STUDI MENGENAI *BCRYPT* DAN *EKSBLOWFISH* SEBAGAI ALGORITMA KRIPTOGRAFI YANG DAPAT BERADAPTASI DI MASA MENDATANG PADA OPENBSD

Nugroho Herucahyono – NIM : 13504038

Program Studi Teknik Informatika

Institut Teknologi Bandung

Jl. Ganesha 10 Bandung

Email : if14038@students.if.itb.ac.ic

Abstrak

Makalah ini akan berisi tentang skema otentikasi dengan password yang dapat beradaptasi dengan perkembangan *hardware*, yang diterapkan pada sistem operasi OpenBSD. Sebagaimana diketahui, perkembangan *hardware* yang semakin pesat, menjadikan komputer dapat melakukan berbagai operasi perhitungan dengan sangat cepat. Hal ini merupakan keuntungan sekaligus masalah dalam keamanan suatu sistem.

Sebagai contoh, sistem otentikasi pada UNIX yang menyimpan enkripsi password pada sebuah file. Pada masa dikembangkannya, dimana komputer dalam waktu 1 detik hanya bisa melakukan kurang dari 4 operasi *crypt* yang digunakan untuk mengenkripsi password, sistem ini relatif aman, karena *attacker* akan memerlukan waktu yang sangat lama untuk melakukan serangan secara *guessing*. Namun, dengan perkembangan *hardware*, kini kita dapat melakukan lebih dari 200.000 operasi *crypt* setiap detiknya, sehingga membuat *attacker* lebih berpeluang mendapatkan password.

OpenBSD merupakan sistem operasi yang hingga saat ini dinyatakan sebagai sistem operasi teraman. OpenBSD menggunakan prinsip *secure by default* dimana sistem operasi diusahakan seaman mungkin, walaupun dalam keadaan *default* tanpa konfigurasi dari user. Sebagai sistem operasi berbasis UNIX, OpenBSD juga memiliki masalah seperti sebagian besar sistem operasi UNIX, yaitu tentang enkripsi password yang mudah ditembus. Dengan berbagai *tools* yang ada saat ini, misalnya *John The Ripper, Lophtrack* dll, *attacker* dapat dengan mudah mendapatkan password dengan cara *guessing.* Menjawab tantangan ini, OpenBSD berupaya mengembangkan skema password baru yang dapat beradaptasi dengan perkembangan *hardware*.

Pada makalah ini kami membahas tentang skema password yang mampu beradaptasi dengan perkembangan hardware dan menunjukkan bahwa cost komputasi dari skema password tersebut harus dapat meningkat seiring dengan perkembangan kecepatan hardware. Kami membahas dua algoritma yang memiliki cost yang dapat diparameterisasi yang digunakan bersamaan dengan password. Eksblowfish, sebuah cipher block, yang memungkinkan kita menyimpan private key di dalam disk dengan aman. Bcrypt, sebuah fungsi hashing, dapat menggantikan fungsi hashing password Unix.

Kata Kunci : Bcrypt, Eksblowfish, OpenBSD, Perkembangan Hardware

1. Pendahuluan

Dengan semakin cepatnya prosesor, maka semakin cepat pula berbagai aplikasi yang berjalan di atasnya, termasuk pula software kriptografi. Semakin cepatnya kriptografi memungkinkan kita untuk membuat suatu sistem menjadi semakin aman. Hal ini memungkinkan kita membuat enkripsi yang dapat digunakan dalam berbagai jenis aplikasi.

Hal ini juga memungkinkan kita untuk meningkatkan keamanan suatu sistem dengan memperbesar parameter keamanan, misalnya panjang kunci pada suatu enkripsi. Dengan memperbesar parameter keamanan ini akan menjadikan kriptografi tersebut secara eksponensial menjadi jauh lebih aman. Ini akan menjadikan pertambahan kecepatan prosesor yang menguntungkan penyerang menjadi tidak berarti.

Peningkatan kecepatan prosesor tentu saja menawarkan keuntungan bagi penyerang. Dengan prosesor yang semakin cepat, penyerang dapat melakukan penyerangan terhadap sistem dengan lebih cepat dan efektif.

Misalnya saja, jika penyerang melakukan *brute force attack*, penyerang bisa membuat suatu program yang dapat meng-*generate* password untuk menebak. Hal ini merupakan masalah yang cukup serius pada pengamanan suatu sistem. Dengan hardware yang ada sekarang, dimungkinkan seorang penyerang mampu melakukan jutaan (bahkan mungkin ratusan juta) kali *password guessing*.

Suatu hal yang sangat disayangkan adalah dimana kecepatan hardware meningkat sangat pesat sehingga kemungkinan untuk membuat suatu sistem menjadi lebih aman terbuka lebar, namun parameter keamanan pada suatu sistem relatif tetap. Misalnya, panjang dan entropi *password* pilihan user selalu tetap dan tidak mengikuti perkembangan kecepatan *hardware*.

Walaupun hampir semua sistem yang ada meminta otentikasi user dengan menggunakan *password*, namun sedikit sekali sistem yang memperhitungkan algoritma mereka untuk menghadapi perkembangan kecepatan *hardware* dan tentu saja penyerang yang semakin *powerfull*.

Salah satu sistem yang menggunakan otentikasi dengan *password* yang banyak digunakan adalah Unix. Sistem *passowrd* merupakan contoh yang bagus bagi sistem keamanan yang gagal beradaptasi dengan perkembangan *hardware*.

Unix merupakan sistem operasi *multi-user*. Untuk dapat mengakses sistem operasi, user harus memasukkan *password* sebagai otentikasi. Seorang user biasanya melakukan login dengan memasukkan *username* dan *passoword* pada program login. Program ini kemudian mencocokkan *passoword* tersebut dengan *file password* yang dimiliki sistem.

Untuk menjaga kerahasiaan *passoword,* sistem operasi Unix tidak menyimpan *password* tersebut dalam bentuk plainteks. Sebagai gantinya, Unix menyimpan *hash* dari *password* tersebut menggunakan fungsi *hashing* satu arah, *crypt*, yang hanya dapat diketahui dengan cara menebak passwordnya. Untuk memverifikasi sebuah *passoword*, program login tersebut membandingkan antara hasil dari *crypt* terhadap password masukan user dengan *hash* password yang tersimpan dalam file.

Pada saat pertama kali digunakan, yaitu sekitar tahun 1976, fungsi hashing crypt hanya dapat memproses kurang dari 4 password setiap detiknya. Karena satu-satunya cara yang diketahui untuk dapat mengetahui password adalah dengan menebaknya, maka algoritma ini membuat password sangat sulit untuk diketahui penyerang. Bahkan, para desainer sistem operasi Unix merasa cukup aman dengan menempatkan file password tersebut terbuka dan dapat dibaca oleh semua user.

