Rancangan dan Implementasi Algoritma Pembangkitan Kunci Kombinasi antara Algoritma RSA dan ElGamal

Ni Made Satvika Iswari - 13508077[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*if18077@students.if.itb.ac.id*

***Abstrak*—RSA adalah sebuah algoritma yang paling populer dalam kriptografi kunci publik. Keamanan algoritma RSA terletak pada sulitnya memfaktorkan bilangan yang besar menjadi faktor – faktor prima. Penemu algoritma RSA menyarankan nilai prima yang digunakan untuk membangkitkan kunci memiliki panjang lebih dari 100 digit untuk alasan keamanan. Algoritma ElGamal juga adalah salah satu algoritma kriptografi kunci publik. Keamanan algoritma ini terletak pada sulitnya menghiting logaritma diskrit. Pada makalah ini, penulis melakukan perbandingan antara algoritma RSA dan ElGamal. Dari hasil perbandingan tersebut didapatkan keunggulan dari masing – masing algoritma. Berdasarkan hasil tersebut, penulis kemudian merancang sebuah algoritma pembangkitan kunci yang dinilai cukup aman karena merupakan kombinasi antara algoritma RSA dan ElGamal. Selain itu, waktu komputasi yang diperlukan juga relatif cukup singkat.**

***Kata kunci*—RSA, ElGamal, perbandingan, rancangan, implementasi, analisis.**

I. PENDAHULUAN

RSA adalah sebuah algoritma yang paling populer dalam kriptografi kunci publik. Algoritma RSA dibuat oleh tiga orang peneliti dari MIT (*Massachussets Institute of Technology*) pada tahun 1976, yaitu Ron Rivest, Adi Shamir, dan Leonard Adleman.

Keamanan algorittma RSA terletak pada sulitnya memfaktorkan bilangan yang besar menjadi faktor – faktor prima. Pemfaktoran dilakukan untuk memperoleh kunci pribadi. Selama pemfaktoran bilangan besar menjadi faktor – faktor prima belum ditemukan algoritma yang mangkus, maka selama itu pula keamanan algoritma RSA tetap terjamin.

Penemu algoritma RSA menyarankan nilai prima yang digunakan untuk membangkitkan kunci memiliki panjang lebih dari 100 digit. Dengan demikian, hasil kali bilangan – bilangan prima tersebut berukuran lebih dari 200 digit. Menurut Rivest dan kawan – kawan, usaha untuk mencari faktor bilangan 200 digit membutuhkan waktu komputasi selama 4 milyar tahun.

Bilangan – bilangan prima yang memiliki panjang lebih dari 100 digit yang dianggap aman untuk algoritma RSA ini tentu akan memakan waktu komputasi yang relatif lebih lama apabila dibandingkan dengan menggunakan bilangan prima yang memiliki digit yang relatif lebih sedikit.

Algoritma ElGamal juga adalah salah satu algoritma kriptografi kunci publik. Algoritma ini pada mulanya digunakan untuk *digital signature*, namun kemudian dimodifikasi sehingga juga bisa digunakan untuk enkripsi dan dekripsi.

Keamanan algoritma ini terletak pada sulitnya menghitung logaritma diskrit. Masalah logaritma diskrit adalah jika p adalah bilangan prima dan g dan y adalah sembarang bilangan bulat, maka carilah x sedemikian sehingga

$$g^x = y \pmod p$$

Pada makalah ini, penulis akan melakukan perbandingan antara algoritma RSA dan ElGamal. Penulis akan menganalisa hasil dari masing – masing algoritma untuk menemukan kekurangan dan kelebihan dari masing – masing algoritma.

Setelah itu, penulis akan melakukan eksperimen, yaitu merancang sebuah pembangkitan kunci untuk algoritma kunci publik yang merupakan kombinasi antara Algoritma RSA dan Algoritma ElGamal.

Algoritma ini akan menggunakan bilangan prima yang memiliki jumlah digit kurang dari 100 digit, dengan harapan proses enkripsi dan dekripsi dapat memiliki waktu komputasi yang lebih sedikit dibandingkan dengan penggunaan bilangan prima lebih dari 100 digit pada Algoritma RSA.

Walaupun jumlah digit bilangan prima yang digunakan kurang dari 100 digit, namun faktor keamanan tetap diperhatikan dengan menggunakan kombinasi Algoritma RSA dan Algoritma ElGamal.

II. ALGORITMA RSA

Dari sekian banyak algoritma kriptografi kunci-publik yang pernah dibuat, algoritma yang paling populer adalah algoritma RSA.

Algoritma RSA dibuat oleh 3 orang peneliti dari MIT (*Massachussets Institute of Technology*) pada tahun 1976,

yaitu: Ron (R)ivest, Adi (S)hamir, dan Leonard (A)dleman.

Keamanan algoritma RSA terletak pada sulitnya memfaktorkan bilangan yang besar menjadi faktor-faktor prima. Pemfaktoran dilakukan untuk memperoleh kunci pribadi. Selama pemfaktoran bilangan besar menjadi faktor-faktor prima belum ditemukan algoritma yang mangkus, maka selama itu pula keamanan algoritma RSA tetap terjamin.

