STUDI MENGENAI ALGORITMA PANAMA DAN IMPLEMENTASINYA

Ray Aditya Iswara – NIM : 13504045

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl Ganesha 10, Bandung
E-mail : if14045@students.if.it.ac.id

Abstrak

Makalah ini akan membahas secara mendalam tentang algoritma Panama yang meliputi bagaimana desain dan implementasinya, serta serangan terhadap algoritma ini. Panama dapat berfungsi sebagai fungsi hash maupun *cipher* aliran. Fungsi hash atau lebih sering disebut fungsi hash kriptografi adalah fungsi yang menerima masukan *string* yang panjangnya sembarang dan mengkonversinya menjadi *string* keluaran yang panjangnya tetap (*fixed*). Fungsi-fungsi ini banyak sekali ditemukan pada aplikasi-aplikasi keamanan data, contohnya otentikasi pesan. *Cipher* aliran adalah algoritma kriptografi yang beroperasi pada plainteks/cipherteks dalam bentuk bit tunggal. *Cipher* aliran ini banyak diaplikasikan pada aliran data yang terus-menerus melalui saluran telekomunikasi, contohnya enkripsi suara pada jaringan GSM.

Panama merupakan modul kriptografi yang diciptakan oleh Joan Daemen dan Craig Clapp pada tahun 1998. Operasi pada Panama didasarkan pada *word* sepanjang 32-bit. Sebagai fungsi hash, Panama menerima masukan yang panjangnya sembarang dan menghasilkan *message digest* (pesan ringkas) sepanjang 256 bit. Sebagai *cipher* aliran, Panama mempunyai panjang kunci 256 bit. Algoritma ini didesain untuk digunakan pada arsitektur 32-bit. Panama dinyatakan memiliki tingkat keamanan yang tinggi dan kecepatan pemrosesan data yang cepat.

Kata kunci: Panama, *hash*, *cipher* aliran, *message digest*.

1. Pendahuluan

Masalah keamanan pada komputer adalah masalah yang sangat mudah ditemui dalam era teknologi informasi sekarang ini. Kejahatan *cyber* seperti pencurian data ataupun manipulasi data banyak terjadi di sekitar kita.

Kriptografi diciptakan sebagai salah satu solusi untuk mengatasi masalah keamanan komputer. Dalam perkembangannya, kriptografi sekarang ini sudah memasuki tahap kriptografi modern. Berbeda dengan algoritma kriptografi klasik, algoritma kriptografi modern dibuat sedemikian kompleks sehingga kriptanalisis sulit untuk dilakukan.

Di antara sekian banyak algoritma kriptografi modern, Panama muncul sebagai salah satu cara untuk menjaga kemanan data. Panama merupakan salah satu algoritma *cipher* aliran yang sekaligus dapat berfungsi sebagai fungsi *hash*. Panama ini diklaim mempunyai tingkat keamanan yang tinggi dalam menjaga data dan mempunyai kecepatan pemrosesan data yang cepat.

2. Sejarah Kriptografi

Kriptografi mempunyai sejarah yang cukup panjang. Ilmu ini sudah dikenal oleh bangsa Mesir 4000 tahun yang lalu berupa hyroglyph yang tidak standar. Di Yunani, kriptografi digunakan oleh tentara Sparta pada awal tahun 400 SM dengan alat yang diberi nama *Scytale*. Lalu pada abad ke-17, kriptografi menyebabkan Queen Mary, ratu Skotlandia, dipancung karena pesan rahasianya dari balik penjara berhasil dipecahkan. Lalu pada perang dunia II, pemerintah Nazi Jerman membuat mesin enkripsi yang diberi nama *Enigma*, dan berhasil dipecahkan oleh pihak Sekutu.

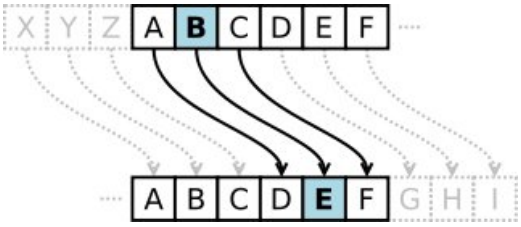
Sampai dengan awal abad ke-19, kriptografi masih termasuk dalam kriptografi klasik. Algoritma yang digunakan dalam enkripsi dan dekripsi tidak sedemikian kompleks sehingga dapat dengan mudah dipecahkan. Seiring dengan perkembangan zaman, terutama perkembangan teknologi informasi (komputer khususnya), kriptografi sekarang sudah memasuki era kriptografi modern. Kriptografi modern

menawarkan tingkat keamanan yang lebih dan tingkat kompleksitas yang lebih rumit.

3. Algoritma Kriptografi Klasik

Algoritma kriptografi klasik digunakan sebelum komputer ditemukan, biasanya hanya menggunakan kertas dan pena. Algoritma-algoritma kriptografi klasik termasuk ke dalam algoritma simetri dan digunakan jauh sebelum sistem kriptografi kunci publik ditemukan. Secara garis besar, algoritma kriptografi klasik terbagi menjadi dua bagian, yaitu: *cipher* substitusi dan *cipher* transposisi.

Cipher substitusi disebut juga sebagai *caesar cipher*, karena mula-mula digunakan oleh kaisar Romawi, Julius Caesar. Algoritma kriptografi ini mengganti setiap karakter denan karakter lain dalam susunan abjad.



Gambar 1. *Caesar cipher* mengganti setiap huruf dari plainteks dengan karakter lain dalam alfabet dengan pergeseran sebanyak tiga karakter

Jenis-jenis *cipher* substitusi:

1. Cipher abjad-tunggal
2. Cipher substitusi homofonik
3. Cipher abjad-majemuk
4. Cipher substitusi poligram

Cipher transposisi menggunakan metode *transpose* terhadap rangkaian karakter dalam teks. Nama lain dari cara ini adalah permutasi. Misalkan untuk plainteks : INFORMATIKA ITB, plaintes ditulis secara horizontal dengan lebar kolom tetap, misalnya selebar 3 karakter (kunci *k=3*):

INF
ORM
ATI
KAI
TB

Maka cipherteksnya dibaca secara vertikal menjadi:

IOAKTNRTABFMII

3.1. Cipher abjad-tunggal

Algoritma ini mengganti satu karakter pada plainteks dengan satu karakter lain yang bersesuaian. Fungsi *ciphering*-nya adalah fungsi satu-ke-satu. *Caesar cipher* merupakan kasus khusus dari cipher abjad tunggal di mana susunan huruf cipherteks diperoleh dengan menggesar huruf-huruf alfabet sebanyak x karakter.