Sekarang, 30 tahun setelah algoritma crypt pertama kali digunakan, kecepatan komputer telah meningkat dengan sangat pesat. Sekarang, sebuah komputer yang cepat dapat melakukan lebih dari 200.000

(bahkan mungkin jutaan) operasi crypt setiap detiknya. Jika seorang penyerang mendapatkan file password tersebut, dia dapat membandingkan seluruh isi file tersebut dengan hasil dari crypt suatu dictionary password yang umum digunakan.

Walaupun demikian, crypt tetap banyak digunakan. Bahkan banyak perangkat lunak yang mengharuskan file passwordnya dapat dibaca oleh semua user.

Sekarang, skema otentikasi telah jauh lebih baik daripada file password Unix tersebut. Namun dalam praktiknya, skema otentikasi tersebut tetap bergantung pada user yang mengingat passwordnya.

Ada alternatif lain, misalnya dengan menggunakan hardware otentikasi spesial (misalnya smarcard dll) atau memberi user kode akses yang digenerate secara acak. Namun, pendekatan-pendekatan tersebut seringkali membuat user tidak nyaman dan membutuhkan biaya tambahan, sehingga skema otentikasi dengan password tetap banyak digunakan.

2. Tentang OpenBSD

OpenBSD merupakan sistem operasi yang mengutamakan keamanan sistem. Para pengembang OpenBSD sangat menekankan pengamanan yang kuat dalam mengembangkan sistem operasi ini. Mereka ingin menjadikan

OpenBSD sebagai sistem operasi nomor satu dalam hal keamanan (bahkan mungkin sudah demikian). OpenBSD dikembangkan dengan model *Open Software Development* sehingga celah-celah keamanan (sekecil apapun) dapat segera diketahui dan diperbaiki.

OpenBSD menggunakan pendekatan *Secure By Default*, yaitu, sistem operasi sudah diset dalam keadaan *secure* tanpa perlu melakukan konfigurasi lagi. Ini merupakan salah satu kelebihan OpenBSD dalam bidang keamanan dibandingkan sistem operasi lain, dimana user harus melakukan konfigurasi ulang jika menginginkan sistem yang benar-benar aman.

OpenBSD juga menerapkan *full disclosure* pada bug-bug dan masalah keamanan yang ditemuinya. Hal ini tentu sangat berbeda dengan vendor lain yang biasanya menyembunyikan bug dan masalah keamanan tersebut dari user. Pengembang OpenBSD juga sangat proaktif dalam proses auditing keamanan.

OpenBSD telah menggunakan berbagai algoritma kriptografi untuk menambah keamanan sistemnya. Kriptografi yang digunakan dalam OpenBSD dapat diklasifikasikan dalam beberapa aspek sebagai berikut:

- OpenSSH
  SSH merupakan protokol remote shell yang mengenkripsi data yang dikirim. OpenSSH merupakan versi gratis dan non patent dari SSH. Mulai digunakan pada OpenBSD mulai rilis 2.6.
- Pseudo Random Number Generator
  Melakukan generate stream angka yang tidak mungkin ditebak walaupun dengan mengetahui output sebelumnya, dan memiliki length cycle yang panjang, sehingga dapat dianggap tidak memiliki pola pengulangan.
- Cryptographic Hash Function
  Melakukan enkripsi satu arah dari stream data ke dalam string dengan panjang yang konstan. Salah satu algoritma yang akan dibahas dalam makalah ini, yaitu bcrypt merupakan bagian dari hash function ini.
- Cryptographic Transform
  Melakukan enkripsi dan dekripsi data. Salah satu algoritma yang akan dibahas dalam makalah ini, yaitu eksblowfish merupakan bagian dari cryptographic transform ini
- Cryptographic Hardware Support
  Hardware yang digunakan dalam kriptografi di OpenBSD, misalnya akselerator dan random number generator.

Sebagai salah satu sistem operasi berbasis Unix, OpenBSD juga memiliki masalah keamanan kriptografi sebagaimana umumnya sistem operasi Unix.

Namun untuk mengatasi hal itu, OpenBSD telah mengembangkan algoritma baru yang mampu beradaptasi dengan perkembangan hardware di masa yang akan datang. Diantaranya adalah bcrypt dan eksblowfish yang akan dibahas dalam makalah ini.

3. Teknik Serangan

Serangan terhadap sistem keamanan merupakan hal yang umum terjadi. Saat ini, dengan dukungan hardware, cukup mudah bagi para attacker untuk mendapatkan password dari suatu sistem operasi Unix dengan menggunakan guessing. Serangan password guessing dapat diketegorikan menurut interaksinya dengan sistem, yaitu online attacks dan offline attacks.

Dalam online attack, penyerang harus menggunakan sistem otentikasi untuk mengecek setiap password yang ditebak. Dalam offline attacks, penyerang mendapatkan informasi, misalnya hash dari password, yang memungkinkan penyerang melakukan pengecekan setiap tebakan password tanpa akses dengan sistem lebih lanjut.

Online attacks biasanya jauh lebih lambat dari offline attacks. Sistem dapat mendetek penyerangan dengan mudah. Karena lambatnya pengecekan password oleh penyerang, serangan ini tidak terlalu berbahaya. Sistem dapat dengan mudah memutuskan hubungan dengan penyerang, misalnya dengan memblok penyerang untuk tidak dapat lagi mengakses sistem.

Akan tetapi, ketika penyerang telah mendapatkan informasi tentang sistem verifikasi password, penyerang dapat melakukan offline attacks. Sekarang, satu-satunya kekuatan proteksi password terletak pada computational cost dari pengecekan password.

Teknik yang biasa digunakan dalam offline attacks adalah dictionary attack dan bruteforce attack.

Dictionary attack adalah membandingkan hasil enkripsi password tebakan yang berasal dari password dictionary dengan enkripsi password yang tersimpan dalam file password. Bruteforce attack hanya membandingkan password secara acak. Sekarang, banyak sekali program-program password cracking yang beredar di internet dan dapat digunakan dengan mudah oleh siapa saja. Beberapa tools password cracking yang terkenal adalah

Online Attacks:

- Hydra (http://www.thc.org)
- Cowpatty (jwright@hashborg.com)
- Medusa (http://foofus.net)
- Mezcal (http://0x90.org)
- TFtp Brute
- THC PPTP
- VN Crack
-

Offline Attacks:

- John The Ripper (http://www.openwall.com/john)
- Loopthrack
- Rainbow Crack
- Hash Collision
- SIP Crack (http://www.remote-exploit.org)

Diantara tools offline yang paling banyak digunakan adalah John The Ripper. John The Ripper ini adalah password gussing tools dengan menggunakan teknik dictionary attacks.

