BAB II

LANDASAN TEORI

2.1 *Spam*

Spam (Cormac G. V, 2005) adalah *email* yang tidak diharapkan dan tidak diminta, yang dikirim secara acak tidak pandang bulu, langsung atau tidak langsung oleh pengirim yang tidak memiliki hubungan dengan penerima.

Ada beberapa jenis *spam*, *spam* dapat dikategorikan sebagai berikut (Pratiwi P. Y, n.d.) :

- *Junk mail* yaitu *email* yang dikirimkan secara besar-besaran dari suatu perusahaan bisnis, yang sebenarnya tidak diinginkan.
- *Non-commercial spam*, misalnya surat berantai atau cerita humor yang dikirimkan secara masal tanpa tujuan komersial tertentu.
- *Pornographic spam* yaitu *email* yang dikirimkan secara masal untuk mengirimkan gambar-gambar pornografi.
- Virus *spam* yaitu *email* yang dikirimkan secara masal, dan mengandung virus atau *Trojans*.

Tujuam dari *spam* itu sendiri (Cormac G. V, 2008) adalah untuk mengirimkan informasi kepada penerima, dimana konten dari pesan yang dikirim umumnya berisi iklan yang menawarkan produk-produk tidak penting, illegal atau produk yang ada, umpan untuk skema penipuan, tujuan promosi, atau menyebarkan malware yang didesain untuk membajak komputer penerima. Karena mengirim *email* relative sangat murah, maka pengirim hanya menargetkan sebagian kecil dari penerima, mungkin satu dari seribu atau bahkan lebih sedikit

lagi yang dibutuhkan untuk merespond *spam* yang mengakibatkan keuntungan bagi pengirim. Untuk mencegah tujuan tersebut dapat tercapai maka dibuatlah *spam filter*, dimana *spam filter* itu sendiri adalah sebuah teknik yang secara otomatis dapat mengidetifikasi *spam* dengan tujuan mencegah terkirimnya *email spam* tersebut.

2.2 Metode Bayesian

2.2.1 Definisi Bayesian

Metode *Bayesian* ini pertama kali dikemukakan oleh ilmuan matematika berkebangsaan inggris bernama Thomas Bayes, dimana metode ini digunakan untuk mengetahui kemungkinan suatu hal terjadi berdasarkan kejadian-kejadian yang terjadi dimasa lalu, Metode *Bayesian* ini kemudian dikembangkan oleh dua kelompok peneliti, yaitu Pantel & Lin dan Microsoft Research sehingga metode ini memungkinkan untuk digunakan dalam bidang teknologi *spam filter*, namun yang membuat Metode *Bayesian* ini popular adalah karena pendekatan yang dilakukan oleh Paul Graham (Adipranata R, 2006).

2.2.2 Bayesian Filter

Bayesian filter mendeteksi *spam* dengan menghitung probabilitas suatu *email* berdasarkan kata-kata yang terdapat didalamnya. Dalam metode ini diperlukan dua buah database yang satu berisi informasi karakteristik dari *legitimate mail* (HAM Database) yang didapat dari sample *legitimate mail* dan yang lainnya berisi tentang informasi karakteristik dari *spam mail* (SPAM Database) yang didapat dari sample *spam mail* (GFI Software, 2011).

Dalam *Bayesian Filter* ini diperlukan suatu latihan yang dilakukan oleh *Bayesian Filter* itu sendiri dengan menghitung ulang probabilitas dari setiap kata yang muncul dalam *legitimate mail folder* ataupun dalam *spam mail* folder, latihan ini sebaiknya dilakukan berkala sehingga *filter*isasi yang dilakukan selalu akurat dan tidak banyak melakukan *false positive*. *False Positive* adalah *legitimate mail* yang dianggap *spam mail* oleh *Bayesian Filter*, hal ini dapat terjadi karena angka probabilitas di dalam database tidak *up to date* dengan *email-email* yang telah masuk.

