# BAB II

# LANDASAN TEORI

## 2.1 Malicious Software

*Malicious Software* atau *malware* merupakan sekumpulan instruksi atau program yang berjalan pada suatu sistem komputer yang membuat sistem tersebut melakukan sesuatu yang diinginkan penyerang (Skoudis dan Zeltser, 2004). Dalam sumber lain menyebutkan bahwa secara sederhana, *malware* merupakan jenis perangkat lunak yang melakukan berbagai tindakan tanpa persetujuan atau sepengetahuan pemilik sistem yang mengakibatkan kekacauan sistem (Oriyano, 2014). Selain mengakibatkan kekacauan sistem, *malware* juga digunakan oleh pelaku kriminal dalam pencurian informasi, seperti informasi akun bank, *credit card*, hingga informasi akun *game online* (Oriyano, 2014). *Malware* diklasifikasi menjadi beberapa tipe. Tabel 2.1 berikut menjelaskan beberapa tipe *malware* secara umum.

Tabel 2.1. Beberapa tipe *malware* (Oriyano, 2014)

| **Tipe** | **Karakteristik** |
|---|---|
| *Virus* | Bentuk umum dari *malware*. Dapat mereplikasi dan menempelkan dirinya sendiri ke *file* atau *process* lain. |
| *Worm* | Penerus atau *successor* dari *virus*. Punya kemampuan replikasi seperti *virus* dengan sangat cepat. Dapat mengakibatkan *Denial of Service*. |
| *Trojan* | Mirip seperti *virus*. Menyamarkan dirinya sebagai program yang sah. |
| *Rootkit* | Dapat menyembunyikan dirinya sendiri dan sulit dideteksi. |
| *Spyware* | Mencuri informasi dari sistem tanpa sepengetahuan korban. |
| *Adware* | Program iklan yang umumnya mengganti *homepage browser* dan menampilkan *pop-up* iklan pada komputer korban. |

## 2.2 File Signature

*Malware* dapat dideteksi oleh antivirus berdasarkan *signature* pada *file* programnya sendiri ataupun *file* lain yang terinfeksi. *File Signature* merupakan data yang merepresentasikan suatu *file* yang digunakan untuk mengidentifikasi dan/atau memverifikasi suatu isi *file*. Ada dua tipe *file signature*: *file magic number* dan *file checksum*. *File magic number* merupakan penanda berupa deretan *bytes* yang berada di awal *file binary*. *Magic number* ini digunakan untuk mengidentifikasi tipe *file* (Sammes dan Jenkinson, 2000).

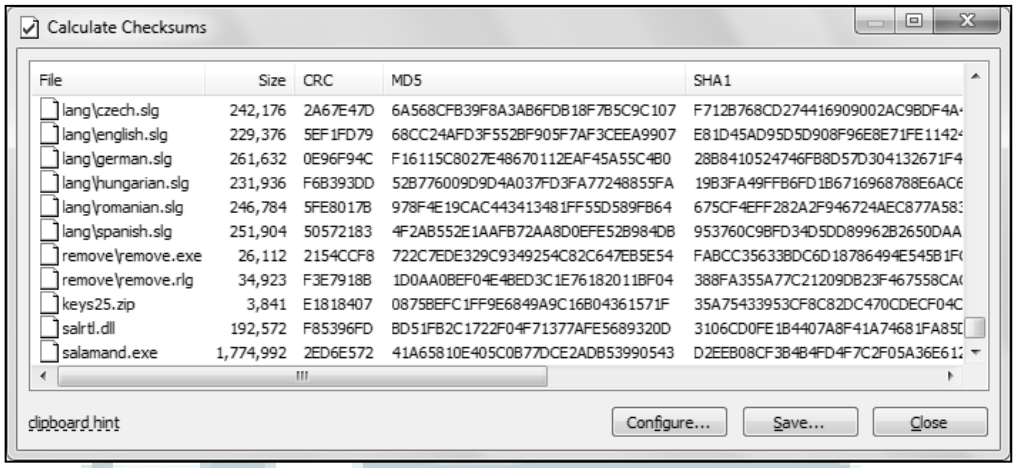
Tabel 2.2. Beberapa tipe *file* beserta *magic number* (Kessler, 2014)

| **Tipe File** | **Ekstensi Default** | **Nilai Heksa** | **ASCII** |
|---|---|---|---|
| Windows/DOS *Executable file* | .com, .dll, .drv, .exe, .pif, .qts, .qtx, .sys | 4D 5A | MZ |
| JPEG/JFIF *Graphic file* | .jfif, .jpe, .jpeg, .jpg | FF D8 FF | ÿØÿà |
| PK ZIP *Archive file* | .zip | 50 4B 03 04 | PK |
| RAR v4.x *file* | .rar | 52 61 72 21 1A 07 00 | Rar! |
| PDF *file* | .pdf | 25 50 44 46 | %PDF |
| *Executable and Linking Format executable file* (Linux/Unix) | - | 7F 45 4C 46 | .ELF |

Seperti yang ditunjukkan tabel 2.2, *file executable* Windows memiliki *magic number* dalam heksadesimal 4D 5A dan MZ dalam format ASCII (Kessler, 2014). Program Hex Editor dapat digunakan untuk membuka *file* dan mengetahui *magic number* dalam format heksadesimal dan ASCII. Bila menggunakan Notepad dari sistem operasi Windows, hanya dalam format ASCII-nya saja.