Dengan menggunakan John The Ripper, kita bisa membandingkan password sistem dengan tebakan kita secara cepat. Satu-satunya perlindungan passoword sistem dari serangan ini adalah panjangnya kunci password tersebut, yang mengakibatkan proses checking menjadi sangat lama.

Teknik yang dapat dilakukan untuk mengurangi kemungkinan offline attacks adalah dengan menyembunyikan file

password atau dengan memperbesar computational cost dari serangan. Sebagai contoh, sistem Unix modern tidak lagi menyimpan file passwordnya readable bagi user, tapi menyimpannya dalam file password *shadow* yang tidak dapat dibaca oleh user.

Beberapa orang telah merancang password protocol yang aman yang memungkinkan user untuk melakukan otentikasi melalui insecure network tanpa mengingat public key.

Namun hal ini tidak akan berpengaruh jika penyerang mampu mengambil file password dan melakukan offline attacks. Karena secure password protocol menggunakan kriptografi kunci publik, maka pasti memiliki parameter panjang kunci yang dapat diatur.

Parameter ini akan mempersulit offline attacks hanya dengan mempengaruhi computational cost dari serangan tersebut. Memperbesar panjang kunci untuk menghindari offline attacks juga menimbulkan beberapa konsekuesi, diantaranya memperbesar panjang messages dan membebani server dengan perhitungan komputasi yang tidak perlu.

Algoritma hashing yang tidak kuat, tidak hanya membahayakan komputer itu sendiri, tetapi juga membahayakan komputer yang lain. Kebanyakan orang menggunakan password yang sama pada beberapa mesin. Bahkan beberapa jaringan menyimpan file password yang sama pada beberapa mesin untuk memudahkan administrasi.

Walaupun shadow password tidak membahayakan keamanan sistem, namun pada kenyataanya kelemahan paling besar dari keamanan password bukan file password yang readable bagi user.

Kelemahan paling besar adalah pemilihan algoritma hashing yang tidak dapat beradaptasi dengan kecepatan hardware dan software yang meningkat 50.000 kali lebih cepat. Makalah ini akan membahas tentang skema password yang dapat beradaptasi dengan peningkatan efisiensi yang demikian pesat.

Beberapa sistem operasi lain telah mencoba menggunakan algoritma hashing yang lebih baik.

Sistem operasi FreeBSD, misalnya, telah menggunakan algoritma hashing pengganti crypt dengan algoritma yang berbasis MD5 message digest. MD5 crypt lebih lama dikomputasi daripada algoritma crypt yang asli. Akan tetapi, MD5 masih memiliki cost yang tetap dan dengan demikian tidak dapat beradaptasi dengan hardware yang lebih cepat.

Sekarang, proteksi MD5 crypt terhadap offline attacks semakin menurun dengan semakin cepatnya hardware. Optimasi juga telah ditemukan untuk mempercepat operasi kalkulasi MD5.

Akhirnya, banyak sistem yang tidak lagi mempercayakan keamanan otentikasi hanya melalui password. Misalnya login program ssh yang terkenal memungkinkan user melakukan otentikasi menggunakan enkripsi RSA.

Server SSH harus memiliki kunci public RSA user, tetapi server tidak perlu menyimpan informasi untuk memverifikasi password user. Tetapi kelemahannya adalah, user harus menyimpan private key nya di suatu tempat, biasanya di disk dan dienkripsi dengan suatu password. Yang terburuk adalah SSH menggunakan 3-DES yang sederhana untuk mengenkripsi kunci privat.

Hal ini menyebabkan cost untuk menebak password sebanding dengan cost untuk menghitung crypt. Meskipun demikian, karena fleksibilitasnya, otentikasi RSA pada SSH secara umum merupakan pendekatan yang lebih baik daripada skema yang hanya tergantung pada password saja.

Sebagai contoh, tanpa memodifikasi protokolnya, SSH dapat dengan mudah menggunakan algoritma eksblowfish yang akan dibahas pada makalah ini.

4. Kriteria Desain Skema Password

Beberapa algoritma yang menggunakan password masukan user sebagai input harus dimodifikasi untuk mencegah password guessing. Ini berarti kunci publik atau output yang berlaku pada waktu yang lama harus diminimalisasi penggunaannya dalam password reconstruction. Beberapa kriteria desain dapat membantu untuk mencapai tujuan ini.

Secara ideal, kita menginginkan algoritma password sebagai fungsi password satu arah yang kuat. Yaitu, jika penyerang memiliki output dan input dari algoritma, maka ia harus sulit untuk mempelajari informasi atau bagian informasi password yang ia tebak.

Sayangnya, fungsi satu arah didefinisikan asymtotical karena panjang inputnya. Seorang penyerang memiliki kemungkinan yang sangat kecil untuk menebak password jika panjang inputnya cukup.

Akan tetapi, panjang password yang dapat diterima oleh user pasti terbatas, maka kita memerlukan kriteria yang berbeda untuk fungsi password.

Secara umum, kita menginginkan skema password itu "sebaik password pilihan user". Jika ada probabilitas distribusi password *D*, kita mendefinisikan kemungkinan password ditebak *R(D)* dari distribusi sebagai kemungkinan tertinggi *Pr(s)* dari sembarang password tunggal *s* di *D*: $R(D) = max_{s \epsilon D} Pr(s)$.

Sebuah fungsi dari password aman dari penyerang jika kemungkinan penyerang untuk mempelajari beberapa bagian informasi parsial tentang password adalah proporsional terhadap kemungkinan prediksi distribusi password.

Apa keuntungan penyerang untuk mempelajari bagian parsial tentang sebuah password? Kita mendefinisikan informasi parsial sebagai nilai dari predikat bit tunggal dari password.

Seorang penyerang selalu dapat menebak predikat tertentu dengan kemungkinan yang tinggi. Sebagai contohnya, predikat trivial *P(s) = 1* yang mengembalikan nilai 1 pada semua password. Jika sebuah fungsi dari password tersebut aman, outputnya harus tidak memberikan kemungkinan bagi penyerang untuk menebak bebarapa predikat lebih akurat daripada yang ia dapat lakukan tanpa mengetahui output dari fungsi password tersebut.

Secara formal, misalkan *F(s, t)* sebuah fungsi. Argumen *s* mewakili password user, yang akan diambil dari probabilitas distribusi *D*.