Untuk menghitung probabilitas *spam* dari suatu kata dapat dilakukan seperti ini (Adipranata R, 2006).:

$$\mathbf{P_{spam} = N_{spam} / (N_{spam} + N_{non\text{-}spam})} \quad \text{… Rumus 2.1}$$

Dimana :

P_{spam} = Probabilitas suatu kata "x" terdapat pada *spam mail*

N_{spam} = jumlah kata "x" yang muncul dalam keseluruhan *spam mail*

$N_{non\text{-}spam}$ = jumlah kata "x" yang muncul dalam keseluruhan *legitimate mail*

Rumus lain yang digunakan untuk menghitung probabilitas dari suatu kata, terutama jika nilai N*spam* dan Nnon-*spam* kecil adalah bahwa probabilitas akan terletak di sekitar probabilitas ketidakpastian (P = 0,5) (Adipranata R, 2006).

$$P_{Spam} = 0{,}5 + \frac{(N_{Spam} - N_{Non-spam})}{C_1 * (N_{Spam} + N_{Non-spam} + C_2)} \quad \text{… Rumus 2.2}$$

Dimana C_1 dan C_2 adalah konstanta yang dipilih melalui eksperimen. Misalkan jika $C_1 = 2$ dan $C_2 = 1$, dan jika suatu kata "x" hanya ditemukan satu pada *spam mail* dan tidak ada pada *non-spam mail* maka probabilitas *email* baru yang mengandung kata tersebut menjadi *spam* hanyalah 0,75. Probabilitas ini tidak terlalu tinggi untuk dikategorikan sebagai *spam*. Sementara jika kata tersebut ditemukan pada sepuluh *spam-mail* dan tidak ditemukan sama sekali pada *non-spam mail*, maka probabilitas lokal-nya akan sama dengan 95.4%, yang cukup tinggi untuk dikategorikan sebagai *spam*. Perhitungan probabilitas ini jika dilakukan dengan persamaan rumus 2.1, akan memberikan hasil yang tidak akurat, yaitu probabilitas mutlak = 1.

Probabilitas masing masing kata tersebut kemudian *menggunakan Bayesian chain rule* untuk menentukan probabilitas total dari suatu *message* adalah *spam*. *Bayesian* chain rule dirumuskan sebagai berikut. (Adipranata R, 2006).

$$\frac{abc \ldots n}{abc \ldots n + (1-a)(1-b)(1-c)(1-n)} \quad \ldots \text{Rumus 2.3}$$

Dimana : abc…n = Probabilitas suatu message dikategorikan sebagai *spam* dengan adanya kata 'a','b','c', sampai dengan kata ke-n.

2.2.3 Kelemahan Bayesian Filter

Metode *Bayesian* juga memiliki kelemahan yaitu bila *spam*mer memasukan kata-kata yang sengaja dibuat menjadi salah ejaan dan karakteristik tersebut tidak

ada pada SPAM database maka *email spam* tersebut akan tetap dimasukan ke dalam inbox user dan dengan adaptasi yang dilakukan otomatis oleh metode *Bayesian* ini tidak sepenuhnya berakibat baik karena sering kali adaptasi ini menyebabkan *false* postitves yang lebih tinggi (Green T, 2005).

2.3 Metode URL Filtering

Menurut penjelasan yang diberikan oleh Ted Green Metode URL *Filtering* adalah metode *filter*isasi *spam* dengan membuat suatu database yang berisi URL blacklist untuk kemudian dipakai menjadi acuan untuk memeriksa *email-email* yang masuk yang mengandung link URL "Click me" didalamnya, jika link URL tersebut ada pada database URL blacklist maka *email* tersebut akan dikategorikan sebagai *email spam*. (Green T, 2005)

2.4 Fungsi Hash

Fungsi *hash* atau yang disebut juga dengan message digest adalah fungsi utama yang memetakan *string* acak menjadi *string* biner dengan panjang *string* yang tetap (Mironov, 2005). Fungsi *hash* yang popular dipergunakan adalah CRC32, MD5, dan SHA 256.

