya:

```
  ==19182== Invalid write of size 4
  ==19182==    at 0x804838F: f (example.c:6)
  ==19182==    by 0x80483AB: main (example.c:11)
  ==19182==   Address 0x1BA45050 is 0 bytes after a
block of size 40 alloc'd
  ==19182==                at    0x1B8FF5CD:    malloc
(vg_replace_malloc.c:130)
  ==19182==    by 0x8048385: f (example.c:5)
  ==19182==    by 0x80483AB: main (example.c:11)
```

Pada *paging* RWX(Read, Write, Execute) kode yang digenerasikan mendapatkan terjemahan dalam blok yang berisi usaha atas penulisan(*writeable)*

program *Self-Modifying Code.* Hal tersebut dapat dapat dilihat bahwa *Valgrind* hanya berusaha mengecek tingkah laku memori yang tidak lazim dan memberikan alamat memorinya. Dengan kata lain fungsi program yang terjadi aktivitas *Self-Modifying Code* harus dieksaminasi kembali oleh *debugger* dengan memulai langsung pada poin entri alamat memori tersebut.

III.2 Perancangan Dynamic Binary Instrumentation

Perancangan *Dynamic Binary Instrumentaion* menggunakan *framework* atau alat *DynamoRIO* dengan memenfaatkan klien API yang sudah tersedia. Interaksi antar muka dilakukan melalui *command-line.* Adapun fungsi yang diperlukan untuk analisis program *Self-Modifying Code* dapat dilihat pada tabel berikut:

Tabel III.18 Perancangan API DynamoRIO

Nama Tabel	Nama API	Parameter	Tipe Data	Keterangan
DynAPI	dr_init	client_id	*unsigned integer*	Berupa ID yang digunakan untuk mengidentifikasi klien. Nilai ini ditetapkan pada pendaftaran klien dan diteruskan ke klien di dr_client_main ()
	dr_emit_flags_t	DR_EMIT_DEFAULT	*enumerations*	Emisi Secara Normal
		DR_EMIT_STORE_TRANSLATIONS		Menyimpan informasi terjemahan pada saat emisi daripada memanggil peristiwa blok atau *trace* nantinya untuk kemudian informasi diciptakan kembali
		DR_EMIT_PERSISTABLE		Mengindikasikan bahwa blok memenuhi syarat untuk bertahan ke file kode *cache* secara terus – menerus pada *disk*
		DR_EMIT_MUST_END_TRACE		Menunjukkan bahwa blok harus mengakhiri sebuah *trace*
		DR_EMIT_GO_NATIVE		Permintaan agar *DynamoRIO* melepaskan kontrol dari *thread* dan membiarkan berjalan secara *native* sampai klien menunjukkan bahwa *DynamoRIO* harus mengambil alih kembali
	dr_get_main_module	module_data_t*	*void pointer*	Melihat data modul untuk dieksekusi utama, Mengembalikan module_data_t harus dihancurkan dengan dr_free_module_data()

	dr_register_bb_event	dr_emit_flags_t*		Memiliki fungsi dimana tempat *drcontext, *tag, instrlist_t *bb diterjemahkan
	dr_free_module_data	module_data_t*		Menghancurkan sebuah penunjuk module_data_t yang dikembalikan oleh dr_module_iterator_next(), dr_lookup_module(),dr_lookup_module_by_name() dan dr_copy_module_data()
	dr_insert_call_instrumentation	drcontext*		Mengasumsi bahwa instruksi dekat dengan *call*
	dr_insert_mbr_instrumentation			Mengasumsi bahwa instruksi dekat dengan *branch*
	dr_exit_process	exit_code	*integer*	Keluar dari proses dan melakukan pembersihan yang akan memicu peristiwa keluar. Pada Linux, bagian dibawah 8 bit exit_code akan dijalankan secara normal. Jika bit 9 sampai 16 bit tidak kosong, DR akan mengirimkan *signal handler* untuk melakukan keluar secara normal.

1. dr_init

Ketika mendaftar proses diperlukan daftar jalan untuk pustaka klien diikuti opsi khusus yang terkait. *DynamoRIO* melihat “dr_client_main” di setiap pustaka dan panggilan fungsi ketika proses dimulai. Klien dapat mendaftar untuk menerima *callback* untuk berbagai perisitiwa dalam dr_client_main().

2. dr_emit_flags_t

Sebelum inisialisasi dilakukan, *DynamoRIO* membutuhkan kendali atas perilaku blok dasar dan *traces* ketika emisi ke kode *cache*. Fungsi ini menandakan dapat mengkombinasikan atau dikombinasikan bersama *bitmasks.* Untuk beberapa klien tanda tersebut dikembalikan oleh setiap klien secara bersamaan. *Enumerator* dalam dr_emit_flags_t dapat berupa *enum* yang harus dikembalikan seperti pada table berikut:

3. dr_get_main_module

Fungsi dr_get_main_module adalah cara yang dapat diandalkan untuk klien mendapatkan informasi tentang *executable* aplikasi utama. “dr_get_application_name” bergantung pada sistem file *proc* yang bisa menjadi tidak tersedia karena beberapa alasan seperti nama modul yang sama. Fungsi ini lebih kuat dalam menghadapi kasus sudut tersebut.

4. dr_register_bb_event

dr_rgister_bb_event merupakan register fungsi *callback* untuk peristiwa blok dasar. *DynamoRIO* memanggil fungsi sebelum memasukkan blok dasar baru ke kode *cache.* Ketika menambahkan blok dasar untuk jejak baru, *DynamoRIO* memanggil kembali fungsi dengan _trace yang diset *True* dalam memberikan klien kesempatan untuk menjaga instrumentasi yang sama di *trace* atau untuk mengubahnya. Instrumentasi blok dasar asli tidak berubah dengan tindakan apapun yang diambil dalam panggilan for_trace.