Argumen *t* mewakili input tambahan yang tidak rahasia yang diambil oleh fungsi *F*. Misalkan nilai *t* diambil dari probabilitas distribusi *T*. Kita memodelkan penyerang sebagai sirkuit boolean random, *A,* yang mencoba menebak predikat *P* sebuah password.

Cost dari serangan atau usaha yang dikeluarkan oleh penyerang adalah jumlah gerbang didalam sirkut, yang kita notasikan $|A|$. Kita menggunakan notasi $Pr[v_1 \leftarrow S_1, v_2 \leftarrow S_2, \ldots; B]$ untuk menotasikan kemungkinan statemen *B* setelah sebuah percobaan dimana variabel $v_1, v_2, \ldots$ di dapatkan dari probabilitas distribusi *S1,S2,.....* Sekarang kita dapat mendefinisikan apa artinya sebuah password dapat bertahan dari sebuah serangan. Kita dapat mengatakan bahwa fungsi *F(s,t)* adalah sebuah ϵ-*secure password function* jika memenuhi:

1. Menemukan informasi parsial tentang input *F* yang rahasia adalah sesulit menebak password. Untuk setiap

$$\begin{aligned}
&\forall D, \forall P, \forall A, \\
&\quad \Big| \Pr\big[t_1 \leftarrow T, \ldots, t_c \leftarrow T, s \leftarrow D, \\
&\qquad b \leftarrow A(t_1, F(s,t_1), \ldots, t_c, F(s,t_c)); \\
&\qquad b = P(s)\big] \\
&\quad - \Pr\big[t_1 \leftarrow T, \ldots, t_c \leftarrow T, s \leftarrow D, \\
&\qquad b \leftarrow A(t_1, F(s,t_1), \ldots, t_c, F(s,t_c)), \\
&\qquad s' \leftarrow D;\ b = P(s')\big] \Big| \\
&\quad < \frac{\epsilon}{2} \cdot |A| \cdot R(D)
\end{aligned}$$

distribusi password *D* dan predikat *P*, seorang penyerang *A* yang menebak *P* berdasarkan pada output dari *F* akan sama dengan ketika *F* dikomputasi dengan password yang tidak berkaitan :

2. Menemukan *preimage* kedua sama sulitnya dengan menebak password. Sebuah *preimage* kedua dari input *(s,t)* adalah sebuah password yang berbeda *s'≠s* yang menghasilkan hasil fungsi yang sama *F(s, t) = F(s', t)*. Disini kita memodelkan penyerang *A* sebagai sirkuit random dengan *multiple output bits:*

$$\forall D,\ \forall A,$$
$$\Pr\big[t \leftarrow T,\ s \leftarrow D,\ s' \leftarrow A(s,t);$$
$$s \neq s' \wedge F(s,t) = F(s',t)\big]$$
$$< \epsilon \cdot |A| \cdot R(D)$$

Kita harus mencatat bahwa definisi ini sama dengan intuisi kita tentang fungsi hashing password seperti crypt. Jika user memilih password yang dapat dipredikasi, mengetahui hash password memberikan keuntungan yang cukup besar.

Mereka dapat membandingkan hash dari password yang paling populer dengan dengan hash password yang ingin mereka pecahkan. Jika seseorang dapat menebak predikat yang berguna bahkan tanpa melihat pada hash password, sebagai contoh mengetahui bahwa karakter ketiga dari password adalah huruf kecil, maka tentu saja penyerang dapat menebak hal ini juga.

Jika tidak ada satu password pun yang didapat dari password populer dengan kemungkinan tinggi, seorang penyerang memerlukan usaha yang cukup tinggi (sebagaimana dihitung pada gerbang sirkuit) untuk menemukan informasi *non-trivial* tentang password.

Akhirnya, kita juga ingin mencegah seorang penyerang mendapatkan string yang lain yang memiliki nilai hash yang sama dengan password; string yang demikian dapat dianggap sama pada proses otentikasi.

Persyaratan untuk mendapatkan *preimages* yang lain yang menyebabkan *collisions* sulit untuk didapat, bahkan dengan mengetahui password aslinya. Hal ini juga mengharuskan *F* tidak mengabaikan satu bit pun dari input masukan user.

Definisi tersebut menyatakan bahwa fungsi password yang aman *F(s,t)* harus membuat kegunaan *non-trivial* dari argumen keduanya, *t.* Untuk melihat ini, misalnya bahwa bit pertama dari *F(s,0)* adalah sebuah predikat yang valid dari password.

Seorang penyerang dapat dengan mudah menebak predikat ini jika *F* mengabaikan argumen keduanya atau string 0 muncul di *T* dengan probabilitas yang tinggi. Sebuah fungsi password dengan satu input *F(s)* dapat di *invert* oleh sebuah sirkuit yang cukup besar untuk meng*encode* sebuah tabel *lookup* yang memetakan *F(s)*. Ukuran dari sirkuit tersebut hanya bergantung pada kemungkinan distribusi password, bukan pada kekhususan *F*.

Tabel *lookup* seperti itu dapat dihalangi dengan memberikan input kedua pada *F* yang disebut *salt.* Jika sebuah *salt* acak dipilih ketika user memilih password baru, dan jika *space* dari salt cukup besar untuk dapat mengabaikan kemunculan kembali, maka tabel *lookup* tidak akan memberikan keuntungan apapun bagi penyerang. Jika mungkin *space* dari *salt* cukup kecil, bit output dari *F* menjadi predikat yang berguna dari password.

Jika *salted password* dapat menghalangi tabel *lookup,* misalnya jika pada *salt* dan hash tertentu, seorang penyerang masih dapat melakukan brute force attack dengan mengisi *F(s, t)* dengan setiap kemungkinan password yang mungkin. Ini mengakibatkan bahwa keamanan yang dapat didapat seseorang adalah $\epsilon \approx 1 / |F|$ dimana $|F|$ adalah cost dari gerbang sirkuit mengimplementasikan *F*.

Keterbatasan penggunaan akan menyebabkan batasan ϵ yang lebih kecil, misalnya jika user hanya dapat menunggu satu detik untuk mengecek password, *F* hanya dapat menggunakan satu detik untuk mengevaluasi password tersebut.

Jumlah gerbang $|A|$ yang dapat dikumpulkan oleh seorang penyerang dalam suatu penyerangan selalu bertambah, seiring dengan perkembangan hardware. Untungnya, begitupula dengan kecepatan mesin yang harus menjalankan *F*. Ini berarti

bahwa password tidak harus di hash oleh sebuah fungsi *F* pada computational cost yang tetap, tetapi oleh sekumpulan fungsi dengan cost yang tinggi.