2.4.1 Cyclic Redundancy Check 32

Cyclic Redundancy Check (CRC) adalah salah satu fungsi *hash* yang dikembangkan untuk mendeteksi kerusakan data dalam proses transmisi ataupun penyimpanan. (Anharku, 2009). CRC menghasilkan suatu checksum yaitu suatu nilai yang dihasilkan oleh dari fungsi *hash*nya, dimana nilai tersebut digunakan

untuk mendeteksi error pada transmisi ataupun penyimpanan data. Nilai CRC dihitung dan digabungkan sebelum dilakukan transmisi data atau penyimpanan, kemudian penerima akan melakukan verifikasi apakah data yang diterima tidak mengalami perubahan atau kerusakan. CRC32, dimana 32 melambangkan panjang dari checksum dalam *bit*. Bentuk CRC yang disediakan untuk algoritma sesuai dengan ide pembagian polynomial dan hal ini digunakan untuk memperhitungkan checksum yang sama dari seluruh algoritma CRC. Algoritma CRC adalah cara yang bagus dan teruji untuk pengecekan byte dalam jumlah besar dari suatu file yang telah termodifikasi maupun tidak.

2.4.2 Message Digest Algorithm 5

Message Digest Algorithm 5 (MD5) (Rivest, 1992) adalah algoritma message digest yang dikembangkan oleh Ron Rivest di MIT. Pada dasarnya algoritma ini adalah versi yang lebih aman dari algoritma yang pernah dibuat oleh Ron Rivest sebelumnya, yaitu MD4. Hingga saat ini, MD5 adalah fungsi *hash* yang aman digunakan dan menjadi acuan standar dalam autentikasi pesan internet (Deepakumara, Heys, & Ventakesan, 2008). Fungsi *hash* ini menerima input pesan dengan ukuran acak dan menghasilkan output 128-*bit* message digest dari input. Fungsi *hash* ini terutama digunakan untuk aplikasi digital signature dimana sebuah data yang berukuran besar harus dikompres dengan aman sebelum dienkripsi dengan kunci privat dibawah kunci public kriptosistem.

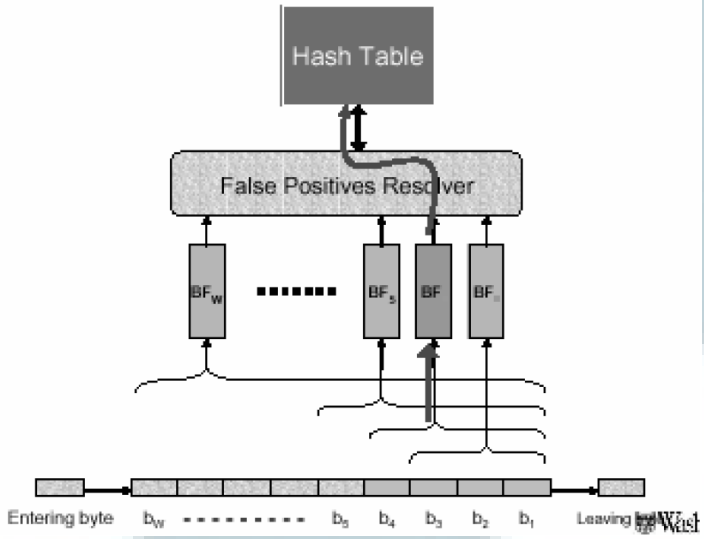
2.4.3 Secure Hash Algorithm 256

Secure Hash Algorithm 256-*bit* (SHA256) pada awalnya dipublikasi oleh National Security Agency (NSA) pada tahun 2001 (Bruce, 2011). Fungsi *hash* ini adalah varian 256-*bit* dari keluarga algoritma SHA-2. Keluarga SHA-2 ini sendiri merupakan penerus dari SHA-1, yang sudah diketahui rusak secara umum dikarenakan banyaknya cara penyerangan terhadap SHA-1. SHA-1 sudah tidak dipergunakan oleh pemerintah Amerika Serikat untuk digital signature pada akhir 2010.