*Magic number* suatu *file* tidak dipengaruhi oleh ekstensinya atau sebaliknya. Apabila suatu *file executable* Windows berekstensi EXE diubah menjadi .JPG, Windows Explorer mendeteksinya berdasarkan ekstensi dan *icon file* tersebut berubah menjadi *icon file* JPG yang tentu saja akan gagal dibuka oleh *image viewer* karena *file* JPG tersebut bukan merupakan *file* gambar yang valid. Walaupun ekstensi telah diubah, isi *file* tetaplah sama sehingga *magic number*-nya pun masih sama. *Magic number* dapat berubah apabila diubah secara *hardcoded* menggunakan program Hex Editor. Walaupun *magic number file executable* Windows, 4D 5A diubah menjadi *magic number* lain yang valid, misalnya FF D8 FF untuk format JPG, program Image Viewer akan gagal membukanya karena struktur keseluruhan *file* tersebut tetaplah *file executable*.

Selain memiliki *signature magic number*, *file* juga memiliki *signature file checksum* atau disebut juga *hash sum*. *Signature* inilah yang digunakan sebagai *signature* utama dalam mendeteksi *malware*. *Checksum* merupakan penanda *file* yang berasal dari hasil suatu *checksum algorithm* yang menggunakan *hash function* terhadap isi suatu *file* yang digunakan untuk memverifikasi integritas *file* terhadap serangan dan/atau kesalahan transmisi data. Perubahan sedikit data saja pada *file*, akan menghasilkan perbedaan nilai *checksum* yang signifikan. *Checksum* tidak bisa digunakan untuk memverfikasi autentikasi data (Sammes dan Jenkinson, 2000). *File checksum* sering ditemukan dalam pengunduhan *file* untuk mendeteksi *corrupt* dan verifikasi integritas pada *file* yang diunduh. Apabila *file* yang diterima memiliki *checksum* yang sesuai, berarti *file* yang telah terunduh sama dengan yang tersedia di *server*. Gambar 2.1 menunjukkan contoh beberapa nilai *checksum* dari beberapa *file*.




Gambar 2.1. *Checksum* kumpulan *file* yang menggunakan CRC, MD5, dan SHA-1 (ALTAP, 2014)

## 2.3 Malware Signature

*Signature malware* menggunakan *file hash sum* saja atau dikombinasi dengan *file magic number* dan ditambah dengan *string signature*. *String signature* merupakan deretan *bytes* tertentu (bukan *file magic number*) yang terdapat dalam tubuh *malware* (Nurjadi, 2013). Beberapa *malware* memiliki *string signature* yang biasanya berupa informasi *file Dynamic Link Library* yang digunakan, alamat *registry*, *URL*, dan *IP Address* yang coba diakses (Hao, 2013). Walaupun *malware*-nya tidak sama, selama deretan *bytes* tersebut tidak berubah atau tidak berbeda, maka *malware* tetap akan terdeteksi. Hal ini membuat risiko salah deteksi menjadi lebih besar karena *file* yang bersih bisa saja kebetulan memiliki deretan *string* yang sama dengan *malware* (Nurjadi, 2013). Sebagai contoh, sebuah *trojan* yang dikenali oleh PCMedia Antivirus sebagai Prorat.F:Backdoor mencoba memodifikasi *registry* Windows pada alamat "HKEY_CURRENT_USER\software\Microsoft\WindowsNTScriptHost\Microsoft\DxDiag\WinSettings\Hata" dengan suatu pesan : "*Sorry! You have not Mac*

*System. its a protected file and deleted automatically.Kindly open this file on Mac OS.XP_FW_Disable*" (Anggiawan, 2014).

## 2.4 Traditional Hashing

Dalam *traditional hashing*, seperti MD5 dan SHA256 (Dunham, 2013), secara garis besar, nilai *hash* didapat dari langkah-langkah berikut.

1. Mulai dari *initial state*.
2. Pecah *input* ke dalam *block* yang berukuran tetap dan lakukan proses berikut.
3. Penghitungan matematis terhadap *current state* dengan *current block*.
4. Mendapatkan *new state* dari hasil langkah (3).
5. Pindah ke *block* selanjutnya.
6. Ulangi langkah (3), (4), dan (5) hingga semua *block* diproses.
7. *State* akhir, keluarkan hasil.