3.2. Cipher substitusi homofonik

Seperti cipher abjad-tunggal, kecuali bahwa setiap karakter di dalam plainteks dapat dipetakan ke dalam salah satu dari karakter cipherteks yang mungkin. Misalnya huruf A dapat berkoresponden dengan 7, 9, atau 16. Huruf B dapat berkoresponden dengan 5, 10, atau 23, dan seterusnya. Fungsi cipheringnya memetakan satu-ke-banyak (one-to-many).

Cipher substitusi homofonik ini pertama kali digunakan pada tahun 1401 oleh wanita bangsawan Mantua.

Cipher ini lebih sulit dipecahkan daripasda cipher abjad-tunggal. Namun dengan known-plaintext attack, cipher ini dapat dipecahkan, sedangkan dengan ciphertext only attack lebih sulit.

3.3. Cipher abjad-majemuk

Merupakan cipher substitusi ganda yang melibatkan penggunaan kunci yang berbeda.

Cipher abjad-majemuk dibuat dari sejumlah cipher abjad-tunggal, masing-masing dengan kunci yang berbeda.

Cipher ini pertama kali ditemukan oleh Leon Battista pada tahun 1568. Metoe ini sempat digunakan oleh tentara AS selama Perang Sipil Amerika.

Meskipun cipher abjad-majemuk dapat dipecahkan dengan mudah (dengan bantuan komputer), namun anehnya banyak program keamanan komputer (computer security) yang menggunakan cipher jenis ini.

3.4. Cipher susbtitusi poligram

Blok karakter disubstitusi dengan blok cipherteks. Misalnya ABA diganti dengan RTQ, ABB diganti dengan SLL, dan lain-lain

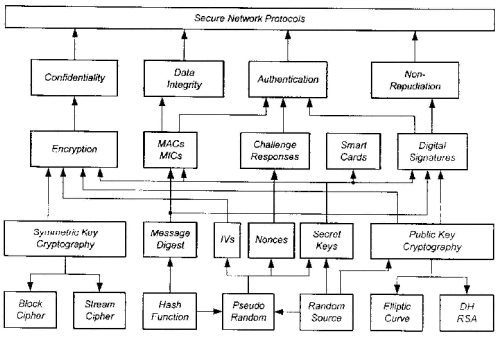
Playfair cipher, ditemukan pada tahun 1854, termasuk ke dalam cipher substitusi poligram dan digunakan oleh negara Inggris selama Perang Dunia I.

4. Algoritma Kriptografi Modern

Kriptografi modern menggunakan gagasan dasar yang sama seperti kriptografi klasik, tetapi penekanannya berbeda. Algoritma kriptografi modern dibuat sedemikian kompleks sehingga sulit untuk dipecahkan oleh orang lain.

Algoritma kriptografi modern pada umumnya beroperasi dalam mode bit, tidak seperti algoritma kriptografi klasik yang pada umumnya beroperasi pada mode karakter.

Pada perkembangannya, penggunaan mode berbasis bit didorong oleh penggunaan komputer digital yang merepresentasikan data dalam bentuk biner.



Gambar 2. Diagram blok kriptografi modern

4.1. Rangkaian Bit

Rangkaian bit yang dipecah menjadi blok-blok bit dapat ditulis dalam sejumlah cara dalam sejumlah blok. Misalkan untuk plainteks 100111010110 dibagi menjadi blok bit yang panjangnya 4 menjadi :

1001 1101 0110

Setiap blok menyatakan bilangan 0 sampai dengan 15, menjadi :

9 13 6

Bila panjang rangkaian bit tidak habis dibagi dengan ukuran blok yang ditentukan, maka blok yang terakhir ditambah dengan bit-bit semu yang disebut dengan *padding bits*. Cara ini dapat mengakibatkan ukuran plainteks hasil dekripsi lebih besar daripada plainteks ukuran semula. Selain cara-cara di atas, rangkaian bit dapat dinyatakan dalam notasi heksadesimal (HEX). Sehingga untuk rangkaian bit:

1001 1101 0110

dapat dinyatakan dalam notasi HEX:

9 D 6

4.2. Operator XOR

Operator XOR merupakan operator biner yang sering diguanakan dalam *cipher* yang beroperasi dalam mode bit. XOR merupakan kependekan dari *exclusive-or*. Notasi XOR biasa dilambangkan dengan $\oplus$.

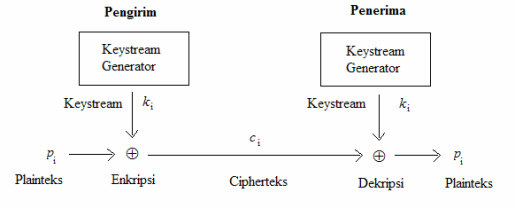
Operasi XOR dioperasikan pada dua bit dengan aturan sebagai berikut:
$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

5. *Cipher* Aliran

Cipher aliran mengenkripsi plainteks menjadi cipherteks bit per bit (1 bit setiap kali transformasi). Pertama kali diperkenalkan oleh Vernam melalui algoritmanya yang dikenal dengan nama Vernam Cipher.

Pada *cipher* aliran, bit hanya mempunyai dua buah nilai, sehingga proses enkripsi hanya menyebabkan dua keadaaan pada bit tersebut: berubah atau tidak berubah. Dua keadaan tersebut ditentukan oleh kunci enkripsi yang disebut aliran-bit-kunci (keystream).

Aliran-bit-kunci dibangkitkan dari sebuah pembangkit yang dinamakan pembangkit aliran-bit-kunci (keystream generator). Aliran-bit-kunci (sering dinamakan running key) di-XOR-kan dengan aliran bit-bit plainteks untuk menghasilkan aliran bit-bit cipherteks.