2.5 Bloom Filter

Bloom filter (*Bloom*, 1970) adalah struktur data probabilistic yang tersusun rapat dan dipergunakan untuk menentukan apakah sebuah elemen berada di dalam suatu set tertentu. Hasil dari tes tersebut memiliki kemungkinan untuk mengembalikan nilai benar untuk suatu elemen yang sebenarya tidak ada didalam set tersebut (*false positive*s), tetapi tidak pernah mengembalikan nilai salah untuk element yang ada di dalam set (*false* negatives). Hal ini membuat *Bloom* fiter berguna untuk berbagai macam pekerjaan yang melibatkan list dan set. Operasi dasar dari *Bloom filter* meliputi penambahan elemen ke dalam set dan *query* kemungkinan adalahnya suatu elemen dalam suatu set atau list. Contoh penggunaan *Bloom filter* ini adalah pada Network Intrusion Detection System (NIDS), dimana *Bloom filter* akan mendeteksi tanda-tanda serangan yang sudah ditetapkan dalam payload paket. Pemetaan dikelompokan menurut panjangnya (dalam byte) dan disimpan dalam *Bloom filter* paralel. Masing-masing *Bloom filter* melakukan pengecekan pada data streaming untuk *string* dengan panjang

yang sesuai. Ketika *Bloom filter* mendeteksi *string* apapun yang mencurigakan pada paket, sebuah analyzer akan memutuskan apakah *string* tesebut merupakan salah satu dari set atau deteksi tersebut adalah *false positive*. Analyzer adalah algoritma pencocokan *string* deterministik yang memastikan apakah suatu *input* elemen merupakan anggota dari suatu set atau bukan. Ketika ada *string* yang mencurigakan ditemukan, tindakan *drop*, *forward* dan *log* dapat dilakukan kepada paket. (Soliman & El-Helw ,2005).



Gambar 2.1 Ilustrasi implementasi *Bloom filter* pada Network Intrusion Detection System dengan pemetaan panjang byte yang berbeda (Soliman & El-Helw, 2005)

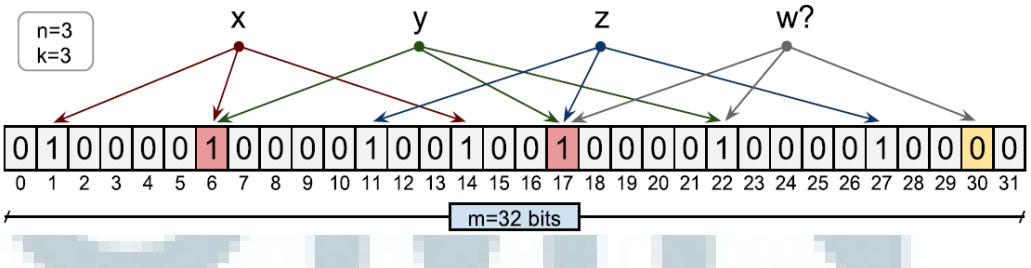
Tingkat akurasi dari *Bloom filter* bergantung pada ukuran dari *filter*, jumlah fungsi *hash* yang dipergunakan pada *filter*, dan jumlah element yang dimasukan ke dalam set. Semakin banyak elemen yang ditambahkan ke dalam *Bloom filter* akan meningkatkan kemungkinan hasi *query* yang *false positives.*

Broder dan Mitzenmancher telah merangkum apa yang menjadi prinsip *Bloom filter* (Broder & Mitzenmacher, 2003) :

"Ketika anda menggunakan list dan set dan besarnya tempat menjadi pertimbangan, pertimbangkan untuk menggunakan *Bloom filter*. Dan ketika anda menggunakan *Bloom filter* pertimbangkan juga akan kemungkinan terjadinya *false positives*."

