Pembangkit Kunci Acak pada *One-Time Pad* Menggunakan Fungsi *Hash* Satu-Arah

Junita Sinambela (13512023)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
sinambelajunita@gmail.com

***Abstract*—Salah satu algoritma kriptografi yang tidak dapat dipecahkan adalah menggunakan *One-Time Pad* (OTP). OTP menggunakan kunci yang benar-benar random dan memiliki panjang yang sama dengan panjang *plaintext* untuk dapat melakukan enkripsi pesan. Makalah ini berisi alternatif pembangkit kunci *random* dengan menggunakan fungsi *hash* satu-arah.**

Kata Kunci*—One-Time Pad *(OTP), fungsi* hash, message digest, *kunci acak

I. PENDAHULUAN

Salah satu tantangan dalam dunia kriptografi adalah menciptakan algoritma yang menghasilkan *unbreakable cipher*. Untuk menghasilkan *unbreakable cipher*, algoritma kriptografi harus memenuhi syarat yaitu kunci harus benar-benar acak dan panjang kunci harus sama dengan panjang *plaintext*. Algoritma kriptografi klasik seperti *Caesar Cipher*, *Vigènere Cipher*, ataupun *Playfair Cipher* terbukti sudah dapat dipecahkan kriptanalis, baik menggunakan teknik analisa frekuensi, ataupun melakukan *brute force* terhadap kemungkinan kunci.

Salah satu algoritma yang diyakini mampu menghasilkan *unbreakable cipher* adalah *One-Time Pad* (OTP). OTP ditemukan pada tahun 1917 oleh Major Joseph Mauborgne. Adapun *pad* yang digunakan berisi deretan karakter-karakter kunci yang dibangkitkan secara acak. OTP sendiri merupakan algoritma kriptografi simetri, sehingga penerima pesan harus memiliki salinan *pad* yang sama dengan *pad* yang digunakan oleh pengirim pesan untuk melakukan dekripsi pesan agar mendapatkan *plaintext*.

Setiap *pad* hanya boleh digunakan sekali saja untuk mengenkripsi satu pesan. Oleh karena itu, setiap kali *pad* digunakan untuk mengenkripsi satu pesan, *pad* tersebut harus dihanguskan supaya tidak dipakai kembali untuk mengenkripsi pesan yang lain. Hal ini dimaksudkan untuk mencegah kriptanalis menemukan pola enkripsi pada lebih dari satu pesan.

Kelemahan algoritma OTP terletak pada ketidakmangkusannya dalam mengenkripsi pesan. Pada OTP, panjang kunci harus sama dengan panjang pesan, sehingga menyulitkan untuk melakukan penyimpanan dan pendistribusian kunci. Secara teoritis, tidak mungkin pengirim dan penerima dapat membangkitkan kunci acak yang sama. Hal ini menyebabkan kunci harus dikirimkan oleh pengirim ke penerima.

Maka, untuk mengatasi kelemahan tersebut, pada makalah ini diajukan solusi pembangkitan kunci untuk OTP dengan menggunakan fungsi *hash* satu-arah. Fungsi *hash* satu-arah memiliki sifat *irreversible*, sehingga menyulitkan kriptanalis untuk menebak kunci yang digunakan.

II. TEORI SINGKAT

A. *Algoritma Kriptografi* One-Time Pad *(OTP)*

Pad yang digunakan sebagai sumber kunci acak merupakan kunpulan karakter alfabetik. Algoritma kriptografi OTP persis sama dengan algoritma kriptografi *Vigènere Cipher*. Setiap karakter pada pesan dienkripsi menggunakan bujur sangkar *Vigènere*.

Pada Gambar 1, c_i merupakan *ciphertext* pada indeks ke-i, sedangkan p_i merupakan *plaintext* pada indeks ke-i, dan k_i merupakan kunci pada indeks ke-i. Enkripsi pada OTP menggunakan *pad* dengan panjang kunci sama dengan panjang *plaintext*. OTP sendiri menggunakan algoritma ini dengan asumsi semua karakter kuncinya adalah huruf alphabet, sehingga enkripsi menggunakan bujur sangkar *Vigènere* dapat dilakukan.