5. dr_free_module_data

Memori yang diakuisisi oleh modul harus dihancurkan. Fungsi ini berkaitan dengan dr_get_main_module yang dipakai. Penghancuran tersebut bukan memori target program melainkan *DynamoRIO* itu sendiri.

6. dr_insert_call_instrumentation

Asumsi bahwa instrumentasi dekat dengan fungsi panggil. Fungsi ini mempunyai dua parameter yaitu alamat instruksi panggilan(*caller)* dan alamat target panggilan(*callee).*

7. dr_insert_mbr_instrumentation

Asumsi bahwa instrumentasi adalah sebuah *indirect branch.* Fungsi ini diikuti dengan parameter alamat instruksi *branch* dan alamat target *branch.*

8. dr_exit_process

Fungsi untuk keluar dari proses dengan melakukan pembersihan penuh pada memori yang akan memicu peristiwa keluar pada target program. *DynamoRIO* akan mengirimkan sebuah *signal handler* dari sekedar proses keluar secara normal.

Tabel III.19 Perancangan API LLVM

Nama Tabel	Nama API	Parameter	Tipe Data	Keterangan
LLVMAPI	LLVMGetBitcodeModule	LLVMMemoryBufferRef	LLVMBool	Membangun Sebuah Modul dari *bitcode* dalam spesifik memori *buffer*
	ELFAsmParser	*null*	*Namespace*	Mengkonstruksi *instance* ELF *writer* yang baru
	DAGUpdateListener	SDNode*	*Void pointer*	Memperbaharui

				listener yang mengatasi DAG(Directed Acyclic Graphs)
	llvm::parseIR	MemoryBufferRef	std::unique_ptr	Mengembalikan modul dari memori *buffer* yang menahan *image bitcode*

Pada tabel diatas dapat dilihat bahwa *Dynamorio* juga diintegrasikan dengan meng-*extend* LLVM dengan menggunakan beberapa komponen *linking* seperti *bitreader, asmparser, selectiondag* dan *irreader.* Penambahan API tersebut untuk merekonstruksi hasil keluaran *DynamoRIO* menjadi terstruktur seperti LLVM IR. Adapun masing-masing fungsinya adalah sebagai berikut:

6. bitreader

Bitreader merupakan pustaka *inteface* untuk mengimplemetasikan input dari format LLVM *bitcode.* Modul ini mengembalikan referensi *buffer* memori secara spesifik pada *cache*

7. asmparser

Asmparser adalah sebuah kelas yang digunakan sebagai *parser* file *assembly.* Dalam rancangan ini *parsing* ditentukan oleh diagnosa pada memori yang mengikuti petunjuk *cache.*

8. Selectiondag

Sebuah instruksi LLVM IR melewati *selectiondag* atau dalam rancangan ini merupakan tahap seleksi proses menerjemahkan ke target spesifik mesin instruksi

9. irreader

irreader adalah fungsi untuk membaca *intermediate representation.* Dalam rancangan ini digunakan sebagai tahap akhir sebagai keluaran yang dapat dilihat dalam catatan file.

Sedikit berbeda dengan alat analisis bawaan LLVM IR, keluaran tidak menghasilkan kode statis melainkan tertera dinamis apa yang terjadi saat program dijalankan. Berikut dapat dilihat contoh hasil konversinya:

Tabel III.20 Intermediate Representation

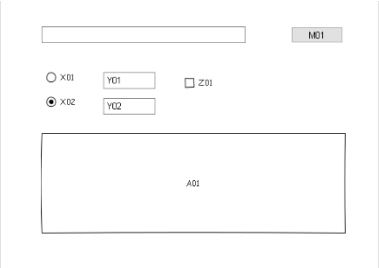
C Source Code	Intermediate Representation

`int main(int argc, char *argv[]) {`
 	
`return argc + 1;`
 	
`}`
	`%1 = alloca i32, align 4`
	`%2 = alloca i32, align 4`
	`%3 = alloca i8**, align 8`
	`store i32 0, i32* %1`
	`store i32 %argc, i32* %2, align 4`
	`store i8** %argv, i8*** %3, align 8`
	`%4 = load i32* %2, align 4`
	`%5 = add nsw i32 %4, 1`
	`ret i32 %5`

Dari tabel diatas dapat dilihat bahwa bahasa sudah berganti dengan suatu yang lebih sederhana dari bahasa *assembly*. Hal ini merupakan salah satu fitur upaya untuk mengetahui apa sebenarnya yang terjadi dalam memori dengan mudah.

Untuk yang tidak terbiasa dengan *commandline,* terdapat program bantuan berupa GUI(*Graphical User Interface)* sebagai pemanggil langsung perintah pada program.Rancangan antarmukanya langsung dihadapkan tiga menu dengan tab

yaitu menu mengubah ELF menjadi DCFG, DFCG menjadi bitcode dan instrumentasti. M*ock-up* dapat dilihat pada gambar berikut ini:



Gambar III.1 Mock-Up

Penjelasan objek Gambar III.2 dijelaskan secara spesifik pada Table III.21 berikut:

Tabel III.21 Mock-Up

Id_Objek	Jenis	Nama	Keterangan
L01	Tab	elf2dcfg	Tab menu
M01	File Chooser	Browse	Memilih lokasi target *executable*
X01	Radio button	X86	Memilih target arsitektur
X02	Radio Button	X86_64	Memilih target arsitektur
Y01	Text Box	func_map	Mengisi fungsi mapping
Y02	Text Box	symbol_entry	Mengisi simbol entri
Z01	Check Box	debug	Menceklis tampilan debug
A01	Text Box	output	Menampilkan hasil keluaran