Tabel 2.3 berikut menunjukkan contoh nilai *hash* yang dihasilkan dari sebuah *file* VBS yang berisi pesan yang ditampilkan dalam sebuah *message box*.

Tabel 2.3. Contoh perubahan nilai *hash* MD5 dan SHA256

| **Isi code** | **Nilai MD5** | **Nilai SHA256** |
|---|---|---|
| msgbox("i *think* i love you") | f298bb291b1924884845dd1ef83ce080 | 1ea494b0c1228cd3d55a2543d5233d69594db22d890655d4ae668bb80dfdf593 |
| msgbox("i *feel* i love you") | 516a33c437ec3db5f8a345b5ba8256a8 | cf242bd14ef810d54fff3fa0062df55bcb09c4594693b7a5f40849c35314632d |
| msgbox("i *realise that* i love you") | c00715513b855c7b884a106872d221f9 | f0abb708a0df6538f951f0c56b09f361b58da663974a984cec6a11c95e108b76 |

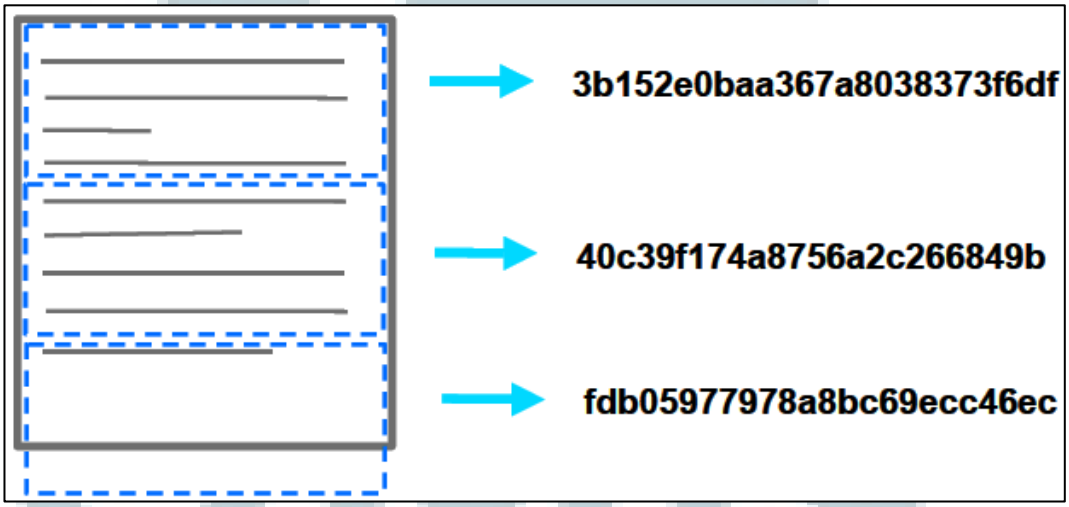
*Hash* MD5, SHA1, dan SHA256 menggunakan metode Merkle–Damgård *hash function* yaitu metode untuk membangun *collision-resistant cryptographic*

*hash function* dari *one-way compression function* (Goldwasser dan Bellare, 2008). *Collision-resistant hash function* merupakan sifat pada *hash function* dimana jika diberikan dua *input* yang berbeda, misalnya M dan M', tidak menghasilkan dua nilai *hash* yang sama (Hash(M) ≠ Hash(M')), atau dengan kata lain, tidak dimungkinkan dengan mudah ditemukan dua nilai *hash* yang sama untuk *input* yang berbeda (Goldwasser dan Bellare, 2008). Nilai *hash* sama untuk setiap *file* yang identik sehingga *hash* seperti MD5 dan SHA256 cocok digunakan untuk *fingerprinting* sebuah *file*, sedangkan yang dimaksud *one-way compression function* merupakan *function* yang mentransformasi dua buah *input* dengan panjang yang sama ataupun yang berbeda menjadi satu *output* dengan panjang yang tetap. Disebut *one-way* karena tidak dimungkinkan menghasilkan kembali data *input* asal dari hasil *output* yang diberikan. Jadi, tidak dimungkinkan untuk merekonstruksi ulang data asal dari nilai *hash* yang dihasilkan. Sebagai contoh, tidak dimungkinkan memproses nilai *hash* MD5 "c00715513b855c7b884a106872d221f9" kembali menjadi "msgbox("i *realise that* i love you")". Seperti yang ditunjukkan pada tabel 2.3, dalam *hash* seperti MD5 dan SHA256, perubahan susunan *byte* pada isi *file* walau sedikit saja akan menghasilkan nilai *hash* yang berbeda (Kornblum, 2007) sehingga *file* hasil modifikasi walau sedikit tidak akan terdeteksi dengan *hash* yang lama. *Malware* juga dimodifikasi sehingga menghasilkan banyak varian dan dapat tidak terdeteksi *antivirus* karena *signature*-nya telah berbeda. Hal ini menjadi kekurangan *antivirus* yang hanya mengandalkan *signature-based detection*, jika *signature*-nya sama dengan di *database*, maka terdeteksi, jika tidak, *malware* akan dibiarkan. Bukan berarti semua pendeteksian *malware* yang menggunakan nilai *hash*-nya itu

buruk, tetapi perlu dipilih algoritma *hash function* yang lebih baik, salah satunya *fuzzy hashing*.