Enkripsi: $c_i = (p_i + k_i) \bmod 26$

Dekripsi: $c_i = (p_i - k_i) \bmod 26$

Gambar 1 Algoritma kriptografi *Vigènere Cipher* versi *standard*

Pada Gambar 2, bagian kolom merepresentasikan kunci, dan bagian baris merepresentasikan karakter *plaintext*. Untuk *Vigènere Cipher* jenis *standard*, kunci dan *plaintext* hanya dapat berupa alfabet, sehingga implementasi algoritma ini terbatas. Untuk itu, *Vigènere Cipher* dapat dikembangkan menjadi versi extended, yaitu menggunakan karakter ASCII.

Karakter ASCII memiliki 256 jenis karakter yang direpresentasikan dalam desimal yang terurut, sehingga *Vigènere Cipher* dapat digunakan untuk melakukan enkripsi pesan yang terdiri dari karakter ASCII. Algoritma enkripsi dan dekripsi untuk *Vigènere Cipher* versi *extended* (ASCII) dapat dilihat pada Gambar 3.

Gambar 2 Bujur sangkar Vigènere

$$\text{Enkripsi: } c_i = (p_i + k_i) \bmod 256$$

$$\text{Dekripsi: } c_i = (p_i - k_i) \bmod 256$$

Gambar 3 Algoritma kriptografi *Vigènere Cipher* versi *extended* (ASCII)

B. Fungsi Hash *Satu-Arah*

Fungsi *hash* merupakan fungsi yang menerima masukan *string* yang panjangnya sembarang dan menghasilkan *string* yang panjangnya konstan walaupun ukuran string masukannya berbeda-beda. *String* keluaran dari fungsi *hash* dinamakan *message digest*. Fungsi ini digunakan untuk melakukan kompresi terhadap suatu pesan atau berkas digital.

Fungsi *hash* dapat digunakan untuk melakukan pengecekan integritas suatu pesan. Suatu pesan yang diubah dan berbeda dengan pesan aslinya akan memiliki *message digest* yang berbeda dengan *message digest* pesan aslinya.

Fungsi *hash* satu-arah (*one-way hash function*) merupakan fungsi *hash* yang bekerja satu arah. Prinsip fungsi *hash* satu-arah adalah *message digest* suatu pesan tidak dapat dikembalikan lagi menjadi bentuk semula (*irreversible*). Ada beberapa sifat fungsi hash satu arah, yaitu:

a) dapat diterapkan pada blok data berukuran berapa saja,

b) menghasilkan *message digest* dengan panjang konstan,

c) mudah dikomputasikan,

d) tidak dapat mengembalikan *message digest* menjadi pesan asalnya,

e) tidak menghasilkan *message digest* yang sama untuk dua pesan yang berbeda, dan

f) tidak mungkin mencari pasangan dua pesan sedemikian yang memiliki *message digest* yang sama.

Message digest pada fungsi *hash* satu-arah dapat direpresentasikan dalam berbagai bentuk. Perhitungan di dalam fungsi *hash* satu-arah sebagian besar menggunakan perhitungan *byte*. Tujuan penghitunag *message digest* menggunakan fungsi *hash* satu-arah salah satunya adalah melakukan pengecekan integritas pada suatu pesan, sehingga *message digest* biasanya diubah dalam bentuk heksadesimal untuk memudahkan pengguna dalam menganalisis. Dalam implementasinya, pengguna dapat mengubah representasi *message digest* menjadi *array of byte*, atau dapat di-*encode* menjadi bentuk lain, salah satunya adalah Base64.

C. Penambahan Salt

Pada awalnya, *salt* digunakan sebagai salah satu indikator aspek keamanan pada *password*. Kebanyakan implementasi sistem terdistribusi tidak menyimpan *password* pengguna secara utuh, melainkan menyimpan *message digest* dari *password*. Oleh sebab itu, jika ada dua pengguna dengan *password* yang sama, dapat dipastikan bahwa *message digest* yang disimpan oleh sistem adalah sama.