Daripada membuang fungsi seperti crypt dan MD5 crypt untuk membuat lagi fungsi dari awal dengan fungsi yang lebih mahal tetapi tidak *compatible*, sistem seharusnya memungkinkan cost dari software untuk bertambah seiring dengan parameter yang dapat diatur dari fungsi tersebut. Demikian € dapat terus menurun secepat perkembangan hardware dan toleransi user.

Sebagai kesimpulan, sebuah fungsi password yang baik akan membuat penyerang yang ingin mengambil informasi parsial dari password sesulit memecahkan password tersebut sendiri. Sebuah parameter yang konkrit, €, dapat menggambarkan kesulitan ini.

Untuk mendapatkan nilai € yang kecil, sebuah fungsi password harus menerima input kedua, *salt*, yang mencegah penyerang menggunakan keuntungan dari tabel *lookup*. Nilai terbaik € adalah berbanding terbalik dengan cost evaluasi dari fungsi password.

Hal ini menunjukkan batas terkecil nilai € didasarkan pada cost maksimum yang dapat ditoleransi pada evaluasi *F* pada penggunaan yang logis. Karena kecepatan hardware semakin meningkat, skema password yang baik harus memungkinkan cost dari evaluasi *F* semakin membesar, sehingga nilai € semakin mengecil dari waktu ke waktu.

Satu kriteria terakhir dari fungsi password yang baik adalah meminimalkan nilai €.|*F*|. Yang berarti harus membuat fungsi password seefisien mungkin untuk setting dimana ia akan dipakai.

Desainer crypt gagal melakukan ini. Mereka membuat crypt berbasis pada DES, sebuah algoritma yang tidak efisien untuk diimplementasikan pada software karena banyak sekali transposisi bit.

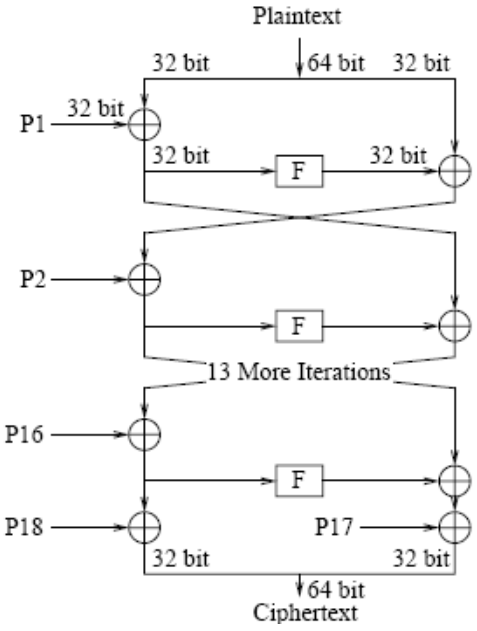
Mereka tidak memperhitungkan serangan hardware, hanya karena crypt tidak dapat dihitung menggunakan hardware DES yang ada. Sayangnya, kemudian ditemukan sebuah teknik software yang disebut *bitslicing* yang dapat mengeliminasi cost dari transposisi bit dalam menghitung banyak enkripsi DES secara simultan. Jika *bitslicing* tidak dapat membantu seseorang untuk login lebih cepat, namun memberikan kecepatan yang luar biasa untuk melancarkan serangan brute force.

Secara umum, algoritma password, berapapun costnya, harus dijalankan pada efisiensi optimal. Algoritma ini harus memperhitungkan kecepatan CPU, misalnya dalam penambahan, bitwise XOR, shifts, dan akses memori pada cache level pertama.

5. Algoritma Eksblowfish

Sekarang, kita akan membahas tentang algoritma block cipher dengan cost yang diparameterisasi dan dengan *salt* yang disebut *eksblowfish*, yaitu *ekspensive key schedule blowfish***.**

Eksblowfish didesain untuk menerima password masukan user sebagai key dan mempertahankan serangan pada key tersebut. Sebagai dasarnya, digunakan blowfish cipher block yang telah dikembangkan dengan baik dan sudah dianalisa oleh banyak pakar.



Blowfish adalah sebuah cipher block 64-bit, yang terbentuk sebagai sebuah 16 putaran jaringan Feitsel. Blowfish ini menggunakan 18 32-bits subkeys, $P_1.....P_{18}$, yang diturunkan dari kunci enkripsi. Subkey ini disebut sebagai *P-Array*.

Blowfish mengenkripsi dengan membagi sebuah blok input 64-bit menjadi dua buah blok 32-bit, L_o dan R_0.

Bagian yang paling signifikan, L_o, di XOR kan dengan subkey P_0 , dan digunakan sebagai untuk fungsi F. Hasil dari fungsi tersebut di XOR kan dengan bagian yang paling tidak signifikan, R_0. Kedua bagian tersebut kemudian ditukar, dan keseluruhan proses diulang 15 kali, dengan total 16 kali iterasi. Sehingga untuk $1 \leq i \leq 16$:

$$\begin{aligned} R_i &= L_{i-1} \oplus P_i, \\ L_i &= R_{i-1} \oplus F(R_i). \end{aligned}$$

Setelah 16 putaran, kedua bagian ditukar kembali, dan setiap bagian di XOR kan dengan subkey 32-bit yang lain:

$$\begin{aligned} R_{17} &= L_{16} \oplus P_{17}, \\ L_{17} &= R_{16} \oplus P_{18}. \end{aligned}$$

Fungsi F didalam Blowfish menggunakan empat array, $S_1.....S_4$ yang diturunkan dari kunci enkripsi. Setiap array mengandung 256 32-bit words.

Array tersebut berperan sebagai kotak subtitusi atau S-boxes, menggantikan input 8-bit dengan output 32-bit. F membagi input 32-bit nya menjadi empat byte 8-bit, a, b, c, d, dengan a sebagai byte yang paling signifikan. Menggantikan setiap byte dengan isi dari sebuah S-box, dan mengkombinasikan hasilnya sebagai berikut:

$$F(a,b,c,d) = \big((S_1[a] \boxplus S_2[b]) \oplus S_3[c]\big) \boxplus S_4[d].$$

```
EksBlowfishSetup (cost, salt, key)
  state ← InitState ()
  state ← ExpandKey (state, salt, key)
  repeat (2^cost)
    state ← ExpandKey (state, 0, salt)
    state ← ExpandKey (state, 0, key)
  return state
```

Eksblowfish melakukan enkripsi yang mirip dengan Blowfish. Eksblowfish menggunakan algoritma yang disebut EksblofishSetup. EksblowfishSetup memiliki tiga parameter input : cost, salt dan kunci enkripsi. Algoritma ini me return sebuah himpunan subkey dan S-boxes, yang disebut sebagai key schedule.