Besaran-besaran yang digunakan pada algoritma RSA:

1. p dan q bilangan prima (rahasia)
2. $r = p \cdot q$ (tidak rahasia)
3. $\phi(r) = (p-1)(q-1)$ (rahasia)
4. PK (kunci enkripsi) (tidak rahasia)
5. 5. SK (kunci dekripsi) (rahasia)
6. 6. X (plainteks) (rahasia)
7. 7. Y (cipherteks) (tidak rahasia)

A. Prosedur Membuat Pasangan Kunci

1. Pilih dua buah bilangan prima sembarang, p dan q.
2. Hitung $r = p \cdot q$. Sebaiknya $p \neq q$, sebab jika $p = q$ maka $r = p^2$ sehingga p dapat diperoleh dengan menarik akar pangkat dua dari r.
3. Hitung $\phi(r) = (p-1)(q-1)$.
4. Pilih kunci publik, PK, yang relatif prima terhadap $\phi(r)$.
5. Bangkitkan kunci rahasia dengan menggunakan persamaan (5), yaitu $SK \cdot PK \equiv 1 \pmod{\phi(r)}$.

Perhatikan bahwa $SK \cdot PK \equiv 1 \pmod{\phi(r)}$ ekivalen dengan $SK \cdot PK = 1 + m\phi(r)$, sehingga SK dapat dihitung dengan :

$$SK = \frac{1 + m\phi(r)}{PK} \qquad (11)$$

Akan terdapat bilangan bulat m yang menyebabkan memberikan bilangan bulat SK.

Catatan: PK dan SK dapat dipertukarkan urutan pembangkitannya. Jika langkah 4 diganti dengan "Pilih kunci rahasia, SK, yang …", maka pada langkah 5 kita menghitung kunci publik dengan rumus yang sama.

B. Enkripsi

- Plainteks disusun menjadi blok-blok $x_1, x_2, \ldots$, sedemikian sehingga setiap blok merepresentasikan nilai di dalam rentang 0 sampai $r - 1$.
- Setiap blok x_i dienkripsi menjadi blok y_i dengan rumus :

$$y_i = x_i^{PK} \bmod r$$

C. Dekripsi

- Setiap blok cipherteks y_i didekripsi kembali menjadi blok x_i dengan rumus :

$$x_i = y_i^{SK} \bmod r$$

III. ALGORITMA ELGAMAL

Algoritma Elgamal juga adalah algoritma kiptografi kunci-publik. Algoritma ini pada mulanya digunakan untuk *digital signature*, namun kemudian dimodifikasi sehingga juga bisa digunakan untuk enkripsi dan dekripsi. Kekuatan algoritma ini terletak pada sulitnya menghitung logaritma diskrit.

Besaran-besaran yang digunakan d dalam algoritma ElGamal adalah sebagai berikut :

1. Bilangan prima, p (tidak rahasia)
2. Bilangan acak, g ($g < p$)(tidak rahasia)
3. Bilangan acak, x ($x < p$) (rahasia)
4. M (plainteks) (rahasia)
5. a dan b (cipherteks) (tidak rahasia)

A. *Prosedur Membuat Pasangan Kunc*i

1. Pilih sembarang bilangan prima p.
2. Pilih dua buah bilangan acak, g dan x, dengan syarat $g < p$ dan $x < p$.
3. Hitung $y = g^x \bmod p$.
4. Kunci publik adalah y, kunci rahasia adalah x. Nilai g dan p tidak dirahasiakan dan dapat diumumkan kepada anggota kelompok.

B. Enkripsi

1. Plainteks disusun menjadi blok-blok $m_1, m_2, \ldots$, sedemikian sehingga setiap blok merepresentasikan nilai di dalam rentang 0 sampai $p - 1$.
2. Pilih bilangan acak k, yang dalam hal ini $0 \le k \le p - 1$, sedemikian sehingga k relatif prima dengan $p - 1$.
3. Setiap blok m dienkripsi dengan rumus

$$a = g^k \bmod p$$

$$b = y^k m \bmod p$$

Pasangan a dan b adalah cipherteks untuk blok pesan m. Jadi, ukuran cipherteks dua kali ukuran plainteksnya.

C. Dekripsi

Untuk mendekripsi a dan b digunakan kunci rahasia, x, dan plainteks m diperoleh kembali dengan persamaan

$$m = b/a^x \bmod p$$

Catatlah bahwa karena :

$$a^x \equiv g^{kx} \pmod p$$

maka

$$b/a^x \equiv y^k m/a^x \equiv g^{xk} m/g^{xk} \equiv m \pmod p$$

yang berarti bahwa plainteks dapat ditemukan kembali dari pasangan cipherteks a dan b.

IV. ANALISIS ALGORITMA RSA

Algortima RSA dijabarkan pada tahun 1977 oleh tiga orang : Ron Rivest, Adi Shamir dan Len Adleman dari Massachusetts Institute of Technology.

Huruf **RSA** itu sendiri berasal dari inisial nama mereka (**R**ivest—**S**hamir—**A**dleman).

Clifford Cocks, seorang matematikawan Inggris yang bekerja untuk GCHQ, menjabarkan tentang sistem equivalen pada dokumen internal di tahun 1973. Penemuan Clifford Cocks tidak terungkap hingga tahun 1997 karena alasan *top-secret classification*.