## 2.5 Fuzzy Hashing

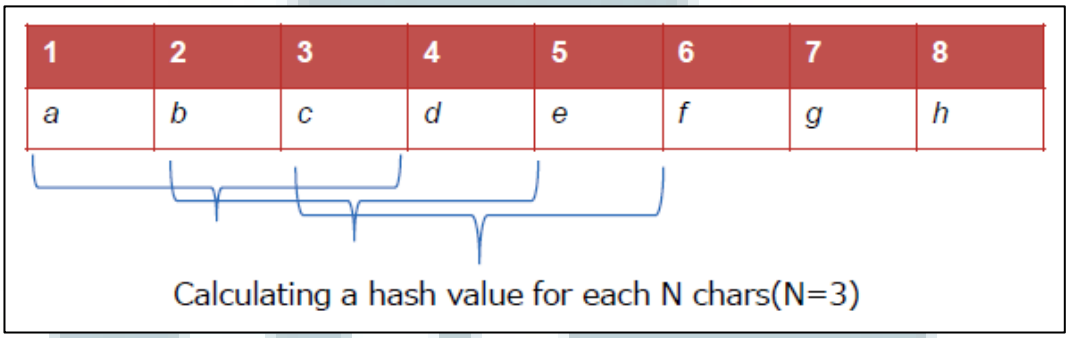
*Fuzzy Hashing* (disebut juga *Context Triggered Piecewise Hash*) merupakan *hash function* yang menggabungkan *Piecewise Hashing* dan *Rolling Hashing* dengan *hash* tradisional (FRRI Inc, 2014). *Piecewise Hashing* adalah algoritma *hashing* yang membagi *input file* per sejumlah *n-byte fixed size block* dan penghitungan *hash* dilakukan per *block* yang menghasilkan nilai *hash* sebanyak jumlah *block* lalu digabung untuk menghasilkan *hash* akhir (Breitinger, 2011). *Piecewise hash* dapat direalisasikan dengan menggunakan *hash* tradisional. *Hash* tradisional yang digunakan dalam *fuzzy hashing* adalah Fowler/Noll/Vo (FNV) *hash* (Kornblum, 2006). Gambar 2.2 berikut menunjukkan ilustrasi *piecewise hash*.




Gambar 2.2. Ilustrasi *Piecewise Hash* (Kornblum, 2007)

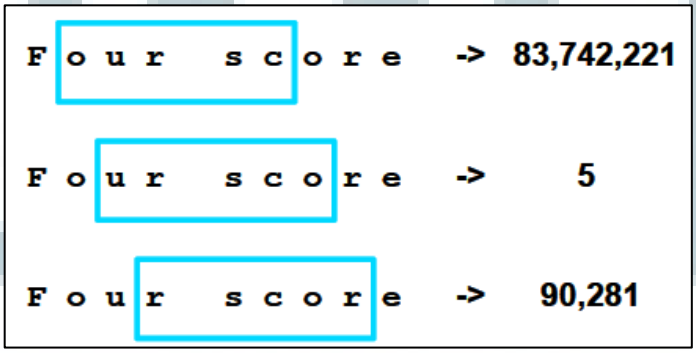
*Rolling Hashing* adalah metode untuk menghitung nilai *hash* per *sub-input* dimana penelusuran *sub-input* dari awal *file* hingga akhir *file* menggunakan

sebuah *fixed size window* yang memiliki lebar *n byte* (FRRI Inc, 2014). *Window* tersebut melakukan penelusuran dan pembacaan *byte* per *byte* dari awal *file* hingga akhir *file*. Gambar 2.3 berikut menunjukkan ilustrasi *rolling hash* dengan lebar *window n=3*.




Gambar 2.3. Ilustrasi *rolling hash* dengan ukuran *n* = 3 (FRRI Inc, 2014)

Dalam *fuzzy hash, signature* dihasilkan dari kumpulan *Least Significant Bit* (LSB) yang berasal dari hasil *hash* tradisional per bagian *file* (*piecewise hash*) dengan batas bagian atau *block* dan pengambilan *hash*-nya dipicu oleh *rolling hash* (Kornblum, 2007). *Rolling hash* berperan sebagai *pseudo-random function* yang bertanggung jawab membagi *input* (Breitinger, 2011). Gambar 2.4 berikut menunjukkan ilustrasi lain mengenai penelusuran dan pembacaan *input file* menggunakan *rolling hash*.