Untuk mengatasi hal ini, *password* yang ingin dihitung *message digest*-nya terlebih dahulu disisipi dengan *salt*. Salt berupa string yang dapat disisipkan pada bagian tertentu dari *password* sebelum dihitung *message digest*-nya. Salt menggunakan *string* acak dengan ukuran tak terbatas, ditujukan untuk menambah *cost* yang dibutuhkan untuk melakukan *brute force* kemungkinan *password*. Dalam implementasinya, *salt* tidak dirahasiakan. *Salt* biasanya disimpan bersamaan dengan *message digest* dari *password* yang sudah disisipkan *salt*.

Salt yang digunakan berbeda-beda untuk setiap kali penggunaan. Hal ini menyebabkan sistem harus membangkitkan string acak untuk setiap *plaintext* yang akan dihitung *message digest*-nya. Karena *salt* tidak perlu dirahasiakan, maka sistem dapat melakukan pembangkitan *string* secara acak untuk dijadikan *salt*. Setelah digunakan, *salt* disimpan untuk digunakan nantinya jika dibutuhkan untuk melakukan pengecekan *password*.

III. RANCANGAN PEMBANGKIT KUNCI OTP

Dalam membangkitkan kunci acak pada OTP, digunakan fungsi *hash* satu-arah. Fungsi *hash* satu-arah memiliki sifat yang dapat dimanfaatkan untuk membangkitkan kunci, yaitu *irreversible*. Selain itu, untuk membangkitkan kunci OTP dengan ukuran yang sesuai dengan panjang pesan yang ingin dienkripsi, dibutuhkan *seed* yang ukurannya sesuai dengan keinginan pengguna.

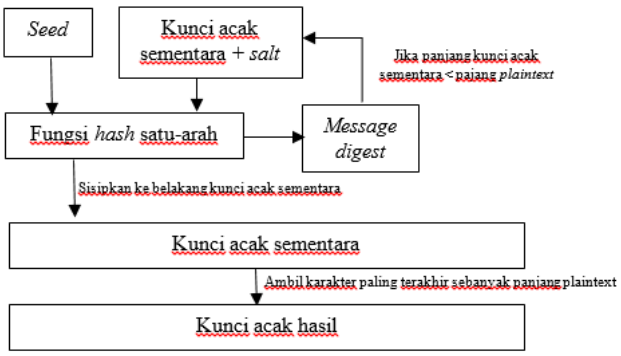
Fungsi *hash* yang digunakan untuk membangkitkan kunci OTP dapat menggunakan fungsi *hash* satu-arah yang sudah ada, seperti SHA1, MD5, atau kelompok SHA2 (SHA-256, SHA-521), dan lain-lain. Dalam memilih fungsi *hash* satu-arah yang akan digunakan, pengguna disarankan untuk memilih fungsi yang dapat menghasilkan perbedaan yang cukup signifikan jika *seed* yang digunakan diubah sedikit saja. Selain itu, pertimbangan dalam menentukan fungsi *hash* satu-arah yang akan digunakan adalah panjang *message digest* yang dihasilkan. Semakin panjang *message digest* yang dihasilkan, maka semakin sulit dipecahkan.

```
compute message digest of seed
append message digest to result
while length of result < length of plaintext
  add salt to result
  compute message digest of result that have
    been salted
  append message digest to result
```

Gambar 4 *Pseudo-code* algoritma pembangkit kunci acak pada OTP

Seed sebagai masukan pengguna dihitung *message digest*-nya. *Message digest* ini dimasukkan sebagai kunci acak pada algoritma kriptografi OTP. Kunci acak tersebut kemudian dicek apakah lebih panjang atau sama dengan panjang *plaintext*. Jika kondisi tersebut tidak terpenuhi, kunci acak akan diberi *salt* dan kemudian *message digest* dari hasil penambahan salt tersebut dihitung kembali. *Message digest* kedua ini akan digabungkan dengan kunci acak pertama. Proses penambahan *salt* pada kunci acak dan penghitungan *message digest* ini akan diulang sampai panjang kunci acak lebih dari atau sama dengan panjang *plaintext.*