Algoritma tersebut dipatenkan oleh Massachusetts Institute of Technology pada tahun 1983 di Amerika Serikat sebagai U.S. Patent 4405829. Paten tersebut berlaku hingga 21 September 2000. Semenjak Algoritma RSA dipublikasikan sebagai aplikasi paten, regulasi di sebagian besar negara-negara lain tidak memungkinkan penggunaan paten. Hal ini menyebabkan hasil temuan Clifford Cocks di kenal secara umum, paten di Amerika Serikat tidak dapat mematenkannya.

Penyerangan yang paling umum pada RSA ialah pada penanganan masalah faktorisasi pada bilangan yang sangat besar. Apabila terdapat faktorisasi metode yang baru dan cepat telah dikembangkan, maka ada kemungkinan untuk membongkar RSA.

Pada tahun 2005, bilangan faktorisasi terbesar yang digunakan secara umum ialah sepanjang 663 bit, menggunakan metode distribusi mutakhir. Kunci RSA pada umumnya sepanjang 1024—2048 bit. Beberapa pakar meyakini bahwa kunci 1024-bit ada kemungkinan dipecahkan pada waktu dekat (hal ini masih dalam perdebatan), tetapi tidak ada seorangpun yang berpendapat kunci 2048-bit akan pecah pada masa depan yang terprediksi.

Semisal Eve, seorang *eavesdropper* (pencuri dengar—penguping), mendapatkan *public key N* dan *e*, dan ciphertext *c*. Bagimanapun juga, Eve tidak mampu untuk secara langsung memperoleh *d* yang dijaga kerahasiannya oleh Alice. Masalah untuk menemukan *n* seperti pada $n^e = c$ mod N di kenal sebagai permasalahan RSA.

Cara paling efektif yang ditempuh oleh Eve untuk memperoleh *n*dari *c* ialah dengan melakukan faktorisasi *N* kedalam *p* dan *q*, dengan tujuan untuk menghitung (*p*-1)(*q*-1) yang dapat menghasilkan *d* dari *e*. Tidak ada metode waktu polinomial untuk melakukan faktorisasi pada bilangan bulat berukuran besar di komputer saat ini, tapi hal tersebut pun masih belum terbukti.

Masih belum ada bukti pula bahwa melakukan faktorisasi *N* adalah satu-satunya cara untuk memperoleh *n* dari *c*, tetapi tidak ditemukan adanya metode yang lebih mudah (setidaknya dari sepengatahuan publik).

Bagaimanapun juga, secara umum dianggap bahwa Eve telah kalah jika *N* berukuran sangat besar.

Jika *N* sepanjang 256-bit atau lebih pendek, *N* akan dapat difaktorisasi dalam beberapa jam pada Personal Computer, dengan menggunakan perangkat lunak yang tersedia secara bebas. Jika *N* sepanjang 512-bit atau lebih pendek, *N* akan dapat difaktorisasi dalam hitungan ratusan jam seperti pada tahun1999. Secara teori, perangkat keras bernama TWIRL dan penjelasan dari Shamir dan Tromer pada tahun 2003 mengundang berbagai pertanyaan akan keamanan dari kunci 1024-bit. Santa disarankan bahwa *N* setidaknya sepanjang 2048-bit.

Pada thaun 1993, Peter Shor menerbitkan Algoritma Shor, menunjukkan bahwa sebuah komputer quantum secara prinsip dapat melakukan faktorisasi dalam waktu polinomial, mengurai RSA dan algoritma lainnya. Bagaimanapun juga, masih terdapat pedebatan dalam pembangunan komputer quantum secara prinsip.

A. Pembangkitan Kunci

Menemukan bilangan prima besar *p* dan *q* pada dasarnya didapatkan dengan mencoba serangkaian bilangan acak dengan ukuran yang tepat menggunakan probabilitas bilangan prima yang dapat dengan cepat menghapus hampir semua bilangan bukan prima.

p dan *q* seharusnya tidak "saling-berdekatan", agar faktorisasi fermat pada *N* berhasil. Selain itu pula, jika *p*-1 atau *q*-1 memiliki faktorisasi bilangan prima yang kecil, *N* dapat difaktorkan secara mudah dan nilai-nilai dari *p* atau *q* dapat diacuhkan.

Seseorang seharusnya tidak melakukan metoda pencarian bilangan prima yang hanya akan memberikan informasi penting tentang bilangan prima tersebut kepada penyerang. Biasanya, pembangkit bilangan acak yang baik akan memulai nilai bilangan yang digunakan. Harap diingat, bahwa kebutuhan disini ialah "acak" ***dan*** "tidak-terduga". Berikut ini mungkin tidak memenuhi kriteria, sebuah bilangan mungkin dapat dipilah dari proses acak (misal, tidak dari pola apapun), tetapi jika bilangan itu mudah untuk ditebak atau diduga (atau mirip dengan bilangan yang mudah ditebak), maka metode tersebut akan kehilangan kemampuan keamanannya. Misalnya, tabel bilangan acak yang diterbitkan oleh Rand Corp pada tahun 1950-an mungkin memang benar-benar teracak, tetapi dikarenakan diterbitkan secara umum, hal ini akan mempermudah para penyerang dalam mendapatkan bilangan tersebut. Jika penyerang dapat menebak separuh dari digit *p* atau *q*, para penyerang dapat dengan cepat menghitung separuh yang lainnya (ditunjukkan oleh Donald Coppersmith pada tahun 1997).