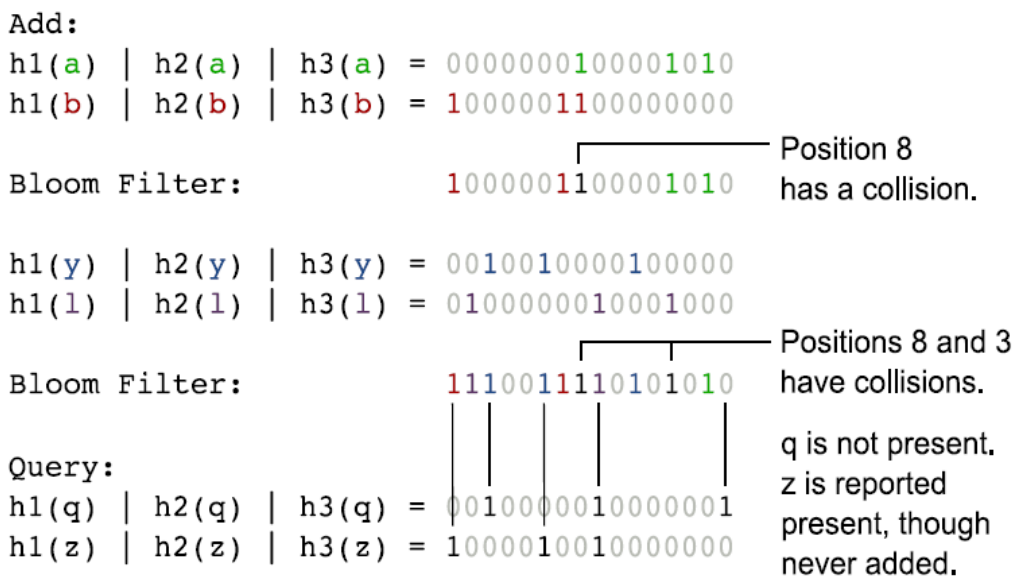
Bloom filter adalah sebuah *array* dari *m bit* yang mempresentasikan set $S=\{x_1,x_2,....,x_n\}$ dari n element. Awalnya, semua *bit* yang ada di dalam *filter* diset menjadi nol. Kuncinya adalah menggunakan fungsi *hash k*, $h_1(x)$, dimana $1 \leq i \leq k$ untuk memetakan $x \in S$ ke dalam angka random dari 1 ... *m*. Fungsi *hash* yang dipergunakan diasumsikan sama. Algoritma *hash* MD5 adalah pilihan yang popular sebagai fungsi *hash*.

Sebuah element $x \in S$ dimasukan ke dalam *filter* dengan meng-set *bit* $h_1(x)$ ke dalam $1 \leq i \leq k$. Sebaliknya, *y* diasumsikan sebagai salah satu member dari *S* jika *bit* $h_1(y)$ telah diset. Dan pasti bukan merupakan member bila $h_1(y)$ tidak diset. Kelemahan dari penggunaan *Bloom filter* adalah adanya kemungkinan ternyadinya *false positives*. *False positives* adalah elemen yang bukan merupakan bagian dari set *S* tetapi dilaporkan ada di dalam set *S* oleh *filter*.



Gambar 2.2 Ilustrasi *Bloom Filter* (Tarkoma, Rothenberg, & Lagerspetz, 2010)

Pada Gambar diatas dijelaskan gambaran *Bloom filter*. *Bloom filter* terdiri dari *bitstring* dengan pangkat 32. Tiga elemen telah dimasukan, yaitu *x, y,* dan *z*. Tiap elemen telah di *hash* dengan menggunakan fungsi *hash k* = 3 ke dalam posisi *bit* dalam *bitstring*. *Bit* yang berkaitan akan diset menjadi 1. Kemudian ketika sebuah elemen yang tidak ada di dalam set, yaitu *w*, *w* juga akan di-*hash* dengan menggunakan tiga fungsi *hash* ke dalam posisi *bit*. Pada kasus ini, salah satu posisi *bit*-nya adalah nol, sehingga *Bloom filter* melaporkan dengan benar bahwa elemen tersebut tidak berada di dalam set. Ada kemungkinan bila semua posisi *bit* dari sebuah elemen menunjukan bahwa *bit* tertentu telah di set, ketika ini terjadi *Bloom filter* akan memberikan laporan yang salah bahwa elemen tersebut berada di dalam set. Laporan yang salah ini disebut sebagai *false positives.*