Gambar 2.4. Ilustrasi lain *rolling hash* dengan ukuran *n* = 6 (Kornblum, 2007)

Secara garis besar, nilai *hash* dari *fuzzy hashing* didapat dari langkah-langkah berikut. (Kornblum, 2007)

1. Tetapkan sebuah *trigger* atau pemicu sebagai penanda *rolling* isi *file* ke dalam *block* (digunakan *blocksize* berdasarkan ukuran *file* sebagai *trigger*).
2. Baca *file*.
3. Lakukan *rolling* dan hitung *hash* tradisional (*hash* tradisional menggunakan Fowler/Noll/Vo (FNV) *hash*). Jika *trigger* ditemukan atau dicapai, ambil *Least Significant Bit* (LSB) dari hasil *hash* tradisional.
4. Ulangi langkah (2) dan (3) hingga *file* berakhir.
5. Ketika selesai, gabungkan LSB-LSB untuk membuat *signature.*

*Rolling* merupakan proses penelusuran isi *file* untuk membagi isi *file* ke dalam *block-block* dengan ukuran *block* tergantung dari *trigger* yang ditetapkan; jika sebuah *trigger* ditemukan, maka isi *file* sampai *trigger* merupakan satu *block* dan dilakukan penghitungan nilai *hash* dengan *hashing* tradisional dan setelah melewati *trigger* merupakan *block* selanjutnya.

*Fuzzy hashing* dapat dipilih sebagai alternatif untuk menghitung *signature malware* dengan nilai *hash* yang dihasilkan dari beberapa *file* yang bersumber dari *file* yang sama dengan modifikasi kecil yang berbeda-beda tetap dapat terlihat kemiripannya dibandingkan dengan nilai *hash* MD5 atau SHA256 yang menghasilkan nilai *hash* yang tidak mirip walau ternyata *file-file* yang dibandingkan *hash*-nya hanya berbeda beberapa *bytes*, sehingga dari kemiripan nilai *fuzzy hash* dapat dilakukan deteksi *generic* dengan membandingkan *fuzzy hash* antara *file* yang dideteksi dengan *signature* yang ada di *database* yang menghasilkan derajat kemiripan (0%-100%). Keuntungan yang didapat yaitu

*signature fuzzy hash* dari satu *file* dapat digunakan untuk mendeteksi beberapa *file* hasil modifikasi kecil (Hurlbut, 2009). Selain itu, berbeda dengan *hash* tradisional yang memiliki panjang nilai *hash* tetap, nilai *hash* yang dihasilkan dari *fuzzy hash* memiliki panjang yang berbeda-beda tergantung ukuran *file*. Tabel 2.4 berikut menunjukkan contoh perbandingan nilai *hash* yang dihasilkan menggunakan *tools* ssdeep.

Tabel 2.4. Contoh perubahan nilai *fuzzy hash* dan SHA256

| **Isi code** | **Nilai Fuzzy Hash** | **Nilai SHA256** |
|---|---|---|
| msgbox("i *think* i love you") | 3:rCmNkRXFErFTn:FN0iZ | 1ea494b0c1228cd3d55a2543d5233d69594db22d890655d4ae668bb80dfdf593 |
| msgbox("i *feel* i love you") | 3:rCmNkDAFmMFTn:FNt | cf242bd14ef810d54fff3fa0062df55bcb09c4594693b7a5f40849c35314632d |
| msgbox("i *realise that* i love you") | 3:rCmNkXAJHDF6AFTn:FN1Z6i | f0abb708a0df6538f951f0c56b09f361b58da663974a984cec6a11c95e108b76 |

Algoritma *spamsum* yang dikembangkan oleh Andrew Tridgell dan Jesse Kornblum digunakan dalam *fuzzy hashing* dan *signature*-nya memiliki format "*blocksize:hash_part1:hash_part2*" dengan tanda *colon* ':' sebagai pemisah. *Hash_part1* memiliki panjang *hash* maksimal (*S*) sebanyak 64 karakter yang berupa karakter *Base64 Data Encoding* berupa karakter A − Z, a − z, 0 − 9, '+', dan '/' (S. Josefsson, 2006). Inisialisasi nilai *blocksize* ($b_{init}$) didapat berdasarkan ukuran *file* yang akan dihitung nilai *hash*-nya dengan rumus (2.1) berikut (Kornblum, 2006).

$$b_{init} = b_{min} 2^{\left\lceil \log_2 \frac{file\_size}{S \cdot b_{min}} \right\rceil} \quad \text{..... rumus (2.1)}$$