Gambar 3. Konsep *cipher* aliran

Keamanan sistem *cipher* aliran bergantung seluruhnya pada pembangkit aliran-bit-kunci. Jika pembangkit mengeluarkan aliran-bit-kunci yang seluruhnya nol, maka cipherteks sama dengan plainteks, dan proses enkripsi menjadi tidak ada artinya.

Jika pembangkit mengeluarkan aliran-bit-kunci dengan pola 16-bit yang berulang, maka algoritma enkripsinya menjadi sama seperti enkripsi dengan XOR sederhana yang memiliki tingkat keamanan yang tidak berarti.

Jika pembangkit mengeluarkan aliran-bit-kunci yang benar-benar acak (truly random), maka algoritma enkripsinya sama dengan *one-time pad* dengan tingkat keamanan yang sempurna. Pada kasus ini, aliran-bit-kunci sama panjangnya dengan panjang plainteks, dan kita mendapatkan *cipher* aliran sebagai *unbreakable cipher*.

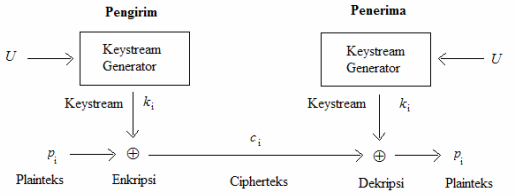
Tingkat keamanan *cipher* aliran terletak antara algoritma XOR sederhana dengan *one-time pad*. Semakin acak keluaran yang dihasilkan oleh pembangkit aliran-bit-kunci, semakin sulit kriptanalis memecahkan cipherteks.

5.1. Pembangkit aliran-bit-kunci (*Keystream* Generator)

Pembangkit aliran-bit-kunci dapat membangkitkan bit-bit kunci berbasis bit per bit dalam bentuk blok-blok bit.

Untuk alasan praktis, pembangkit aliran-bit-kunci diimplementasikan sebagai prosedur algoritmik, sehingga bit-bit kunci (*keystream*) dapat dibangkitkan secara simultan oleh pengirim dan penerima pesan.

Prosedur algoritmik tersebut menerima masukan sebuah kunci U. Pembangkit harus menghasilkan bit-bit kunci yang kuat secara kriptografi.



Gambar 4. *Cipher* aliran dengan pembangkit aliran-bit-kunci yang bergantung pada kunci U

Karena pengirim dan penerima harus menghasilkan bit-bit kunci yang sama, maka keduanya harus memiliki kunci *U* yang sama. Kunci *U* ini harus dijaga kerahasiaannya.

Cipher aliran menggunakan kunci *U* yang relatif pendek untuk membangkitkan bit-bit kunci yang panjang.

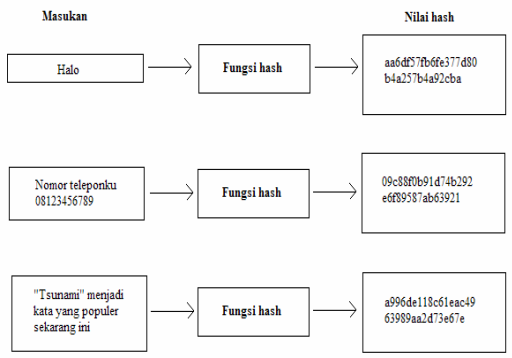
6. Fungsi Hash

Fungsi hash atau lebih sering disebut fungsi hash kriptografi adalah fungsi yang menerima masukan *string* yang panjangnya sembarang dan mengkonversinya menjadi *string* keluaran yang panjangnya tetap (*fixed*).

Fungsi hash dapt menerima masukan string apa saja. Jika string menyatakan pesan, maka sembarang pesan M berukuran bebas dikompresi oleh fungsi hash H melalui persamaan:

h = H(M)

Keluaran dari fungsi hash disebut juga nilai hash (hash-value) atau pesan-ringkas (message digest). h di atas adalah nilai hash atau message digest dari fungsi H untuk masukan M. Dengan kata lain, fungsi hash mengkompresi sembarang pesan yang berukuran berapa saja menjadi message digest yang ukurannya selalu tetap (dan lebih pendek dari panjang pesan semula). Nama lain dari fungsi hash adalah: fungsi kompresi/kontraksi, cetak-jari (fingerprint), cryptographic checksum, message integrity check (MIC), manipulation detection code (MDC).



Gambar 5. Contoh hashing beberapa buah pesan dengan panjang yang berbeda-beda

Aplikasi fungsi hash misalnya untuk memverifikasi kesamaan salinan suatu arsip dengan arsip aslinya yang tersimpan di dalam sebuah basis data terpusat. Daripada mengirim salinan arsip tersebut secara keseluruhan ke komputer pusat (yang membutuhkan waktu transmisi lama dan ongkos yang mahal), lebih mangkus mengirimkan message digest-nya saja. Jika messange digest salinan arsip sama dengan message digest arsip asli, berarti salinan arsip tersebut sama dengan arsip di dalam basis data.

6.1. Fungsi Hash Satu-Arah

Fungsi hash satu-arah adalah fungsi hash yang bekerja dalam satu arah: pesan yang sudah diubah menjadi message digest tidak dapat dikembalikan lagi menjadi pesan semula. Dua pesan yang berbeda akan selalu menghasilkan nilai hash yang berbeda pula. Sifat-sifat fungsi hash satu-arah adalah sebagai berikut:

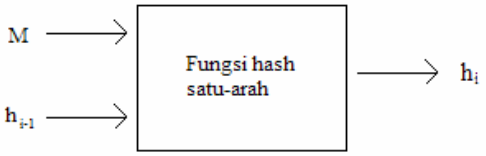
1. Fungsi H dapat diterapkan pada blok data berukuran berapa saja.
2. H menghasilkan nilai (h) dengan panjang tetap (fixed-length output).
3. H(x) dapat dihitung untuk setiap nilai x yang diberikan.
4. Untuk setiap h yang diberikan, tidak mungkin menemukan x sedemikianm sehingga H(x) = h. Itulah sebabnya fungsi H dikatakan fungsi hash satu-arah.
5. Untuk setiap x yang diberikan, tidak mungkin mencari y tidak sama dengan x sedemikian sehingga H(y) = H(x).
6. Tidak mungkin (secara komputasi) mencari pasangan x dan y sedemikian sehingga H(x) H(y).