Saat panjang kunci acak OTP melebihi panjang *plaintext*, kunci acak yang nantinya akan digunakan sebagai kunci yang valid adalah karakter sebanyak `n` terakhir pada kunci acak, dengan `n` merupakan panjang *plaintext.*



Gambar 5 Skema solusi pembangkit kunci acak pada OTP

A. *Fungsi* Hash *Satu-Arah*

Ada banyak fungsi *hash* satu-arah yang telah dikembangkan. Beberapa diantaranya masih dipertanyakan dan diuji apakah menimbulkan *collision* apa tidak.

Collision adalah keadaan di mana dua pesan yang berbeda memiliki *message digest* yang sama. Hal ini dapat terjadi, mengingat ukuran *message digest* yang konstan berapapun ukuran pesannya. Beberapa fungsi *hash* satu-arah yang sudah terbukti mengalami *collision* pada panjang masukan tertentu adalah MD5 dan SHA0.

Pada Gambar 6, dapat dilihat bahwa beberapa fungsi hash satu-arah sudah diketahui *collision* yang terjadi dari hasil eksperimen yang dilakukan. Hal ini mempertegas bahwa ada fungsi *hash* satu arah yang dapat diserang.

Algoritma	Ukuran *message digest* (bit)	Ukuran blok pesan	Kolisi
MD2	128	128	Ya
MD4	128	512	Hampir
MD5	128	512	Ya
RIPEMD	128	512	Ya
RIPEMD-128/256	128/256	512	Tidak
RIPEMD-160/320	160/320	512	Tidak
SHA-0	160	512	Ya
SHA-1	160	512	Ada cacat
SHA-256/224	256/224	512	Tidak
SHA-512/384	512/384	1024	Tidak
WHIRLPOOL	512	512	Tidak

Gambar 6 Beberapa jenis fungsi *hash* satu-arah dan ciri-cirinya. Sumber dari [3]

B. *Penambahan* Salt

Salt digunakan untuk menciptakan perubahan yang signifikan pada pesan yang akan dihitung *message digest*-nya. *Salt* adalah *string* yang akan disisipkan pada bagian yang sudah ditentukan. Pada algoritma ini, setiap kali *message digest* akan dihitung, salt ditambahkan oleh pengguna algoritma sesuai keinginan. Bagian ini dapat menjadi alternatif penggunaan pembangkit kunci acak pada OTP ini.

IV. Eksperimen dan Analisis

Algoritma ini diimplementasikan pada bahasa pemrograman Java, menggunakan fungsi *hash* SHA1, dan menggunakan *extended Vigènere Cipher* (untuk karakter ASCII). Hal ini disebabkan karena pada eksperimen yang dilakukan, hasil *message digest* direpresentasikan dalam bentuk *string* heksadesimal, sehingga kemungkinan karakter pada kunci acak adalah karakter heksadesimal. Oleh karena itu, *Vigènere Cipher* versi *standard* tidak dapat diimplementasikan untuk eksperimen ini. Ada beberapa contoh kasus pengujian dengan *seed* yang berbeda yang dilakukan, yaitu pada Tabel 1.

Tabel 1 Perbandingan beberapa *seed* dan kunci acak yang dibangkitkan

No	Seed	Panjang *plaintext*	Kunci Acak (dalam format heksadesimal)
1	`nih contoh key`	38	`32074fa2018e195f816812 36047446c48834ea`
2	`nih contoh key1`	38	`f9b40ce14e1cbe4a02634f 24936a3af75a86aa`
3	`nih contoh key2`	38	`1e73fbbf1b8b5a34002313 75194a73aa51b54c`
4	`Nih contoh key`	38	`da7c67f7d0650e62d25dea 99a9d6b42b215e3a`

Pada Tabel 1, dapat dilihat bahwa sedikit perubahan pada *seed* dapat menyebabkan perubahan yang signifikan pada kunci acak yang dihasilkan. Keseluruhan karakter pada kunci acak terganti. Bahkan, pada kunci pertama dan keempat, perbedaan

seed hanya terletak pada awal huruf pada kalimat saja, dan seluruh karakter pada kunci acak terganti.