Sangatlah penting bahwa kunci rahasia *d* bernilai cukup besar, Wiener menunjukkan pada tahun 1990 bahwa jika *p* di antara *q*dan 2*q* (yang sangat mirip) dan *d* lebih kecil daripada $N^{1/4}/3$, maka *d* akan dapat dihitung secara effisien dari *N* dan *e*. Kunci enkripsi *e* = 2 sebaiknya tidak digunakan

B. Kecepatan

RSA memiliki kecepatan yang lebih lambat dibandingkan dengan DES dan algoritma simetrik lainnya. Pada prakteknya, Bob menyandikan pesan rahasia menggunakan algoritma simetrik, menyandikan kunci simetrik menggunakan RSA, dan mengirimkan kunci simetrik yang dienkripsi menggunakan RSA dan juga mengirimkan pesan yang dienkripasi secara simetrik kepada Alice.

Prosedur ini menambah permasalahan akan keamanan. Singkatnya, Sangatlah penting untuk menggunakan pembangkit bilangan acak yang kuat untuk kunci simetrik yang digunakan, karena Eve dapat

melakukan *bypass* terhadap RSA dengan menebak kunci simterik yang digunakan.

C. Distribusi Kunci

Sebagaimana halnya *cipher*, bagaimana *public key* RSA didistribusi menjadi hal penting dalam keamanan. Distribusi kunci harus aman dari *man-in-the-middle attack*. Anggap Eve dengan suatu cara mampu memberikan kunci yang bukan sebenarnya kepada Bob dan membuat Bob percaya bahwa kunci tersebut milik Alice. Anggap Eve dapat "menghadang" sepenuhnya transmisi antara Alice dan Bob. Eve mengirim Bob *public key* milik Eve, dimana Bob percaya bahwa *public key* tersebut milik Alice. Eve dapat menerima seluruh *ciphertext* yang dikirim oleh Bob, melakukan dekripsi dengan kunci rahasia milik Eve sendiri, menyimpan salinan dari pesan tersebut, melakukan enkripsi menggunakan *public key* milik Alice, dan mengirimkan *ciphertext* yang baru kepada Alice. Secara prinsip, baik Alice atau Bob tidak menyadari kehadiran Eve di antara transmisi mereka. Pengamanan terhadap serangan semacam ini yaitu menggunakan sertifikat digital atau komponen lain dari infrastuktur *public key*.

D. Penyerangan Waktu

Kocher menjelaskan sebuah serangan baru yang cerdas pada RSA di tahun 1995: jika penyerang, Eve, mengetahui perangkat keras yang dimiliki oleh Alice secara terperinci dan mampu untuk mengukur waktu yang dibutuhkan untuk melakukan dekripsi untuk beberapa *ciphertext*, Eve dapat menyimpulkan kunci dekripsi *d* secara cepat. Penyerangan ini dapat juga diaplikasikan pada skema "tanda tangan" RSA. Salah satu cara untuk mencegah penyerangan ini yaitu dengan memastikan bahwa operasi dekripsi menggunakan waktu yang konstan untuk setiap*ciphertext* yang diproses. Cara yang lainnya, yaitu dengan menggunakan properti multipikatif dari RSA. Sebagai ganti dari menghitung $c^d \bmod N$, Alice pertama-tama memilih nilai bilangan acak *r* dan menghitung $(r^e c)^d \bmod N$. Hasil dari penghitungan tersebut ialah $rm \bmod N$ kemudian efek dari *r* dapat dihilangkan dengan perkalian dengan inversenya. Nilai baru dari *r* dipilih pada tiap *ciphertext*. Dengan teknik ini, dikenal sebagai *message blinding* (pembutaan pesan), waktu yang diperlukan untuk proses dekripsi tidak lagi berhubungan dengan nilai dari *ciphertext*sehingga penyerangan waktu akan gagal.

V. ANALISIS ALGORITMA ELGAMAL

Algoritma Elgamal dibuat oleh Taher ElGamal pada tahun 1984. Algoritma ini pada umumnya digunakan untuk digital signature, namun kemudian dimodifikasi sehingga juga bisa digunakan untuk enkripsi dan deskripsi. ElGamal digunakan dalam perangkat lunak sekuriti yang dikembangkan oleh GNU, program PGP, dan pada sistem sekuriti lainnya.

Kekuatan algoritma ini terletak pada sulitnya menghitung logaritma diskrit. Masalah logaritma diskrit yang dimaksud adalah jika *p* adalah bilangan prima dan *g* dan *y* adalah sembarangan bilangan bulat, maka carilah *x* sedemikian sehingga :

$$g^x \equiv y \pmod{p}$$

Keamanan dari algoritma ElGamal bergantung pada properti dari group besaran – besaran yang digunakan serta skema *padding* yang digunakan untuk mengenkripsi pesan. Jika asumsi komputasi Diffie-Helman berlaku dalam kelompok besaran yang digunakan, maka fungsi enkripsi dilakukan secara satu arah.

Jika asumsi desisional Difiie-Helman berlaku dalam kelompok besaran yang digunakan, maka ElGamal mencapai keamanan semantik. Kemanan yang semantik tersebut tidak hanya diterapkan oleh asumsi komputasi Diffie-Helman saja, namun terdapat asumsi lain yang juga diterapkan.