Keenam sifat di atas penting sebab sebuah fungsi hash seharusnya berlaku seperti fungsi acak. Sebuah fungsi hash dianggap tidak aman jika (i) secara komputasi dimungkinkan menemukan pesan yang bersesuaian dengan pesan ringkasnya, dan (ii) terjadi kolisi (collision), yaitu terdapat beberapa pesan berbeda yang mempunyai pesan ringkas yang sama.

Fungsi hash bekerja secara iteratif. Masukan fungsi hash adalah blok pesan (M) dan keluaran dari hashing blok pesan sebelumnya,

$h_i = H(M_i, h_{i-1})$

Fungsi hash adalah publik (tidak dirahasiakan), dan keamanannya terletak pada sifat satu arahnya itu. Berikut ini adalah skema dari fungsi hash:



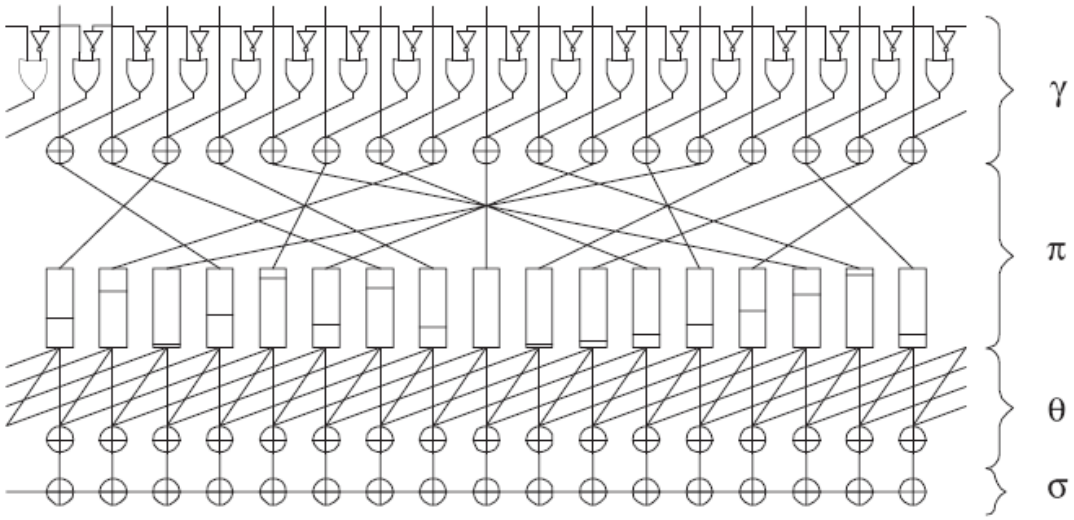
Gambar 6. Fungsi hash satu-arah

7. Algoritma Kriptografi Panama

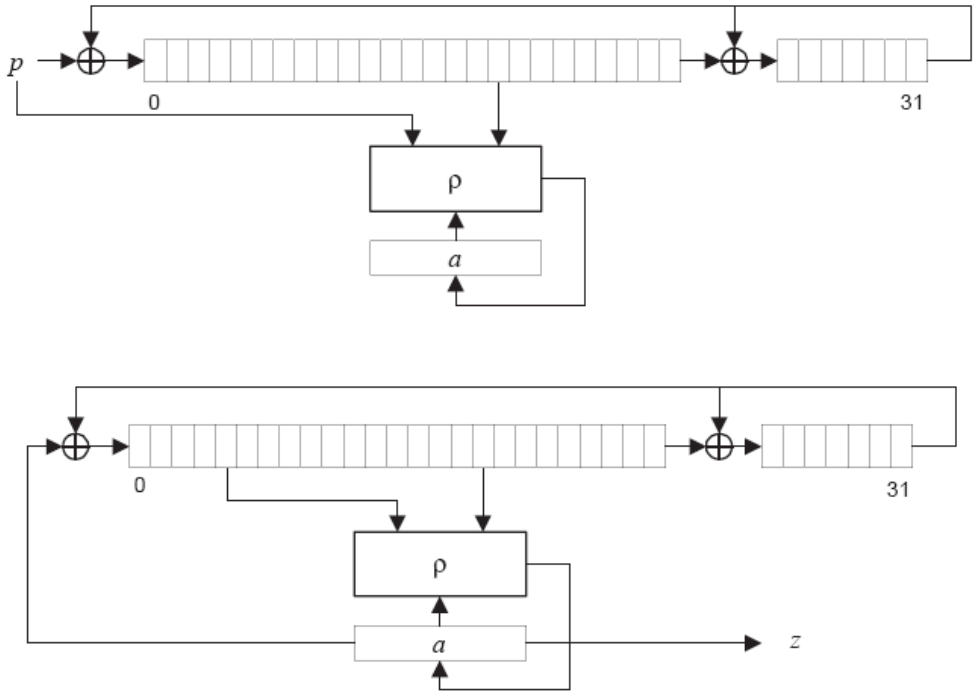
7.1. Desain

Desain dari Panama didasarkan pada mesin *state* berhingga dengan *state* 544-bit dan *buffer* atau penyangga sepanjang 8192 bit. *State* dan penyangga dapat diubah dengan melakukan sebuah iterasi. Ada dua macam mode pada fungsi iterasi. Yang pertama adalah mode *Push*, yang dapat memasukkan sebuah masukan dan tidak menghasilkan keluaran, dan yang kedua adalah mode *Pull*, yang tidak mengambil input dan menghasilkan keluaran. Sebuah iterasi Pull kosong adalah sebuah iterasi Pull yang keluarannya diabaikan.

Perubahan transformasi pada state mempunyai sifat difusi yang tinggi dan nonlinear yang terdistribusi. Desainnya ditujukan pada penyediaan kenon-linearan yang sangat tinggi dan difusi yang cepat untuk iterasi yang banyak. Hal ini disadari dari kombinasi dari empat transformasi masing-masing dengan kontribusi sendiri. Ada satu untuk kenon-linearan, satu untuk dispersi bit, satu untuk difusi inter-bit, dan



Gambar 7. Transformasi ρ



Gambar 8. Mode Push (atas) dan Mode Pull (bawah) pada Panama.

satu untuk pemasukan penyangga dan bit-bit input.