Parameter cost mengatur kemahalan dari komputasi key schedule. Salt adalah sebuah nila 128-bit yang mengubah key schedule, sehingga key yang sama tidak selalu menghasilkan nilai yang sama. Yang terakhir, argumen key adalah sebuah kunci enkripsi rahasia, yang bisa merupakan password pilihan user hingga 56 byte.

EksblowfishSetup mulai dengan memanggil InitState, sebuah fungsi yang mengkopi digits dari bilangan π kedalam subkey, kemudian kedalam S-boxes.

ExpandKey(state, salt, key) mengubah P-Array dan S-boxes didasarkan pada nilai salt 128-bit dan panjang variabel key.

Pertama, fungsi ini meng XOR kan semua subkey dalam P-array dengan kunci enkripsi. 32-bit pertama dari kunci di XOR dengan P_1, selanjutnya 32 bit dengan P_2 dan seterusnya. Kunci dilihat sebagai siklus, ketika proses mencapai akhir dari kunci, maka kembali menggunakan bit dari awal untuk meng XOR kan subkey.

Berikutnya, ExpandKey mengenkripsi dengan blowfish 64 bit pertama argumen salt menggunakan key schedule yang ada. Chiperteks hasilnya akan menggantikan subkey P_1 dan P_2.

Cipherteks tersebut juga di XOR kan dengan 64 bit kedua dari salt. Dan hasilnya dienkripsi dengan key schedule yang baru. Output dari enkripsi kedua menggantikan subkey P_3 dan P_4. Output ini juga di XOR kan dengan 64 bit pertama salt dan dienkripsi kemudian menggantikan P_5 dan P_6.

Proses tersebut berlangsung terus, berganti-ganti menggunakan 64 bit pertama dan 64 bit kedua dari salt. Ketika ExpandKey selesai menggantikan semua masukan dalam P-Array, maka ExpandKey melanjutkan dengan mengganti dua

masukan terakhir dari S-box terakhir. $S_4[254]$ dan $S_4[255]$, ExpandKey me return key schedule yang baru.

Didalam menjalankan ExpandKey(state, 0 key), sebuah blok dari 128 0-bit digunakan sebagai pengganti salt. Hal ini sama dengan dengan sebuah iterasi tunggal key schedule blowfish standar. Pemanggilan ExpandKey(state, 0, salt) akan menganggap salt sebagai key 16 byte.

Setelah memanggil *InitState* untuk mengisi key schedule dengan digit dari π, EksblowfishSetup memanggil *ExpandKey* dengan salt dan key. Hal ini memastikan bahwa semua state berikutnya tergantung pada kedua parameter tersebut, dan tidak ada bagian dari algoritma dapat dijalankan tanpa salt dan key tersebut.

Kemudian, ExpandKey dipanggil secara bergantian dengan salt dan key selama 2^{cost} iterasi. Untuk semua penggunaan ExpandKey, kecuali yang pertama, argumen kedua adalah sebuah blok 0bit sebanya 128.

Hal ini lebih mendekati key schedule blowfish asli, dan juga memungkinkan EksBlowfishSetup diimplementasikan lebih efisien pada arsitektur CPU dengan register yang sedikit.

6. Algoritma Bcrypt

Masalah yang ada pada skema password hashing Unix tradisional memunculkan sebuah skema password baru yang disebut bcrypt. Bcrypt menggunakan salt 128-bit dan mengenkripsi sebuah 1920-bit *magic value*. Algoritma ini menggunakan keuntungan dari expensive key setup dari eksblowfish.

Algoritma Bcrypt dijalankan dalam dua fase. Fase pertama, EksBlowfishSetup dipanggil dengan cost, salt dan password untuk menginisialisasi state eksblowfish. Sebagian besar waktu bcrypt dihabiskan dalam expensive key schedule.

Kemudian, nilai 192-bit "OrpheanBeholderScryDoubt" dienkripsi 64 kali menggunakan eksblowfish dalam mode ECB dengan state dari fase sebelumnya. Outputnya adalah cost dan salt 128-bit yang digabung dengan hasil dari loop enkripsi.

```
bcrypt (cost, salt, pwd)
  state ← EksBlowfishSetup (cost, salt, key)
  ctext ← "OrpheanBeholderScryDoubt"
  repeat (64)
    ctext ← EncryptECB (state, ctext)
  return Concatenate (cost, salt, ctext)
```

7. Implementasi Bcrypt

Para desainer OpenBSD telah mengimplementasikan bcrypt sebagai bagian dari sistem operasi OpenBSD. Bcrypt telah menjadi skema password default sejak OpenBSD 2.1.

Sebuah requirement yang penting dari implementasi bcrypt adalah bahwa bcrypt menggunakan space salt 128-bit secara penuh. OpenBSD meng generate salt 128-bit dari sebuah arcfour key stream, yang ditanam dengan data random yang dikumpulkan kernel dari device timing.

OpenBSD memungkinkan administrator memilih skema hashing password melalui file konfigurasi spesial, passwd.conf. passwd.conf ini memungkinkan kontro secara detail tipe password mana yang akan digunakan pada user atau grup tertentu. Konfiguarasi juga memungkinkan penggunaan skema password yang berbeda untuk lokal dan remote password.

Untuk bcrypt, administrator juga dapat menspesifikasikan costnya. Hal ini memungkinkan seseorang mengatur waktu verifikasi password disesuaikan dengan perkembangan kecepatan prosesor.

Pada saat ini, nilai default untuk cost adalah 6 untuk user biasa dan 8 untuk superuser. Tentu saja, nilai berapapun yang dipilih, harus dievaluasi dari waktu ke waktu.

Untuk membedakan password yang di hash menggunakan algoritma yang berbeda, setiap fungsi password selain crypt yang asli menuliskan identifier versi pada outputnya. Pada implementasi OpenBSD yang ada saat ini, bcrypt password dimulai

dengan “$2a$”, sedangkan MD5 crypt dimulai denga “1”. Karena hasil dari crypt tradisional tidak pernah dimulai dengan $, maka tidak mungkin terjadi ambiguitas.

8. Evaluasi Algortima Bcrypt

Karena Bcrypt memiliki nilai cost yang dapat disesuaikan, kita tidak dapat hanya mengevaluasi performansi dari algoritmanya saja.

Karena kecepatan algoritma juga ditentukan oleh nilai cost yang dapat dipilih. Sebagai gantinya, disini akan dibahas bermacam serangan dan optimasi yang dapat diterapkan pada fungsi hashing popular lainnya dan mendiskusikan penerapannya pada bcrypt.

a. Perbandingan

Berikut ini, diberikan gambaran singkat mengenai dua fungsi hashing yang banyak digunakan saat ini dan perbedaanya dengan bcrypt.