Gambar 2.3 Contoh Penambahan dan *Query Bloom Filter*

(Tarkoma, Rothenberg, & Lagerspetz, 2010)

Gambar diatas menggambarkan sebuah contoh dari *Bloom filter* melalui penambahan dan *query* elemen. Pada contoh berikut, dipergunakan *bitstring* dengan panjang 16. Posisi dari *bit* diberi angka nol sampai dengan lima belas, dari kanan ke kiri. Tiga fungsi *hash* yang digunakan : h_1, h_2, dan h_3 adalah MD5, SHA1, dan CRC32. Element yang ditambahkan adalah *string text* yang hanya memiliki satu huruf. *Bloom filter* dimulai dengan semua posisi *bit* diset menjadi nol. Ketika menambahkan elemen, nilai dari *h1*, *h2,* dan *h3* (*modulus* 16) dikalkulasi untuk elemen, dan *bit* yang berkorespondensi diset menjadi satu. Setelah menambahkan element *a* dan *b, Bloom filter* memiliki posisi 15, 9, 8, 3, dan 1 diset. Pada kasus ini *a* dan *b* memiliki satu posisi *bit* yang sama, yaitu 8. Kemudian ditambahkan lagi elemen baru, *y* dan l. Setelah penambahan tersebut posisi 15, 14, 13, 10, 9, 7, 5, 3, dan 1 diset. Ketika dilakukan *query* terhadap *q* dan *z,* fungsi *hash* yang sama dipergunakan. Posisi *bit* yang sesuai dianalisa. Bila tiga *bit* dari elemen diset, maka elemen tersebut dianggap ada di dalam set. Pada saat *q* di-*query*, posisi 0 tidak diset, sehingga *q* dianggap tidak ada didalam set oleh *Bloom filter*, akan tetapi *z* diasumsikan ada didalam *filter*, karena *bit* yang berkorespondensi telah di set. Nilai *z* adalah *false positives*.

Untuk kinerja optimal, tiap fungsi *hash k* harus menjadi anggota dari kelas fungsi *hash universal*, yang berarti tiap fungsi *hash* memetakan tiap komponen yang ada secara beragam. Solusi yang hampir ideal mengenai fungsi *hash* sudah ada (Ostin & Pagh, 2003). Fungsi *hash* yang memiliki output yang beragam seperti MD5 atau CRC32 adalah yang paling cocok untuk *filter* yang berbasis probabilitas.

Bloom filter dibangun berdasar pada *S* memiliki ruang *O(n)* dan dapat menjawab *query* member pada *O(1)* waktu. Bila $x \in S$, *Bloom filter* akan selalu memberitahukan bahwa *x* termasuk dalam *S*, tetapi bila $y \notin S$, ada kemungkinan *Bloom filter* memberitahukan bahwa $y \in S$.

Parameter	Penambahan
Jumlah fungsi *hash* (*k*)	Menambah waktu komputasi, mengurangi *false positive rate* $k \rightarrow k_{opt}$
Ukuran *filter* (*m*)	Ruang yang diperlukan bertambah, *false positive*s *rate* berkurang.
Jumlah elemen yang ada pada set (*n*)	*False positive rate* bertambah.