Penyangga bertindak sebagai LFSR (Linear Feedback Shift Register) yang memastikan bahwa bit masukan telah dimasukkan ke dalam state melalui iterasi dengan interval yang panjang. Dalam mode Push, input ke shift register dibentuk oleh masukan eksternal, dan pada mode Pull, dibentuk oleh bagian dari state.

Fungsi hash Panama didefinisikan sebagai melakukan iterasi-iterasi Push dengan input blok-blok pesan. Jika semua blok pesan telah dimasukkan, sejumlah iterasi Pull kosong dilakukan agar blok-blok pesan akhir terdifusi ke

dalam penyangga dan state. Hal ini diikuti oleh sebuah iterasi Pull akhir untuk mendapatkan hasil hashnya.

Skema enkripsi aliran dari Panama diinisialisasi dengan melakukan dua iterasi Push untuk memasukkan kunci dan parameter diversifikasi diikuti sejumlah iterasi Pull kosong agar kunci dan parameter terdifusi ke dalam penyangga dan state. Setelah inisialisasi ini, skema tersebut siap untuk menghasilkan bit-bit *keystream* dengan melakukan iterasi-iterasi Pull.

7.2. Spesifikasi

State dinotasikan dengan a dan terdiri dari 17 (32-bit) *word*, a_0 sampai dengan a_{16}. Penyangga b adalah sebuah Linear Feedback Shift Register dengan 32 tahapan, masing-masing mengandung 8 *word*. Sebuah tahapan 8-word dinotasikan dengan b^j dan word-wordnya b^j_i. Keduanya mempunyai indeks dimulai dari 0.

Tiga mode yang mungkin untuk Panama adalah Reset, Push, dan Pull. Dalam mode Reset, state dan penyangga diset menjadi 0. Dalam mode Push, sebuah masukan 8-word diaplikasikan dan tidak ada keluaran. Dalam mode Pull, tidak ada masukan dan dihasilkan sebuah keluaran 8-word.

Operasi update penyangga dinotasikan dengan λ. Kita mempunyai (dengan $d = \lambda(b)$):

$$\begin{aligned} d^j &= b^{j-1} \text{ if } j \notin \{0, 25\}, \\ d^0 &= b^{31} \oplus q \ , \\ d^{25}_i &= b^{24}_i \oplus b^{31}_{i+2 \bmod 8} \text{ for } 0 \le i < 8 \ . \end{aligned}$$

Dalam mode Push, q merupakan blok input p, dalam Pull mode, hal ini merupakan bagian dari state a, dengan 8 word komponen:

$$q_i = a_{i+1} \text{ for } 0 \le i < 8 \ .$$

Transformasi update state dinotasikan dengan ρ. Hal tersebut terdiri dari sejumlah transformasi spesifik:

$$\rho = \sigma \circ \theta \circ \pi \circ \gamma \ .$$

Disini ◦ menotasikan komposisi asosiatif dari transformasi di mana transformasi paling kanan dilakukan terlebih dahulu.

θ adalah transformasi linear yang dapat dibalik dan didefinisikan sebagai:

$$c = \theta(a) \Leftrightarrow c_i = a_i \oplus a_{i+1} \oplus a_{i+4} \text{ for } 0 \le i < 17 \ ,$$

dengan indeks modulo 17. Sifat dapat dibalik dari θ dapat ditelusuri dari fakta bahwa $1 \oplus x \oplus x^4$ adalah koprima terhadap $1 \oplus x^{17}$.

γ adalah sebuah transformasi non linear yang dapat dibalik, didefinisikan sebagai:

$$c = \theta(a) \Leftrightarrow c_i = a_i \oplus a_{i+1} \oplus a_{i+4} \text{ for } 0 \le i < 17 \ ,$$

dengan indeks modulo 17.

Permutasi π mengkombinasikan pergeseran word putar dan sebuah permutasi dari posisi-posisi word. Jika kita mendefinisikan τ_k sebagai rotasi terhadap posisi-posisi k dari LSB ke MSB, kita mendapatkan:

$$c = \pi(a) \Leftrightarrow c_i = \tau_k(a_j) \ ,$$

dengan

$$\begin{aligned} j &= 7i \bmod 17 \quad \text{and} \\ k &= i(i+1)/2 \bmod 32 \ . \end{aligned}$$

Transformasi σ korespon dengan penambahan bitwise dari penyangga dan word masukan. Hal ini didefinisikan dengan (misal c = σ(a)) :

$$\begin{aligned} c_0 &= a_0 \oplus 00000001_{\text{hex}} \ , \\ c_{i+1} &= a_{i+1} \oplus \ell_i \quad \text{untuk } 0 < i < 8 \ , \\ c_{i+9} &= a_{i+9} \oplus b^{16}_i \quad \text{untuk } 0 \le \imath < 8. \end{aligned}$$

Dalam mode Push l korespon dengan input p, di dalam mode Pull $l = b^4$.

Dalam mode Pull, keluaran z terdiri dari 8 word yang didefinisikan sebagai:

$z_i = a_{i+9}$ untuk $0 \le i < 8$.

7.2.1. Fungsi Hash Panama

Fungsi hash Panama memetakan sebuah pesan dengan panjang M menjadi sebuah keluaran 256 bit. Fungsi hash Panama dieksekusi dalam dua tahap:

- Padding (penambahan bit pengganjal) pada M menjadi string M' dengan panjang yang merupakan kelipatan dari 256. Proses ini dilakukan denan menambahkan sebuah bit 1 dan diikuti sejumlah d bit 0 dengan $0 \le d < 256$.

- Iterasi. Sekuen masukan M' = $p^1p^2 ... p^V$ dimasukkan ke dalam modul Panama menurut Tabel 1.

Setelah semua blok masukan telah di-*load*, sebanyak 32 Pull kosong dilakukan. Lalu, dihasilkan hasil hash *h*. Banyaknya Push dan Pull untuk *hash* blok input V adalah V +33.