Enkripsi ElGamal adalah *unconditionally malleable*, oleh karena hal tersebut, maka proses enkripsi tidak aman apabila dilakukan dibawah penyerangan *ciphertect* yang telah dipilih. Sebagai contoh, diberikan sebuah enkripsi (c_1,c_2) dari beberapa pesan (mungkin tidak diketahui) *m*, dengan hal tersebut maka seseorang dapat dengan mudah untuk membentuk enkripsi valid dari $(c_1, 2c_2)$ dari pesan *2m*.

Untuk mendapatkan keamanan dari *ciphertext* yang dipilih, maka skem yang dilakukan harus dimodifikasi, atau skema padding yang sesuai harus digunakan. Berdasarkan pada modifikasi tersebut, asumsi DDH bisa jadi dibutuhkan namun bisa juga tidak dibutuhkan.

Skema lain yang berhubungan dengan algoritma ElGamal yang mencapai ketahanan terhadap serangan *ciphertext* terpilih telah diusulkan. Cramer-Shoup cryptosystem adalah aman dibawah penyerangan *ciphertext* terpilih dengan mengasumsikan DDH yang diterapkan pada group besaran – besaran yang digunakan. Pembuktiannya tidak menggunakan *random oracle model*. Skema lain yang diusulkan adalah DHAES, dimana pembuktiannya membutuhkan asumsi yang lebih lemah daripada asumsi DDH.

Algoritma Elgamal tidak dipatenkan. Tetapi, algoritma ini didasarkan pada algoritma Diffie – Hellman, sehingga hak paten algoritma Diffie – Hellman juga mencakup algoritma ElGamal. Karena hak paten algoritma Diffie – Hellman berakhir pada bulan April 1997, maka algoritma ElGamal dapat diimplementasikan untuk aplikasi komersil.

1. Efisiensi

Algoritma ElGamal adalah probabilistik, yang berarti bahwa sebuah *plaintext* tunggal dapat dienkripsi menjadi beberapa *ciphertext* yang mungkin, yang konsekuensinya adalah proses enkripsi ElGama yang umum menghasilkan 2:1 perluasan pada ukuran dari *plaintext* ke *ciphertext*.

Proses enkripsi dengan menggunakan algoritma ElGamal membutuhkan dua buah perpangkatan; bagaimanapun, perpangkatan ini adalah tidak terikat pada pesan dan dapat dikomputasikan dengan cepat jika dibutuhkan. Proses dekripsi hanya membutuhkan sebuah perpangkatan.

VI. RANCANGAN ALGORITMA PEMBANGKITAN KUNCI

Berdasarkan hasil analisis yang telah dilakukan di atas, maka penulis mendapatkan beberapa kesimpulan mengenai kedua buah algoitma, yaitu algoritma RSA dan algoritma ElGamal. Kesimpulan tersebut adalah sebagai berikut :

- Keamanan algoritma *RSA* terletak pada tingkat kesulitan dalam memfaktorkan bilangan non prima menjadi faktor primanya, yang dalam hal ini $r = p \times q$.
- Sekali r berhasil difaktorkan menjadi p dan q, maka $\phi(r) = (p - 1)(q - 1)$ dapat dihitung. Selanjutnya, karena kunci enkrispi *PK* diumumkan (tidak rahasia), maka kunci dekripsi *SK* dapat dihitung dari persamaan $PK \cdot SK \equiv 1 \pmod{\phi(r)}$.
- Penemu algoritma *RSA* menyarankan nilai p dan q panjangnya lebih dari 100 digit. Dengan demikian hasil kali $r = p \times q$ akan berukuran lebih dari 200 digit. Menurut Rivest dan kawan-kawan, uasaha untuk mencari faktor bilangan 200 digit membutuhkan waktu komputasi selama 4 milyar tahun (dengan asumsi bahwa algoritma pemfaktoran yang digunakan adalah algoritma yang tercepat saat ini dan komputer yang dipakai mempunyai kecepatan 1 milidetik).
- Untunglah algoritma yang paling mangkus untuk memfaktorkan bilangan yang besar belum ditemukan. Inilah yang membuat algoritma *RSA* tetap dipakai hingga saat ini. Selagi belum ditemukan algoritma yang mangkus untuk memfaktorkan bilangan bulat menjadi faktor primanya, maka algoritma *RSA* tetap direkomendasikan untuk menyandikan pesan.
- Keamanan algoritma ElGamal terletak pada sulitnya menghitung logaritma diskrit.
- Masalah logaritma diskrit yang dimaksud adalah jika p adalah bilangan prima dan g dan y adalah sembarangan bilangan bulat, maka carilah x sedemikian sehingga :

$$g^x \equiv y \pmod{p}$$

Berdasarkan hasil analisis tersebut, pada bagian ini penulis akan membuat sebuah rancangan algoritma pembangkitan kunci untuk algoritma RSA dengan menggunakan gabungan algoritma pembangkitan kunci RSA dan ElGamal.