Tabel 2.1 Kunci Parameter *Bloom Filter*

(Tarkoma, Rothenberg, & Lagerspetz, 2010)

Tabel 2.1 memberikan gambaran tiga kunci parameter beserta konsekuensi yang didapat ketika nilai dari parameter tersebut bertambah atau berkurang. Dengan menambah atau mengurangi nilai dari fungsi *hash* terhadap k_{opt} dapat mengurangi rasio *false positive*s namun menambah beban pada waktu komputasi yang diperlukan untuk pencarian elemen. Beban yang diperlukan berbanding lurus dengan jumlah fungsi *hash* yang dipergunakan. Besarnya ukuran *filter* juga dapat dipergunakan untuk menyempurnakan kebutuhan ruang yang diperlukan dan *false positive rate* (FPR). *Filter* yang lebih besar akan menghasilkan *false positive* yang lebih kecil, dan yang terakhir adalah ukuran dari set yang dimasukan pada *filter* juga menentukan *false positive rate.*

Kemungkinan terjadinya *false positive rate* dapat dikalkulasi dengan mengasumsikan bahwa fungsi *hash* memilih setiap posisi di dalam set (*array*) dengan probabilitas yang sama. *m* adalah jumlah *bit* di dalam *Bloom filter* .

Ketika memasukan elemen ke dalam *filter*, kemungkinan *bit* tertentu tidak diset menjadi 1 oleh fungsi *hash* adalah

$$1 - \frac{1}{m} \quad \text{..... (Rumus 2.4)}$$

Dengan mengasumsikan bahwa *k* adalah jumlah fungsi *hash*, maka kemungkinan tidak diset menjadi satu adalah

$$(1 - \frac{1}{m})^k \quad \text{..... (Rumus 2.5)}$$

Setelah memasukkan sejumlah *n* elemen ke dalam *filter*, kemungkinan bahwa set masih nol adalah

$$(1 - \frac{1}{m})^{kn} \quad \text{..... (Rumus 2.6)}$$

Untuk sebuah elemen dalam membership *test*, bila semua yang ada di posisi *array k* dalam *filter* yang telah dikomputasi diset menjadi 1, *Bloom filter* mengasumsikan bahwa elemen ada di dalam set. Probabilitass bahwa kejadian ini terjadi ketika elemen sebenarnya tidak di dalam set adalah.

$$(1 - \left(1 - \frac{1}{m}\right)^{kn})^k \approx (1 - e^{-\frac{kn}{m}})^k \text{ ..... (Rumus 2.7)}$$

Mengetahui bahwa $e^{-\frac{kn}{m}}$ mendekati $(1 - \frac{1}{m})^{kn}$ (Broder & Mitzenmacher, 2003), kemungkinan terjadinya *false positive* (FPR) berkurang seiring dengan bertambahnya ukuran *m* dari *Bloom filter*. FPR bertambah berbanding lurus dengan *n* elemen yang bertambah. Dengan mengoptimalkan jumlah *k* fungsi *hash*,

dapat meminimalisir terjadinya *false positive* dengan mengambil *derivative* dan menyetarakan dengan nol, yang memberikan nilai optimal dari *k*.

$$K_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \quad \text{..... (Rumus 2.8)}$$

Sehingga didapatkan bahwa kemungkinan terjadinya FPR adalah.

$$(\frac{1}{2})^k \approx 0{,}6185^{m/n} \quad \text{..... (Rumus 2.9)}$$

Dengan menggunakan angka optimal dari *hash* k_{opt}, FPR dapat ditulis ulang menjadi.

$$\frac{m}{n} \geq \frac{1}{ln2} \quad \text{..... (Rumus 2.10)}$$

Dengan persamaan ini, diketahui bahwa untuk memiliki nilai FPR tertentu, ukuran *Bloom filter* harus berbanding lurus dengan jumlah elemen yang dimasukan ke dalam *filter*. Jumlah *bit m* untuk jumlah elemen *n* dan FPR *p*, adalah.

$$m = -\frac{n \ln p}{(ln2)^2} \quad \text{..... (Rumus 2.11)}$$

Berdasarkan rumus 2.9, untuk mendapatkan jumlah fungsi *hash* berdasarkan nilai FPR (*p*) tertentu adalah.

$$p = (\frac{1}{2})^k$$

$$k = \log_{\frac{1}{2}} p$$

$$k = \log_{2^{-1}} p$$

$$k = \log_2 p^{-1}$$

$$k = \log_2 \frac{1}{p} \quad \text{..... (Rumus 2.12)}$$

Kemudian berdasarkan rumus 2.7 dan rumus 2.8 jumlah elemen maksimal atau kapasitas maksimal (*n*) yang dapat ditampung oleh *Bloom filter* secara optimal adalah.