Time step t	Mode	Input	Output
0	reset	–	–
$1, \ldots, V$	Push	p^t	–
$V+1, \ldots, V+32$	Pull	–	–
$V+33$	Pull	–	h

Tabel 1. Diagram iterasi dari fungsi hash Panama

Tujuan desain fungsi hash Panama adalah *hermetic* (tertutup rapat, aman, tak ada celah). Sebuah fungsi hash *hermetic* dapat dijadikan sebuah MAC yang aman dengan menambahkan sebuah kunci rahasia pada pesan masukan. Keamanannya independen terhadap posisi kunci dalam pesan tersebut. Kunci tersebut dapat saja ditambahkan di awal, di tengah, maupun di akhir pesan.

7.2.2. Skema Enkripsi Aliran Panama

Cipher aliran Panama diinisialisasi dengan kunci *K* sepanjang 256-bit, parameter diversifikasi *Q* dan melakukan 32 Pull kosong. Pada pembangkitan aliran-bit-kunci, sebuah blok 8-word *z* keluat sebagai keluaran untuk setiap iterasi. Pada prakteknya, parameter diversifikasi mengizinkan sinkronisasi ulang tanpa harus mengubah kunci.

Tujuan desain dari skema enkripsi aliran Panama adalah *hermetic* dan *K-secure*.

7.3. Pembahasan

7.3.1. Transformasi *update state*

γ adalah *nonlinear shift-invariant transformation* yang paling sederhana. Secara singkat:

- Korelasi maksimum antara masukan dan keluaran mengecil secara eksponensial dengan bobot Hamming dari vektor seleksi keluaran
- Probabilitas perbedaan propagasi mengecil secara eksponensial dengan bobot Hamming dari vektor perbedaan masukan.

Transformasi θ korespon dengan multiplikasi dari modulo binomial biner $1 \oplus x^{17}$. Hal ini dipilih dari polinomial yang dapat dibalik dengan bobot Hamming 3 pada basis properti difusi yang baik. Sebuah perbedaan masukan dapat menghasilkan tiga buah keluaran yang berbeda.

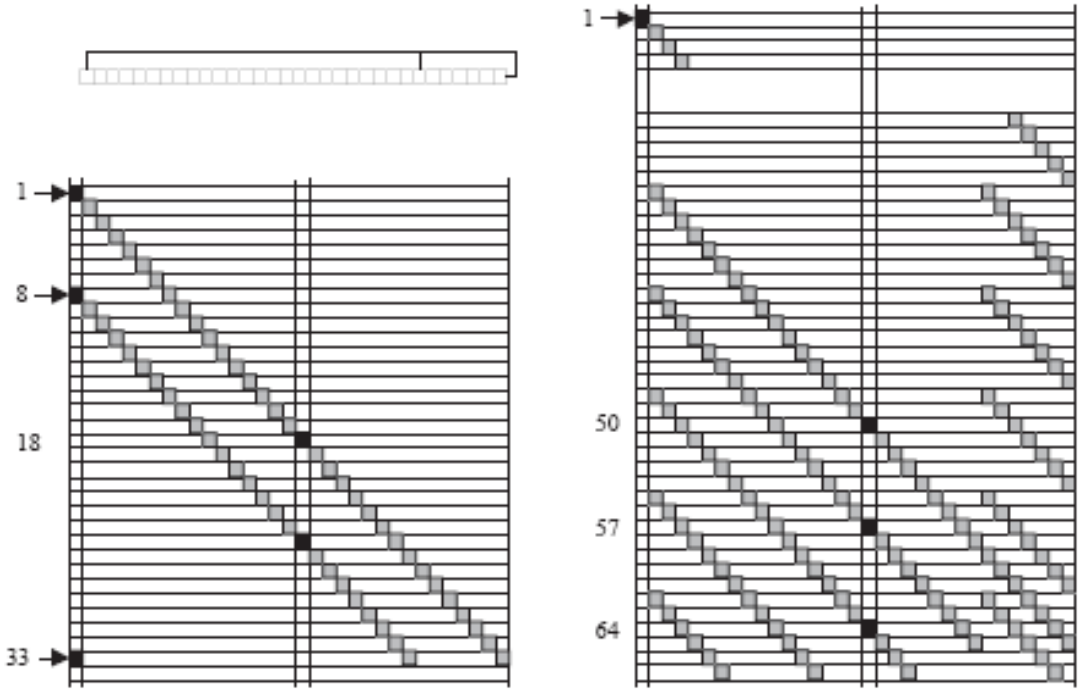
Koefisien pergeseran putar dari π membentuk sebuah array yang terdiri dari 17 konstanta yang berbeda. Faktor permutasi word 7 dipilih agar setiap komponen dari ρ bergantung pada bit state 9. Untuk parameter π yang terpilih, ia sudah diverifikasi bahwa *dot product* ρ terhadap dirinya sendiri mempunyai properti propagasi dan korelasi yang dekat untuk mengoptimalkan parameter π. Secara rata-rata, sebuah perbedaaan dalam satu bit mendifusi 6 bit setelah satu iterasi. Karena θ, γ, π, σ dapat dibalik, maka transformasi update state ρ dapat dibalik juga.

7.3.2. Fungsi hash

Desain dari fungsi hash Panama berbeda dengan algoritma turunan MD4, seperti MD5, SHA-1, dan RIPEMD-160 dalam 3 hal yang penting:

- Transformasi iterasi paralel
  Desain dari turunan MD4 mempunyai sebuah fungsi iterasi yang terkandung dalam irinya sendiri dari sebuah sekuen dari bilangan besar putaran. Pada Panama, fungsi iterasi mengandung sebuah putaran dengan struktur paralel.
- State berantai yang besar
  Fungsi hash turunan MD4 mempunyai sebuah variabel berantasi yang mempunyai ukuran yang sama dengan hasil hash, variabel berantai pada Panama membentuk state internal dan penyangga lebih dari 1 KB.
- Adanya transformasi final
  Pada desain turunan MD4, hasil hash adalah nilai akhir dari variabel berantai. Pada Panama, 32 iterasi Pull kosong membentuk sebuah transformasi final yang memetakan nilai akhir dari variabel berantai (state dan penyangga) kepada hasil hash.