Dengan *blocksize* minimal ($b_{min}$) = 3 dan panjang *hash* maksimal (*S*) = 64, untuk memenuhi rumus (2.1), dapat dilakukan pendekatan iteratif yang

menghasilkan $b_{init}$ dengan melakukan *i* iterasi dimulai dari 0 dalam $3 * 2^i$ hingga memenuhi $3 * 2^i * 64 > file\_size$. *Blocksize* yang didapat dalam $b_{init}$ belum tentu menjadi *blocksize* yang akan ditulis dalam *signature* akhirnya. Jika *hash_part1* yang dihasilkan setelah penghitungan *hash* yang menggunakan *blocksize b* memiliki panjang kurang dari $\frac{S}{2}$, maka nilai *blocksize b* akan dikurangi menjadi $\frac{b}{2}$, dan dilakukan penghitungan *hash* ulang dengan nilai *blocksize* baru (Breitinger dan Baier, 2011). Gambar 2.5 dan 2.6 berikut menunjukkan *pseudocode*-nya.

```
b = compute_initial_block_size(input)
done = FALSE
while (done = FALSE) {
      initialize_rolling_hash(r)
      initialize_traditional_hash(h1)
      initialize_traditional_hash(h2)
      signature1 = ""  signature2 = ""
      foreach byte d in input {
            update_rolling_hash(r,d)
            update_traditional_hash(h1,d)
            update_traditional_hash(h2,d)
            //jika trigger pertama dicapai, ambil nilai dan
append ke signature1
            if (get_rolling_hash(r) mod b = b − 1)then {
                  signature1+= get_traditional_hash(h1) mod 64
                  initialize_traditional_hash(h1)
            }
            //jika trigger kedua dicapai, ambil nilai dan
append ke signature2
            if (get_rolling_hash(r) mod (b * 2) = b * 2 − 1) then {
                  signature2+= get_traditional_hash(h2) mod 64
                  initialize_traditional_hash(h2)
            }
      }
      if length (signature1)< S/2 then
            b = b/2
      else
            done = TRUE
}
signature = b+ ":" +signature1+ ":" +signature2 //block_size:hash_part1:hash_part2
```

Gambar 2.5. *Pseudocode* algoritma *spamsum* dalam *fuzzy hashing* (Kornblum, 2006)

```
//traditional hash menggunakan FNV hash
//menggunakan angka-angka dalam implementasi fuzzy hash dengan
//aplikasi ssdeep (Jesse Kornblum) dan spamsum (Andrew
Tridgell)

FNV_PRIME = 0x01000193
FNV_INIT = 0x28021967
size = 7 //ukuran rolling window

compute_initial_block_size(input)
{
        b=3
        while(b*S < input.file_size)
        {
                b=b*2
        }
        return b
}

initialize_rolling_hash(r)
{
        r.x = r.y = r.z = r.c = 0
        r.window = Array[size]
}

update_rolling_hash(r,d)
{
        r.y = r.y −r.x
        r.y =r. y + size * d
        r.x =r.x + d
        r.x =r.x – r.window [r.c mod size]
        r.window [r.c mod size] = d
        r.c =r.c + 1
        r.z =r. z << 5
        r.z =r. z XOR d
        return r.x + r.y +r. z
}

initialize_traditional_hash(h)
{
        return h = FNV_INIT
}

update_traditional_hash(h,d)
{
        h = h * FNV_PRIME
        return h = h XOR d
}
```

Gambar 2.6. Lanjutan *pseudocode* algoritma *spamsum* dalam *fuzzy hashing* (Kornblum, 2006)

## 2.6 Levenshtein Distance

Derajat kemiripan antara dua *fuzzy hash* ditentukan dengan menggunakan algoritma *Levenshtein Distance* yang menghitung jarak antara dua *fuzzy hash* yang berupa *string* atau kumpulan *bytes. Distance* atau jarak yang dimaksud dalam algoritma ini adalah jumlah minimal operasi yang dibutuhkan untuk mengubah suatu *string* ke *string* yang lain, dimana operasi-operasi tersebut adalah operasi penyisipan (*insertion*), penghapusan (*deletion*), atau penggantian sebuah karakter (*substitution*) (Ilmy, dkk., 2006). Jika jarak yang dihasilkan semakin mendekati nol, maka kedua *string* yang dibandingkan semakin mirip; dan jika jarak yang dihasilkan semakin mendekati dengan panjang *string* terpanjang, maka kedua *string* yang dibandingkan semakin tidak mirip. Sebagai contoh, *string* "apple" dibandingkan dengan "opel", maka jarak (jumlah minimal operasi untuk mengubah "apple" menjadi "opel" atau sebaliknya) yang dihasilkan adalah 3, dengan langkah sebagai berikut:

1. ubah apple menjadi opple (operasi substitusi 'a' menjadi 'o');
2. ubah opple menjadi oppl (operasi penghapusan 'e');
3. ubah oppl menjadi opel (operasi substitusi 'p' menjadi 'e').