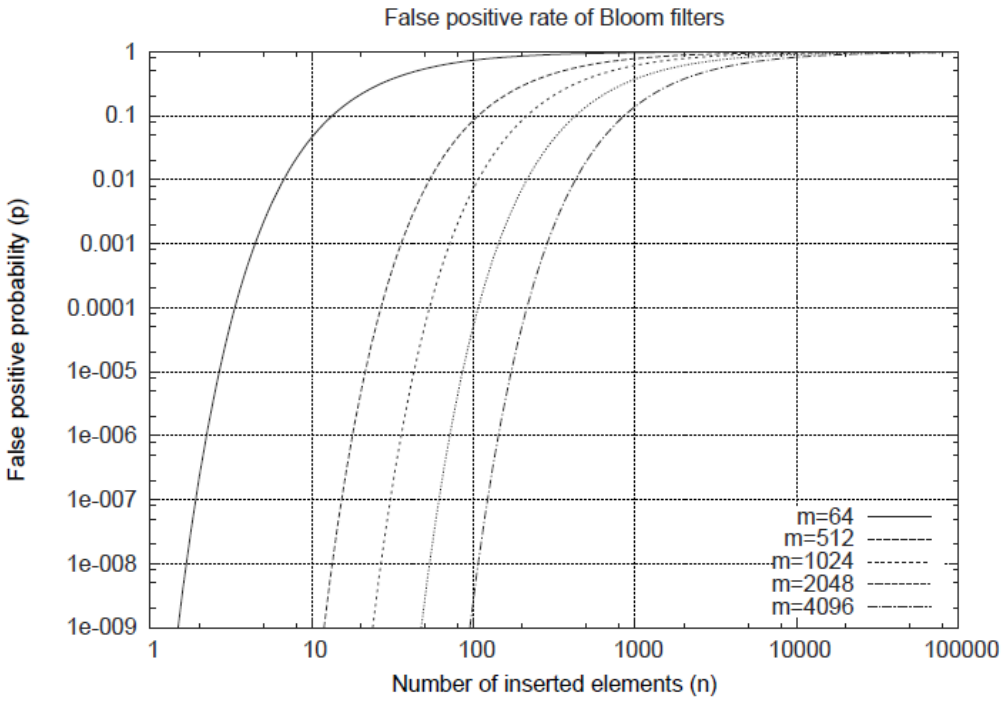
$$p = (1 - \left(1 - \frac{1}{m}\right)^{kn})^k, k = \frac{m}{n} \ln 2$$

$$p = (1 - \left(1 - \frac{1}{m}\right)^{\frac{m}{n} \ln 2 \, n})^{\frac{m}{n} \ln 2}$$

$$\log_{1-(1-\frac{1}{m})^{m \ln 2}} p = \frac{m \ln 2}{n}$$

$$n = \frac{m \ln 2}{\log_{1-(1-\frac{1}{m})^{m \ln 2}} p} \quad \text{..... (Rumus 2.13)}$$

Pada gambar 2.4 diketahui bahwa FPR *p* sebagai sebuah fungsi untuk sejumlah elemen *n* dalam *filter* dan *filter* memiliki ukuran sebesar *m*. Asumsi yang dipakai adalah jumlah fungsi *hash* yang optimal adalah k = (m/n) ln 2.



Gambar 2.4 False Positive Rate dari Bloom Filter

(Tarkoma, Rothenberg, & Lagerspetz, 2010)