Perbedaan ini adalah konsekuensi dari perbedaan strategi desain. Pada fungsi hash turunan MD4, transformasi interasi didesain untuk tahan terhadap kolisi terhadap dirinya sendiri. Mekanisme iterasi dan fakta bahwa hasil hash adalah nilai dari variabel berantai setelah blok pesan terakhir diproses, meyakinkan bahwa



Gambar 9. Propagasi perbedaaan pada penyangga dari Panama.

$t = 1$	-	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	-	-	-	-	-	-	-	-
$t = 8$	-	d_2	d_3	d_4	d_5	d_6	d_7	d_0	d_1	-	-	-	-	-	-	-	-
$t = 18$	-	-	-	-	-	-	-	-	-	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
$t = 25$	-	-	-	-	-	-	-	-	-	d_2	d_3	d_4	d_5	d_6	d_7	d_0	d_1
$t = 33$	-	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	-	-	-	-	-	-	-	-

Gambar 10. Perbedaan motif pada σ

fungsi hash tersebut tahan terhadap kolisi. Pada fungsi hash Panama, adalah difusi dan kenon-linearan yang diharapkan untuk mencegah kelemahan-kelamahan secar kriptografi.

7.3.3. Ketahanan terhadap kolisi

Hasil hash secara penuh ditentukan dari nilai akhir dari variabel berantai: state a^{V+1} dan isi penyangga b^{V+1}. Pasangan-pasangan pesan dapat ditemukan dengan nilai yang berbeda untuk variabel berantai final, kedua pesan tersebut konsisten dengan hasil hash yang sama. Dalam kasus ini, kolisi tersebut sebenarnya dihasilkan dari fakta bahwa hasil hash tidak sama dengan state hash final. Hal ini disebut kolisi *terminal*. Kolisi dimana dua pesan yang berbeda menghasilkan variabel berantai yang sama disebut *internal*. Pada fungsi-fungsi hash tanpa transformasi final, seperti MD4 dan turunannya, kolisi terminal tidak dapat terjadi.

Strategi yang mungkin untuk menemukan kolisi internal adalah mencari sebuah modus perbedaan masukan yang dapat menghasilkan trail diferensial yang berakhir dalam nol perbedaan . Strategi lainnya adalah:

- mencari poin-poin dari mode Push atau lebih umum, sekuen tetap dari mode Push
- Serangan meet-in-the-middle mengeksploitasi sifat keterbalikan dari fungsi iterasi.

Serangan-serangan ini sudah diperhitungkan dalam strategi desain dan dalam pemilihan transformasi update state dan penyangga.

Karena umpan balik dalam penyangga, sebuah modus perbedaan dalam sebuah blok pesan menghasilkan propagasi perbedaaan yang tak berhingga dalam penyangga. Hanya modus

perbedaan pada sekuen masukan yang dapat mengalami kondisi propagasi perbedaan berhingga di dalam penyangga.

Pada kolisi internal, motif perbedaan dalam state dan penyangga adalah nol sebelum state hash final dicapai. Jelas bahwa hal ini melarang motif perbedaaan di dalam sekuen masukan untuk superposisi instansi-instansi motif perbedaan sederhana pada bagian kiri gambar 9. Motif perbedaan masukan ini menghasilkan vektor perbedaan yang tidak nol pada σ pada bagian yang setidaknya terdapat 32 iterasi.

7.3.4. Skema enkripsi aliran

Untuk setiap iterasi Pull, 16 word dari penyangga dimasukkan ke dalam state dan word 8-state diberikan pada keluaran. Jumlah dari bit penyangga yang dimasukkan ke dalam state adalah dua kali lebih besar dari jumlah bit yang diberikan pada keluaran. Dapat dicek bahwa hal ini adalah kasus untuk setiap jumlah dari iterasi lebih kecil dari 12. Umpan balik dari state ke penyangga menyebabkan isi penyangga diperbarui setiap 32 iterasi. Faktor-faktor ini menyebabkan korelasi antara bit-bit keluaran dan komvinasi linear dari state dan bit penyangga menjadi terlalu kecil pada kriptanalisis.

Serangan sinkronisasi ulang dapat ditahan oleh 32 iterasi Pull kosong setelah menginisialisasi blok-blok. Karena umpan balik dari state ke penyangga pada mode Pull, state dan hampir semua tahapan penyanggan bergantung secara kompleks terhadap blok-blok akhir pada tahap inisialisasi.

7.4. Aspek Implementasi

Ketergantungan Panama terhadap operasi logika bitwise pada word 32-bit membuat algoritma ini cocok diimplementasikan pada prosesor 8, 16, ataupun 32-bit.

Akses-akses terhadap penyangga *b* dan operasi-operasi antara penyangga *b* dan state *a* dapat dilakukan secara 64-bit dalam satuan waktu pada prosesor yang mendukung ukuran word demikian. Karena *a* bukan sebuah integer kelipatan dari 64-bit, sebuah *dummy* word 32-bit harus dimasukkan terlebih dahulu ke a_0.

Jadi, θ dan γ dapat dilakukan secara efisien dengan ukuran word sampai dengan 32-bit, dan π lebih cenderung pada arsitektur 32-bit. σ dan λ dapat dilakukan secara efisien dengan ukuran word sampai dengan 64-bit. Analisi berikutnya lebih diutamakan pada prosesor 32-bit.

7.4.1. Hasil Pengujian (Benchmark)

Jumlah paralelisme yang banyak pada Panama membuat algoritma ini secara natural dapat diimplementasikan secara efisien pada sebuah prosesor.

Pertama kali Panama diuji pada prosesor Philips TriMedia TM-1000. Prosesor ini terdiri dari lima unit eksekusi 32-bit. Namun prosesor ini memiliki beberapa keterbatasan, misalnya pada π yang memanggil instruksi putar secara intensif. Prosesor ini hanya dapat memenuhi tuntutan tersebuit sebanyak dua putaran per cycle.

Algoritma ini kemudian diimplementasikan dengan bahasa C dan diuji pada prosesor Trimedia 100 MHz, dengan penyangga data sebesar 128 KB. Hasil enkripsi tecatat 470 Mbps (Mega Bits Per Second). RC4 hanya mencatat angka 75 Mbps untuk enkripsinya, sebuah peningkatan yang luar biasa dari Panama. Sebagai fungsi hash, Panama mencatat angka 510 Mbps.