Besaran – besaran yang digunakan pada rancangan algoritma RSA adalah sebagai berikut :

1. p dan q bilangan prima (rahasia)
2. $r = p \cdot q$ (tidak rahasia)
3. $\phi(r) = (p - 1)(q - 1)$ (rahasia)
4. *PK* (kunci enkripsi) (tidak rahasia)
5. *SK* (kunci dekripsi) (rahasia)
6. *X* (plainteks) (rahasia)
7. *Y* (cipherteks) (tidak rahasia)

Sedangkan untuk algoritma yang penulis rancang menggunakan tambahan besaran yang merupakan besaran yang didapat dari algoritma ElGamal, yaitu sebagai berikut :

1. Bilangan prima, *pEl* (tidak rahasia)
2. Bilangan acak, g ($g < pEl$)(tidak rahasia)
3. Bilangan acak, x ($x < pEl$) (rahasia)
4. $y = g^x \bmod pEl$ (tidak rahasia, kc. publik)

Untuk bilangan prima, pEl, penulis mengenerate bilangan prima acak yang lebih besar daripada bilangan acak g dan bilangan acak x.

Bilangan acak g yang tidak dirahasiakan merupakan kunci enkripsi, PK yang didapatkan dari pembangkitan kunci untuk algoritma RSA. Sedangkan bilangan acak x yang dirahasiakan diapatkan dari kunci dekripsi, SK yang didapatkan dari pembangkitan kunci untuk algoritma RSA.

Tahapan – tahapan untuk melakukan pembangkitan kunci akan dijelaskan sebagai berikut :

Gambar 1 Tahapan Pembangkitan Kunci

Tahapan yang dilakuan di awal adalah sama dengan pembangkitan kunci dengan menggunakan algoritma RSA. Pertama – tama dilakuan pembangkitan dua buah bilangan prima, yaitu p dan q. Setelah itu yang dilakukan adalah menghitung r – p.q . Kemudian menghitung $\phi(r) = (p-1)(q-1)$. Setelah itu dilakukan pembangkitan bilangan acak, PK yang kemudian akan menjadi kunci publik (kunci enkripsi) dengan syarat PBB(PK, $\phi(r)$) = 1. Setelah didapatkan kunci publik, maka kita dapat menghitung kunci privat dengan rumus $SK \equiv PK^{-1} \bmod (\phi(r))$.

Setelah tahap pembangkitan kunci dengan algoritma RSA ini, dilanjutkan dengan pembangkitan kunci dengan menggunakan algoritma ElGamal. Pertama – tama kunci publik, PK akan menjadi bilangan acak g untuk algoritma ElGamal yang bersifat tidak rahasia. Kemudian, kunci private SK akan menjadi bilangan acak, x untuk algoritma ElGamal yang bersifat rahasia. Setelah itu, dilakukan pembangkitan bilangan prima pEl yang lebih besar daripada SK dan PK. Setelah itu, barulah kita dapat menghitung kunci publik dari algoritma ElGamal dengan menggunakan rumus :

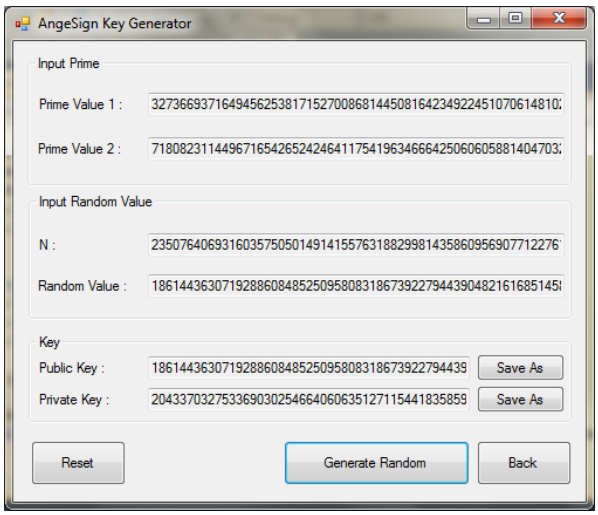
$$y = PK^{SK} \bmod pEL$$

Nilai kunci publik, y yang telah didapat ini barulah kemudian digunakan untuk menghitung kembali kunci private untuk algoritma RSA, dengan menggunakan rumus:

$$SK \equiv y^{-1} \bmod (\phi(r))$$

Namun, untuk perhitungan kunci private ini, kunci publik, y yang digunakan haruslah berdasarkan syarat memiliki PBB(y, $\phi(r)$) = 1. Oleh karena itu, jika kunci publik, y tidak memenuhi syarat ini, maka proses pembangkitan bilangan prima, pEl kembali dilakukan dan perhitungan kunci publik kembali dilakukan.

Berikut adalah tampilan program untuk pembangkitan kunci dengan menggunakan kombinasi algoritma RSA dan ElGamal tersebut. Sedangkan untuk source code dari program tersebut terlampir pada akhir dokumen ini.