Tabel 2 Pengujian enkripsi menggunakan *extended Vigènere Cipher*

Plaintext:	`nih contoh message`
Seed:	`nih contoh key`
Random Key Generated:	`1236047446c48834ea`
Ciphertext (dilakukan *encode* ke Base64):	`n5ubVpOjpaijnoOhnaumlczG`

Tabel 3 Pengujian enkripsi menggunakan *extended Vigènere Cipher*

Plaintext:	`nih contoh message`
Seed:	`nih contoh key1`
Random Key Generated:	`4f24936a3af75a86aa`
Ciphertext (dilakukan *encode* ke Base64):	`os+aVJyipNWiyYakmtSrl8jG`

Pada Tabel 2 dan Tabel 3, dapat dilihat hasil eksperimen yang dilakukan pada algoritma *Vigènere Cipher* menggunakan pembangkit kunci acak. Analisis frekuensi hanya dapat dilakukan dengan mengecek frekuensi *byte* pada *ciphertext*, dengan melakukan *decode* dari Base64 terlebih dahulu. Setiap jenis karakter pada plaintext akan dienkripsi dengan kunci yang berbeda, sehingga menutup celah analisis kunci menggunakan analisis frekuensi.

Seed yang digunakan untuk melakukan pembangkitan kunci acak berukuran bebas, sehingga menyulitkan kriptanalis untuk melakukan *brute force*. *Seed* dengan ukuran apapun akan dapat menghasilkan kunci acak dengan ukuran yang sesuai dengan ukuran *plaintext* yang akan dienkripsi. *Seed* dapat berupa *array of byte*, sehingga berkas apapun juga dapat dimanfaatkan sebagai *seed*. Untuk melakukan pendistribusian kunci, cukup mendistribusikan *seed* saja. *Seed* dapat didistribusikan menggunakan algoritma kunci asimetri, salah satunya adalah RSA.

Jika seorang kriptanalis dapat menebak *message digest* dari *seed* dengan melakukan *brute force* saja tanpa menebak *seed*, maka dapat dipastikan kunci acak dapat diketahui. Hal ini menyebabkan pentingnya penambahan *salt* pada akhir *string* yang akan dihitung *message digest*-nya, sehingga *brute force* yang dilakukan menjadi semakin lama dan *cost* untuk melakukan pemecahan kunci acak semakin besar.

Salt yang ingin ditambahkan disetujui terlebih dahulu oleh kedua pihak yang ingin berkomunikasi. Hal ini disebabkan karena secara teoretis, tidak mungkin *string* yang dibangkitkan secara acak pada dua pihak yang berbeda dapat menghasilkan *string* yang sama persis. *Salt* disarankan untuk memiliki ukuran yang besar, sehingga kriptanalis yang ingin memecahkan kunci acaknya kesulitan dalam melakukan *brute force*. *Salt* juga disarankan berbeda-beda untuk setiap kali penggunaan, sehingga memperbesar *cost* yang dibutuhkan untuk memecahkan kunci.

Salah satu cara untuk memperbesar *cost* untuk memecahkan kunci adalah dengan mengambil `n` terakhir dari kunci acak yang telah dibangkitkan untuk melakukan enkripsi *plaintext*, dengan `n` adalah panjang *plaintext*. Jika kriptanalis melakukan *brute force* dimulai dari bagian terakhir *ciphertext*, akan mempersulit kriptanalis untuk menebak kunci acak sebelumnya, bahkan setelah mendapatkan kunci acak yang benar dengan ukuran yang sama dengan panjang *message digest* yang digunakan. Hal ini disebabkan karena sifat fungsi *hash* satu-arah yang digunakan untuk menghitung *message digest*, yaitu *irreversible*. Jika kriptanalis melakukan *brute force* dari bagian awal *ciphertext*, maka ia harus menebak kemungkinan awalan yang dihilangkan dari kunci acak yang telah dibangkitkan, sehingga menambah *cost*. Untuk panjang *plaintext* dengan kelipatan panjang *message digest*, cost dapat diperbesar dengan menggunakan *salt*.