Algoritma ini dapat diimplementasikan dengan pendekatan rekursif dan *dynamic programming*. Dalam penelitian ini, akan digunakan pendekatan *dynamic programming* dengan *space complexity* Θ(mn), dimana m = panjang *string* pertama dan n = panjang *string* kedua. Gambar 2.7 menunjukkan implementasinya menggunakan pendekatan *dynamic programming* (Fischer dan Robert, 1974).

```
int levenshtein_distance(string s1, string s2)
{
       //inisialisasi cost untuk semua operation
       int cost = 1;

       //alokasi dan inisialisasi array2D
       int[,] d = [s1.Length + 1, s2.Length + 1];
       for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;
       for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

       //penghitungan
       for (int j = 1; j <= s2.Length; j++)
       {
              for (int i = 1; i <= s1.Length; i++)
              {
                     //kondisi bila ada karakter yang sama
                     if (s1[i - 1] == s2[j - 1])
                         d[i, j] = d[i - 1, j - 1];

       //selain itu, cek cost minimal perubahan s1
       //menjadi s2 antara deletion, insertion, dan substitution
              else
              {              d[i, j] =
                                  Math.Min(Math.Min(
                                         //deletion satu karakter
                                  d[i - 1, j] + cost,
                                         //insertion satu karakter
                                  d[i, j - 1] + cost),
                                         //substitution satu karakter
                                  d[i - 1, j - 1] + cost
                                  );
              }
       }
     return d[s1.Length, s2.Length];
}
```

Gambar 2.7. *Levenshtein Distance* dengan pendekatan *dynamic programming* (Fischer dan Robert, 1974)

Setelah didapat *distance*, maka derajat kemiripan *string* pertama (*s1*) dan *string* kedua (*s2*) dengan rentang dari nol (kedua *string* tidak sama) sampai dengan satu (kedua *string* sama persis) dapat ditentukan. Acuan panjang *string* total dalam menentukan *similarity* adalah panjang *string* terpanjang. Rumus (2.2) digunakan untuk menentukan *similarity* (Chapman, 2005).

$$Similarity = 1 - \left(\frac{levenshtein\_distance(s1, s2)}{\max(s1.Length, s2.Length)}\right) \quad \text{.....rumus (2.2)}$$

```
int lev_dist(byte[] s1, byte[] s2)
{
      int insert_cost = delete_cost = 1;
      int substitute_cost = 3;
      int swap_cost = 5;
      ...
      if (s1[i - 1] == s2[j - 1])
            d[i, j] = d[i - 1, j - 1];

      //pengecekan apakah ada karakter yang mengalami operasi swap
      else if (i - 1 > 0 && j - 1 > 0 && s1[i - 1] == s2[j - 2] &&
s1[i - 2] == s2[j - 1])
            d[i, j] = Math.Min(
                        Math.Min(
                              Math.Min(
                              d[i - 2, j - 2] + swap_cost,
                               d[i - 1, j] + delete_cost),
                              d[i, j - 1] + insert_cost),
                              d[i - 1, j - 1] + substitute_cost);
      else
            d[i, j] = Math.Min(Math.Min(
                        d[i - 1, j] + delete_cost,
                        d[i, j - 1] + insert_cost),
                        d[i - 1, j - 1] + substitute_cost);
      ...
     return d[s1.Length, s2.Length];
}
```

Gambar 2.8. *Levenshtein Distance* termodifikasi dengan pengubahan nilai *cost* dan penambahan operasi *swap* (Breitinger, 2011; Rutenberg, 2008)

Seperti yang ditunjukkan gambar 2.8, untuk menghitung *distance* antara dua *signature* (*s1* dan *s2*), diperlukan sedikit modifikasi *Levenshtein Distance* yang asli (perubahan ditandai dengan huruf dicetak tebal), yaitu dengan mengizinkan operasi *swap* (contohnya, perubahan dari 'ict' menjadi 'itc' adalah satu operasi *swap*) dan *cost* yang dibutuhkan untuk tiap operasi berbeda. *Cost* untuk operasi *insertion* dan *deletion* adalah 1, *cost substitution* = 3, dan *cost swap* = 5 (Breitinger, 2011). Rumus (2.3) digunakan untuk memperoleh nilai *similarity* setelah *distance* dihitung dengan panjang *hash* (*S*) = 64 (Breitinger, 2011).

$$Similarity = 100 - \left(\frac{100 \cdot S \cdot lev\_dist(s1, s2)}{64 \cdot (s1.Length + s2.Length)}\right) \quad \text{.....rumus (2.3)}$$

Sebelum dilakukan pembandingan *hash* antara dua buah *signature*, *blocksize* keduanya dibandingkan terlebih dahulu. Pembandingan baru bisa dilakukan jika *blocksize signature* pertama ($b_x$) dan *blocksize signature* kedua ($b_y$) memenuhi $b_x = b_y$ atau $2b_x = b_y$ atau $b_x = 2b_y$. Selain itu, pembandingan *distance* tidak dilakukan dan hasilnya 0 (Kornblum, 2006). Jika $signature_x$ ($b_x$:$hash\_part1_x$:$hash\_part2_x$) dan $signature_y$ ($b_y$:$hash\_part1_y$:$hash\_part2_y$) akan dihitung *distance*-nya. Tabel 2.5 berikut menunjukkan cara membandingkannya.