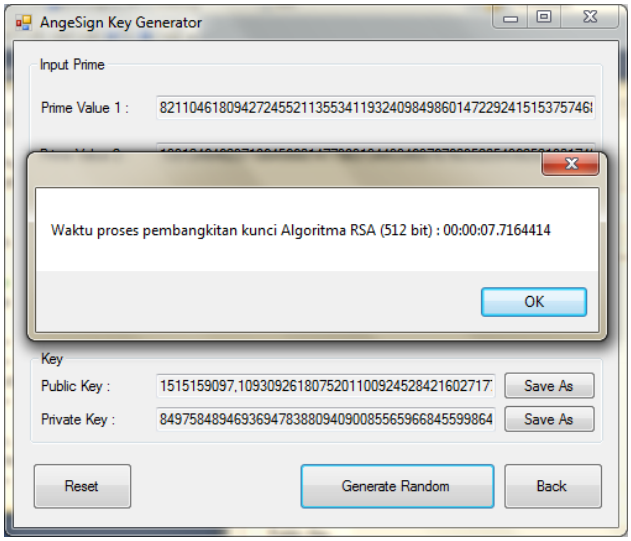


Gambar 2 Tampilan program pembangkitan kunci algoritma kombinasi RSA dan ElGamal

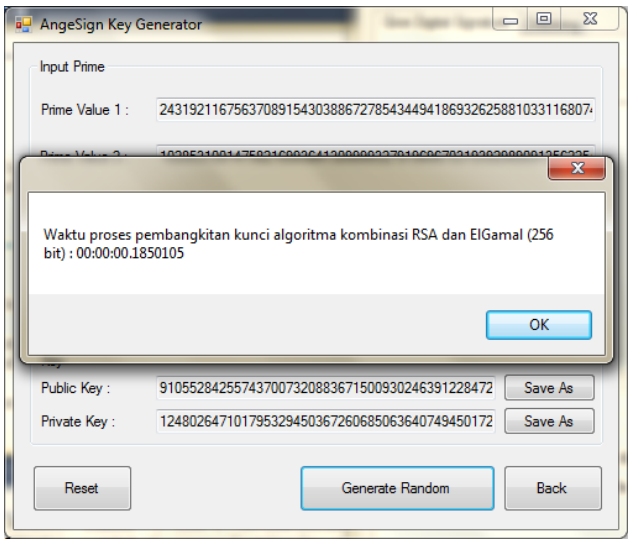
VII. EKSPERIMEN DAN ANALISIS

Pada bagian ini, penulis mencoba membandingan waktu proses antara pembangkitan kunci dengan algoritma RSA dengan panjang bilangan prima sebanyak 512 bit dibandingkan dengan pembangkitan kunci kombinasi antara algoritma RSA dengan ElGamal dengan panjang bilangan prima sebanyak 256 bit.

Berikut adalah tampilan antarmuka program yang telah dibuat.



Gambar 3 Proses pembangkitan kunci dengan algoritma RSA (panjang bilangan prima 512 bit)



Gambar 4 Proses pembangkitan kunci dengan algoritma kombinasi RSA dan ElGamal 256 bit

Secara umum, dapat disimpulkan bahwa RSA hanya aman jika n, yaitu perkalian antara 2 buah bilangan prima yang dibangkitkan cukup panjang. Jika panjang n hanyak 256 bit atau kurang, maka n dapat difaktorkan hanya dalam waktu beberapa jam saja dengan sebuah komputer PC dan program yang tersedia secara bebas.

Jika panjang n adalah 512 bit atau kurang, maka n dapat difaktorkan dengan beberapa ratus komputer. Pada eksperimen kali ini, penulis mencoba untuk menggunakan pembangkitan kunci dengan algoritma RSA dengan panjang bilangan prima 512 bit yang diasumsikan cukup aman walaupun dapat difaktorkan juga oleh beberapa ratus komputer.

Hasil yang didapatkan ternyata cukup siginifikan, untuk pembangkitan kunci dengan menggunakan algoritma RSA dengan bilangan prima sepanjang 512 bit membutuhkan waktu komputasi sebanyak 7.71 detik. Sedangkan pembangkitan kunci dengan menggunakan algoritma kombinasi RSA dan Elgamal dengan panjang bilangan prima 256 bit hanya membutuhkan waktu komputasi sebanyak 0.18 detik.

Waktu yang lebih sedikit dibutuhkan oleh algoritma kombinasi dikarenakan komputasi hanya menggunakan angka – angka yang relatif pendek, yaitu 256 bit jika dibandingkan dengan pembangkitan kunci algoritma RSA yang menggunakan bilangan prima sepanjang 512 bit.

Namun, algoritma kombinasi yang penulis rancang memiliki keunggulan karena walaupun jika n dapat difaktorkan hanya dalam waktu beberapa jam saja, namun sang pembobol tidak langsung dapat mengetahui kunci publik dan kunci privat karena harus menghitung logaritma diskrit yang merupakan kekuatan dari algoritma ElGamal.

VIII. KESIMPULAN

Dari penulisan makalah ini terdapat beberapa kesimpulan oleh penulis, yaitu sebagai berikut:

1. Algoritma RSA adalah algoritma kunci publik yang paling terkenal dan paling banyak aplikasinya.
2. Keamanan algoritma RSA terletak pada sulitnya memfaktorkan bilangan yang besar menjadi faktor – faktor prima.
3. Algoritma ElGamal adalah algoritma kunci publik yang pada awalnya digunakan untuk *digital signature*.
4. Keamanan algoritma ElGamal adalah terletak pada sulitnya menghitung logaritma diskrit
5. Rancangan pembangkitan kunci yang merupakan kombinasi antara algoritma RSA dan ElGamal memiliki keamanan yang ganda, yaitu pada sulitnya memfaktorkan bilangan yang besar menjadi faktor – faktor prima dan juga sulitnya menghitung logaritma diskrit.

REFERENSI

[1] Algoritma RSA
http://kur2003.if.itb.ac.id/file/Algoritma%20RSA.doc.
Waktu akses : 25 April 2011 pukul 21.00.