OTP sendiri dapat dipecahkan menggunakan *brute force* semua kemungkinan kunci untuk setiap karakter. Dalam kasus ini, kompleksitas untuk menemukan kunci acak pada suatu cipher adalah `O(n)`, dimana `n` merupakan panjang *plaintext*. Untuk melakukan *brute force* kemungkinan message digest dari *seed*, kompleksitas untuk menemukan `n` kunci acak pertama adalah `O(n)`, dimana `n` merupakan panjang *message digest* yang digunakan. Untuk menemukan salt yang digunakan juga kemungkinannya tidak berhingga, sehingga memperbesar *cost* yang dibutuhkan oleh kriptanalis untuk melakukan *brute force*.

Selain itu, adanya ketergantungan antar *message digest* menunjukkan diterapkannya konsep *diffusion* dari Shannon pada pembangkit kunci. Hal ini menyebabkan ketergantungan kunci acak pada satu blok sangat bergantung pada kunci acak pada blok sebelumnya. Konsep *confusion* dari Shannon juga ada pada algoritma ini, karena semua kuncinya dianggap benar-benar acak. *Seed* yang digunakan untuk melakukan pembangkitan kunci acak berukuran bebas, sehingga menyulitkan kriptanalis untuk melakukan *brute force*. *Seed* dengan ukuran apapun akan dapat menghasilkan kunci acak dengan ukuran yang sesuai dengan ukuran *plaintext* yang akan dienkripsi. *Seed* dapat berupa *array of byte*, sehingga berkas apapun juga dapat dimanfaatkan sebagai *seed*. Untuk melakukan pendistribusian kunci, cukup mendistribusikan *seed* saja. *Seed* dapat didistribusikan menggunakan algoritma kunci asimetri, salah satunya adalah RSA.

V. KESIMPULAN DAN SARAN

Dalam melakukan pembangkitan kunci acak pada algoritma kriptografi OTP, pengguna dapat menggunakan fungsi *hash* satu-arah. Sifat *irreversible* pada fungsi hash satu-arah menjadikan fungsi tersebut kandidat yang kuat untuk melakukan pembangkitan kunci acak. Kunci yang dihasilkan tidak

sepenuhnya acak, karena dapat dibangkitkan kembali menggunakan *seed* yang ditentukan oleh pengguna.

Dalam melakukan pemecahan kunci acak, ada dua cara yang dapat digunakan oleh kriptanalis, yaitu *brute force* dan analisis keterkaitan antar *message digest*. Namun, karena adanya penambahan salt, mengakibatkan keterkaitan antar *message digest* menjadi sulit dilakukan, karena ada konsep *confusion* dan *diffusion* yang diterapkan pada pembangkit kunci acak. Maka, untuk menggunakan algoritma pembangkit kunci ini, sebaiknya digunakan fungsi *hash* satu-arah yang diketahui tidak memiliki *collision*, *message digest* yang dihasilkan berukuran cukup panjang, dan *salt* yang digunakan cukup kuat.

VI. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Bapak Rinaldi Munir sebagai dosen mata kuliah IF4020 Kriptografi yang telah membimbing dalam perkuliahan sehingga penulis dapat menyelesaikan makalah ini. Ucapan terima kasih juga penulis sampaikan kepada seluruh pihak yang telah membantu penulis dalam penulisan makalah ini. Semoga makalah ini dapat bermanfaat pada bidang keilmuan informatika, khususnya pada bidang kriptografi.

Referensi

[1] http://techglimpse.com/sha256-hash-certificate-openssl/, diakses pada 18 Mei 2016

[2] https://crackstation.net/hashing-security.htm#salt, diakses pada 18 Mei 2016

[3] Munir, Rinaldi (2015). Diktat kuliah IF4020 : Kriptografi. Informatika ITB

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Mei 2015

Junita Sinambela
(13512023)