- Crypt Tradisional
  Desain Crypt Tradisional dirasionalkan dengan waktu saat pembuatannya, yaitu 1976. Algoritma ini menggunakan password hingga delapan karakter sebagai kunci bagi DES.

  Kunci DES 56-bit dibentuk dengan mengkombinasikan urutan yang lebih rendah dari setiap karakter dalam password. Jika password kurang dari 8 karakter, maka diisi dengan padding bit di kanan.

  Sebuah salt 12 bit digunakan untuk mengubah algoritma DES, sehingga password yang sama dapat menghasilkan 4.096 kemungkinan enkripsi.

  Sebuah modifikasi pada algoritma DES, mengganti bit i dan bit 24+i di output E-Box DES ketika bit i di set di dalam salt, hal ini menyebabkan hardware enkripsi DES tidak dapat digunakan untuk menebak password.

  Konstanta 64 bit “0” dienkripsi 25 kali dengan kunci DES. Output akhirnya adalah salt 12-bit yang digabungkan dengan nilai 64 bit yang dienkripsi. Hasil 76-bit dikodekan menjadi 13 karakter ASCII yang dapat ditulis.

  Pada saat itu, dapat dipahami bahwa crypt cukup cepat untuk ontentikasi tetapi terlalu berat untuk password guessing. Saat ini, kita waspada pada tiga keterbatasan yang dimiliki crypt, yaitu: panjang password yang terbatas, salt space yang kecil, dan cost eksekusi yang konstan.

  Sebaliknya, bcrypt memungkinkan password yang lebih panjang, salt yang cukup besar untuk dianggap unik, dan cost yang dapat beradaptasi. Maka keterbatasan crypt tersebut tidak ada pada bcrypt.

- MD5 Crypt
  MD5 Crypt ditulis oleh Poul-Henning Kamp untuk FreeBSD. Alasan utama menggunakan MD5 adalah untuk menghindari masalah dengan larangan ekspor produk kriptografi Amerika, dan agar memungkinkan password yang lebih panjang dari 8 karakter yang digunakan oleh crypt DES.

  Panjang password hanya dibatasi oleh ukuran maksimum message MD5, yaitu 2^{64} bit. Salt dapat bervariasi antara 12 – 48 bit.

  MD5 melakukan hashing pada password dan salt pada berbagai kombinasi berbeda untuk memperlambat waktu evaluasi. Beberapa langkah pada algoritma ini meragukan bahwa skema ini didesain dari sudut pandang kriptografi.

  Sebagai contohnya, representasi binari dari panjang password pada beberapa tempat menentukan data mana yang akan di hash, untuk setiap bit nol pada byte pertama password dan untuk setiap bit terakhir byte

pertama dari perhitungan hash sebelumnya.

Outputnya adalah gabungan version identifier “1”, salt, sebuah “$” separator dan output hash 128-bit.

MD5 memiliki panjang password tanpa batasan, sedangkan bcrypt memiliki password maksimum 55 byte. Ini bukan keterbatasan yang serius dari bcrypt, karena user tidak akan memilih password yang demikian panjang.

Ukuran output MD5 crypt yang hanya 128-bit merupakan keterbatasan keamanan. Seorang penyerang brute force dapat dengan lebih mudah menemukan hash string yang pendek yang memiliki nilai hash yang sama dengan password user daripada menemukan password aslinya. Kemudian, seperti DES crypt, MD5 crypt juga memiliki cost yang tetap.

b. Serangan dan Kelemahan

Sekali seorang penyerang mendapatkan file daftar hash password, password dapat ditebak dengan membandingkan list target dengan sebuah list dari candidate password yang di hash.

Hal ini didukung oleh kenyataan bahwa user lebih memilih password yang mudah ditebak. Metode yang paling umum disebut sebagai dictionary attack. Hal ini didasari pengetahuan bahwa kebanyakan user memilih password dengan cara yang sangat mudah ditebak.

Sering password seorang user dapat ditemukan dalam sebuah kamus, atau nama dari teman dekat dengan sedikit modifikasi. Penyerang mengumpulkan daftar nama dan kata yang umum digunakan sebagai password. Sebagai salt, kata dalam daftar di hash kan dengan skema password dan membandingkannya dengan masukan dari salt yang sama dari file password.

Jika ada yang sama, password plainteks telah ditemukan

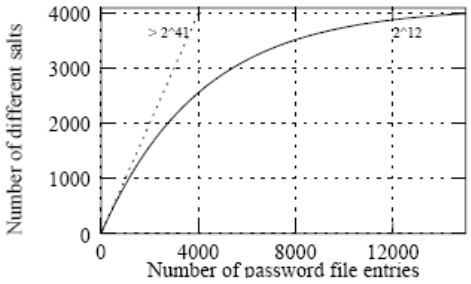
Secara umum, daftar password tersebut mengandung ratursan ribu kata. Dictionary attack dapat digunakan jika fungsi satu arah dapat dikomputasi dengan cepat. Cost dari Bcrypt dapat dibuat agar dapat ditoleransi oleh user, namun cukup lama untuk dilakukan dictionary attack.

.

n	10 digits	26 lowercase	36 lowercase alphanumeric	52 mixed case	62 mixed case alphanumeric	95 keyboard characters
4	0.04 sec	1.9 sec	7 sec	30.5 sec	61.6 sec	5.7 min
5	0.4 sec	49.5 sec	4.2 min	26.4 min	1.1 hours	9 hours
6	4.2 sec	21.5 min	2.5 hours	22.9 hours	2.7 days	35.5 days
7	41.6 sec	9.3 hours	3.8 days	49.6 days	169 days	9.2 years
8	6.9 min	10 days	136 days	7 years	28.8 years	875 years
9	1.2 hours	261 days	13.4 years	366 years	1786 years	83180 years

- **Salt Collisions**
  Salt Collisions terjadi ketika dua password dienkripsi dengan menggunakan salt yang sama. Secara ideal, tidak akan ada salt collisions. Salt dari enkripsi password seharusnya berbeda untuk setiap password yang ada di file password.

  Karena crypt tradisional hanya menggunakan 4096 salt yang berbeda, maka akan menyebabkan collisions yang banyak.