Tabel 2.5. Aturan pembandingan *fuzzy hash* antara dua *signature* (Kornblum, 2006)

| **Kondisi Blocksize** | **Pembandingan yang dilakukan untuk dihitung distance-nya** |
|---|---|
| $b_x = b_y$ | Nilai maksimal antara pembandingan $hash\_part1_x$ dengan $hash\_part1_y$ dan pembandingan $hash\_part2_x$ dengan $hash\_part2_y$ |
| $2b_x = b_y$ | $hash\_part2_x$ dengan $hash\_part1_y$ |
| $b_x = 2b_y$ | $hash\_part1_x$ dengan $hash\_part2_y$ |
| Selain itu | Tidak ada. *Distance* = 0 |

## 2.7 Signature-Based Detection

Berdasarkan kemampuan aktivitas *scanning*, biasanya *antivirus* memiliki dua fitur, yaitu *on-demand scanner* dan *real time protection* (Graves, 2010). Istilah seperti *real time protection, on-access scanning, background guard, resident shield* adalah sinonim yang merujuk pada kemampuan *antivirus* untuk melakukan proteksi otomatis dan memonitor aktivitas sistem komputer, seperti melakukan *scan* terhadap *object* atau *file* yang diakses *user*, ketika DVD atau *removable disk*, *email attachment*, dan halaman *web* dibuka (CYREN Global View Security Lab, 2014). Sedangkan pada *antivirus* yang hanya memiliki

kemampuan *on-demand scanner* saja, akan melakukan *scan* ketika secara eksplisit diminta oleh *user*, misalnya dengan melakukan klik kanan pada suatu *folder* dan memilih opsi *scan files in this folder*. Tidak semua *antivirus* memiliki kedua fitur tersebut, *antivirus* yang paling sederhana memiliki kemampuan *on-demand scanner* saja.

Menurut Frederick B. Cohen, tidak ada metode yang sempurna yang dapat mendeteksi semua virus secara tepat (Chess dan White, 2011). Secara garis besar, ada dua metode *antivirus* dalam identifikasi dan deteksi *malware*, yaitu *static detection* dan *dynamic detection*. *Static detection* menggunakan atribut-atribut statis berdasarkan *signature* dari *malware* (Yonts, 2009). Pendeteksian dilakukan dengan mencari *signature* di dalam *database signature*. Jika *signature* dari suatu *file* ditemukan, maka *file* tersebut dideteksi sebagai *malware* dengan informasi berdasarkan keterangan *signature* di dalam *database*. Pencarian *signature* pada *database* dapat dilakukan melalui pasangan *key-value* (*hash* sebagai *key*, informasi *malware* sebagai *value*). Sedangkan metode *dynamic detection* menggunakan hasil analisis perilaku *malware* pada saat *runtime* (disebut juga *Behavioural-based detection*). Dalam penelitian yang dilakukan, implementasi metode *signature detection* menggunakan atribut berupa nilai *hash* dari *malware* saja dengan kemampuan *on-demand scanner*.

## 2.8 Tingkat Deteksi dan Akurasi

Tingkat deteksi merupakan rasio antara jumlah serangan yang dapat dideteksi terhadap jumlah total serangan (Sen, Clark, dan Tapiador, 2010). Jika diterapkan dalam pengujian deteksi *malware*, maka tingkat deteksi merujuk pada jumlah *malware* yang dapat dideteksi dibagi jumlah total *malware* yang diuji.

Rumus (2.4) berikut digunakan untuk memperoleh nilai tingkat deteksi dalam pengujian (Sen, Clark, dan Tapiador, 2010).

$$Tingkat\ Deteksi = \frac{Jumlah\ malware\ terdeteksi}{Jumlah\ sampel\ malware\ dalam\ pengujian} \quad \text{...rumus (2.4)}$$

Menurut Charles E. Metz, akurasi merupakan rasio antara jumlah *correct decisions* terhadap jumlah total *case* yang diuji. Jumlah *correct decisions* berupa jumlah *true positive* ditambah jumlah *true negative* (Metz, 1978). Jika diterapkan dalam pengujian deteksi *malware*, maka tingkat akurasi deteksi merujuk pada perbandingan antara seberapa banyak aplikasi dengan suatu *database signature* dapat mendeteksi secara benar bahwa suatu *file* merupakan *malware* (*true positive* atau *TP*) dan seberapa banyak aplikasi dengan suatu *database signature* mendeteksi secara benar bahwa suatu *file* bukan merupakan *malware* dari kumpulan *file* yang diuji (*true negative* atau *TN*). Rumus (2.5) berikut digunakan untuk memperoleh nilai tingkat akurasi dalam pengujian (Metz, 1978).

$$Tingkat\ Akurasi = \frac{Jumlah\ TP + Jumlah\ TN}{Jumlah\ file\ yang\ diuji} \quad \text{.....rumus (2.5)}$$