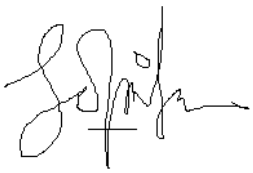
[2] Algoritma ElGamal
http://sisyboy.files.wordpress.com/2007/12/algoritma-elgama1.doc
Waktu akses : 25 April 2011 pukul 21.15.

[3] A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithm.
http://caislab.kaist.ac.kr/lecture/2010/spring/cs548/basic/B02.pdf
Waktu akses : 7 Mei 2011 puku 21.00

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, 9 Mei 2011



Ni Made Satvika Iswari (13508077)

Lampiran
Source Code

Program Pembangkitan Kunci Kombinasi Algoritma RSA dan ElGamal

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Org.BouncyCastle.Math;

namespace DigitalSignature
{
    class KeyGenerator
    {
        private BigInteger Prime1;      //Random prime 1
        private BigInteger Prime2;      //Random prime 2
        private BigInteger PublicKey;   //Public key
        private BigInteger PrivateKey;  //Private key
        private BigInteger N;             //N = Prime1 * Prime2

        //ElGamal
        private BigInteger PrimeEl;
        private BigInteger RandomEl;
        private BigInteger PrivateKeyEl;
        private BigInteger PublicKeyEl;

        //Constructor
        public KeyGenerator()
        {
            //Generate random prime value
            Random random = new Random();
            Prime1 = new BigInteger(256, random).NextProbablePrime();
            Prime2 = new BigInteger(256, random).NextProbablePrime();

            while (Prime1 == Prime2)
            {
                Prime2 = new BigInteger(256, random).NextProbablePrime();
            }

            PublicKey = getCoPrime(ToitentEuler());

            while (PublicKey.IntValue >= ToitentEuler().IntValue ||
PublicKey.IntValue <= 1)
            {
                //Generate random prime value
                Prime1 = new BigInteger(256, random).NextProbablePrime();
                Prime2 = new BigInteger(256, random).NextProbablePrime();

                while (Prime1 == Prime2)
                {
                    Prime2 = new BigInteger(256, random).NextProbablePrime();
                }

                PublicKey = getCoPrime(ToitentEuler());
            }

            N = Prime1.Multiply(Prime2);
            PrivateKey = (new BigInteger(PublicKey.ToString()).ModInverse(new
BigInteger(ToitentEuler().ToString())));

            //ElGamal
            PrimeEl = new BigInteger(256, random).NextProbablePrime();
            while (PublicKey.IntValue < PrimeEl.IntValue && PrivateKey.IntValue
< PrimeEl.IntValue)
            {
                PrimeEl = new BigInteger(256, random).NextProbablePrime();
            }

            RandomEl = PublicKey;
            PrivateKeyEl = PrivateKey;
            PublicKeyEl = RandomEl.ModPow(PrivateKeyEl, PrimeEl);
```

```
            System.Console.WriteLine("Random : " + RandomEl);
            System.Console.WriteLine("Public key : " + PublicKeyEl);

            //GCD
            BigInteger g = PublicKeyEl.Gcd(ToitentEuler());
            System.Console.WriteLine("GCD : " + g.CompareTo(new
BigInteger("1")));
            System.Console.WriteLine("Public key : " + PublicKeyEl);
            System.Console.WriteLine("Euler : " + ToitentEuler());

            while (g.CompareTo(new BigInteger("1")) != 0)
            {
                PrimeEl = new BigInteger(256, random).NextProbablePrime();
                while (PublicKey.IntValue < PrimeEl.IntValue &&
PrivateKey.IntValue < PrimeEl.IntValue)
                {
                    PrimeEl = new BigInteger(256, random).NextProbablePrime();
                }

                RandomEl = PublicKey;
                PrivateKeyEl = PrivateKey;
                PublicKeyEl = RandomEl.ModPow(PrivateKeyEl, PrimeEl);
                System.Console.WriteLine("Random : " + RandomEl);
                System.Console.WriteLine("Public key : " + PublicKeyEl);

                //GCD
                g = PublicKeyEl.Gcd(ToitentEuler());
                System.Console.WriteLine("GCD : " + g.CompareTo(new
BigInteger("1")));
            }

            PrivateKeyEl = (new
BigInteger(PublicKeyEl.ToString()).ModInverse(new
BigInteger(ToitentEuler().ToString())));
        }

        public BigInteger getPrime1()
        {
            return Prime1;
        }

        public BigInteger getPrime2()
        {
            return Prime2;
        }

        public BigInteger getPublicKey()
        {
            return PublicKeyEl;
        }

        public BigInteger getPrivateKey()
        {
            return PrivateKeyEl;
        }

        public BigInteger getN()
        {
            return N;
        }

        private BigInteger getCoPrime(BigInteger prime)
        {
            bool done = false;
            Random rand = new Random();
            BigInteger result = new BigInteger(256, rand);
```

```
            while (!done)
            {
                result = new BigInteger(rand.Next().ToString());

                // gcd test
                BigInteger g = result.Gcd(prime);
                if (g.CompareTo(new BigInteger("1")) == 0) done = true;
            }

            return result;
        }

        public BigInteger ToitentEuler()
        {
            return (Prime1.Add(new BigInteger("-1")).Multiply(Prime2.Add(new
BigInteger("-1"))));
        }
    }
}
```