Untuk mengoptimasi dictionary attack, penyerang dapat mengelompokkan password menurut saltnya dan hanya melakukan hash pada setiap kandidat passwordnya hanya sekali untuk setiap salt. Pertambahan kecepatannya dapat dihitung menurut :

$$\frac{\text{number of passwords}}{\text{number of different salts}}.$$

Jika salt digenerate dengan generator angka yang baik, maka jumlah salt yang berbeda untuk n password dengan s salt yang mungkin adalah

$$EV(n,s) = \sum_{i=0}^{n-1} \left(\frac{s-1}{s}\right)^i = s - (s-1)^n s^{1-n}.$$

dalam sebuah file password dengan 15.000 masukan, space salt sebesar 2^{41} akan memastikan bahwa kemungkinan besar salt tersebut unik. Untuk 2^{12} salt yang mungkin, sebaliknya, kita hanya bisa mendapatkan 3.991 salt yang berbeda.

Pada 2^{24} salt yang mungkin, angka tersebut menjadi 14.994. Pada prakteknya, kita menemukan bahwa salt collision terjadi jauh lebih sering dari yang diduga. Alasannya adalah kebanyakan sistem operasi menggenarate angka random yang kurang baik (sering berulang).

- **Precomputing Dictionaries**

Menggunakan precomputation, seorang penyerangan dapat membuat daftar dari hash dengan setiap kemungkinan salt, dan menyimpan daftar tersebut pada data storage. Mengembalikan hash dari password yang umum kemudian cukup dengan *lookup* pada database, dengan sedikit cost komputasi.

Pada tahun 1934, Webster Dictionary mengandung (setelah dipotong menjadi 8 karakter, dan menghapus entri yang sama) 171.395 entri yang unik. Menggunakan crypt standar, hasil dari hash setiap kata dalam kamus dengan setiap kemungkian salt 12-bit yang mungkin dapat dimuat dalam satu harddisk 9GB.

Seseorang dapat melakukan yang lebih baik dengan menyimpan kurang dari seluruh output dalam database. Program cracking password QCrack menggunakan pendekatan ini.

QCrack melakukan prekomputasi pada database dari password yang umum digunakan, melakukan hash dengan setiap kemungkinan salt. QCrack tidak menyimpan 13 karakter output dari crypt, tetapi melakukan hash pada output crypt menjadi satu byte karakter.

Ketika melakukan cracking sebuah password dari kamus, QCrack menggunakan database untuk mewakili 255 dari setiap 256 kandidat password tanpa perlu melakukan komputasi terhadap hash.

Sebuah database QCrack dari Webster Dictionary hanya menggunakan 670 MB. QCrack dapat meyimpan sekitar 2.350.000 hash kata dalam harddisk 9GB.

Bcrypt memiliki space salt yang cukup besar, sehingga tidak mungkin menyimpan hasil hash dari bcrypt dalam storage.

9. Kesimpulan

Banyak skema otentikasi yang bergantung pada password rahasia. Sayangnya panjang dan entropi dari password tersebut tetap sepanjang waktu.

Sebaliknya, hardware terus berkembang semakin cepat, memberikan kekuatan komputasi bagi penyerang.

Hasilnya, skema password (termasuk sistem otentikasi pada Unix Tradisional) gagal bertahan menghadapi serangan offline password guessing.

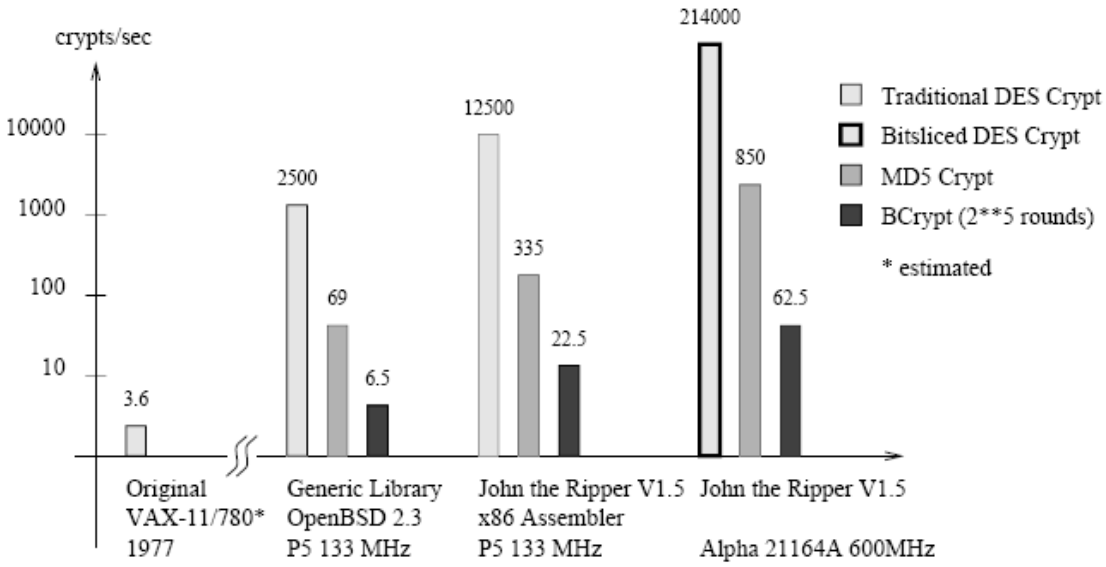


Figure 5: Impact of Algorithm Optimization and Advance in Processors

Pada makalah ini kami membahas tentang skema password yang mampu beradaptasi dengan perkembangan hardware dan menunjukkan bahwa cost komputasi dari skema password tersebut harus dapat meningkat seiring dengan perkembangan kecepatan hardware.

Kami membahas dua algoritma yang memiliki cost yang dapat diparameterisasi yang digunakan bersamaan dengan password. Eksblowfish, sebuah cipher block, yang memungkinkan kita menyimpan private key di dalam disk dengan aman. Bcrypt, sebuah fungsi hashing, dapat menggantikan fungsi hashing password Unix.

Algoritma – algoritma ini telah diimplementasikan dalam sistem operasi OpenBSD.

Algoritma ini telah dibandingkan dengan algoritma enkripsi lain yang banyak digunakan. Algoritma ini memungkinkan user melakukan penyesuaian cost dengan mengubah satu nilai konfigurasi sederhana pada file konfigurasi.

10. Referensi

[1]. Provos Niels and Mazieres David, “A Future-Adaptable Password Scheme” Usenix 1999.

[2]. De Raadt Theo, Hallqvist Niklas, Grabowski Artur, “Cryptography in OpenBSD” 1999.

[3]. D. Keromytis Angelos, L. Wright Jason, and De Raadt Theo, "The Design of the OpenBSD Cryptographic Framework." Usenix 2003.
[4]. http://www.openbsd.org/crypto.html
[5]. http://ezine.daemonnews.org
[6]. http://www.false.com/security/john
[7]. http://online.securityfocus.com/archives/1
[8]. http://citeseer.ist.psu.edu
[9]. http://www.usenix.org