Performa Panama pada prosesor Pentium Pro 200 MHz tercatat 198 Mbps sebagai cipher dan 214 Mbps sebagai fungsi hash.

Terakhir, pengujian dilakukan pada prosesor AMD Opteron 1.6 GHz. Sebagai fungsi hash, Panama mencatat angka 303,407 MB/s untuk little endian, dan 220, 832 MB/s untuk big endian. Angka ini jauh lebih unggul dari SHA-1, RIPEMD-160, maupun MD5. Sebagai cipher, Panama dapat mengenkripsi 253,717 MB/s untuk little endian, dan 192,590 MB/s untuk big endian, sebuah performa yang sangat cepat dalam enkripsi data.

8. Kesimpulan

Panama merupakan modul kriptografi yang menyediakan dua fungsi, baik sebagai cipher maupun fungsi hash. Dari hasil pengujian, Panama mempunyai tingkat keamanan yang cukup baik dan kecepatan pengolahan data yang sangat besar.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. "Diktat Kuliah IF5054 : Kriptografi". 2006. Bandung : Institut Teknologi Bandung

[2] http://en.wikipedia.org/wiki/Panama_(cipher)

[3] http://www.users.zetnet.co.uk/hopwood/crypto/scan/md.html

[4] http://en.wikipedia.org/wiki/Cryptographic_hash_function

[5] http://citeseer.ist.psu.edu/cache/papers/cs/13215/http:zSzzSzstandard.pictel.comzSzftpzSzresearchzSzsecurityzSzpanama.pdf/daemen98fast.pdf

[6] http://www.quadibloc.com/crypto/co4821.htm

[7] http://www.vlsi.ee.upatras.gr/~gselimis/papers/2004/icecs2004_selimis1.pdf

[8] http://72.14.235.104/search?q=cache:4PZzh3USSZUJ:www.cosic.esat.kuleuven.be/publications/article-1.ps+producing+collisions+for+panama&hl=id&gl=id&ct=clnk&cd=1&client=firefox-a

[9] http://www.cosic.esat.kuleuven.be/publications/thesis-16.pdf

[10] http://www.cryptopp.com/benchmarks.html

[11] http://en.wikipedia.org/wiki/Caesar_cipher

LAMPIRAN FILE INCLUDE UNTUK C

```
#ifndef __QUICKHASH_H__
#define __QUICKHASH_H__

#if defined (_MSC_VER) && (_MSC_VER >= 1020)
#pragma once
#endif

#if defined (_MSC_VER)
#if !defined( _SL_QUICKHASH_IMPLEMENTATION ) && !defined( _SL_NOFORCE_LIBS )

#ifdef _SL_STATIC
       #pragma comment( lib, "QuickHashS.lib" )
#else
       #pragma comment( lib, "QuickHash.lib" )
#endif //_SL_STATIC

#endif

#endif


#if ( defined (_MSC_VER) && (_MSC_VER >= 800) ) || defined(_STDCALL_SUPPORTED)
#define SL_HASHCALL _stdcall
#else
#define SL_HASHCALL
#endif
#define SLC_MD4_DIGESTSIZE                  16
#define SLC_MD4_HEXDIGESTSIZE        ( 2 * ( SLC_MD4_DIGESTSIZE ) + 1 )
#define SLC_MD4_CONTEXTSIZE                 88
#define SLC_MD4_BLOCKSIZE                   64
#define SLC_MD4_ALGID                       8

#define SLC_MD5_DIGESTSIZE                  16
#define SLC_MD5_HEXDIGESTSIZE        ( 2 * ( SLC_MD5_DIGESTSIZE ) + 1 )
#define SLC_MD5_CONTEXTSIZE                 88
#define SLC_MD5_BLOCKSIZE                   64
#define SLC_MD5_ALGID                       0

#define SLC_SHA1_DIGESTSIZE                 20
#define SLC_SHA1_HEXDIGESTSIZE              ( 2 * ( SLC_SHA1_DIGESTSIZE ) + 1 )
#define SLC_SHA1_CONTEXTSIZE         92
#define SLC_SHA1_BLOCKSIZE                  64
#define SLC_SHA1_ALGID                      1

#define SLC_SHA256_DIGESTSIZE        32
#define SLC_SHA256_HEXDIGESTSIZE     ( 2 * ( SLC_SHA256_DIGESTSIZE ) + 1 )
#define SLC_SHA256_CONTEXTSIZE              104
#define SLC_SHA256_BLOCKSIZE         64
#define SLC_SHA256_ALGID                    2

#define SLC_SHA512_DIGESTSIZE        64
#define SLC_SHA512_HEXDIGESTSIZE     ( 2 * ( SLC_SHA512_DIGESTSIZE ) + 1 )
#define SLC_SHA512_CONTEXTSIZE              208
#define SLC_SHA512_BLOCKSIZE         128
#define SLC_SHA512_ALGID                    3

#define SLC_SHA384_DIGESTSIZE        48
#define SLC_SHA384_HEXDIGESTSIZE     ( 2 * ( SLC_SHA384_DIGESTSIZE ) + 1 )
#define SLC_SHA384_CONTEXTSIZE              208
#define SLC_SHA384_BLOCKSIZE         128
#define SLC_SHA384_ALGID                    4

#define SLC_RIPEMD160_DIGESTSIZE     20
#define SLC_RIPEMD160_HEXDIGESTSIZE  ( 2 * ( SLC_RIPEMD160_DIGESTSIZE ) + 1 )
#define SLC_RIPEMD160_CONTEXTSIZE    92
#define SLC_RIPEMD160_BLOCKSIZE             64
```

```
#define SLC_RIPEMD160_ALGID                 7

#define SLC_RIPEMD128_DIGESTSIZE      16
#define SLC_RIPEMD128_HEXDIGESTSIZE   ( 2 * ( SLC_RIPEMD128_DIGESTSIZE ) + 1 )
#define SLC_RIPEMD128_CONTEXTSIZE     88
#define SLC_RIPEMD128_BLOCKSIZE             64
#define SLC_RIPEMD128_ALGID                 12

#define SLC_RIPEMD256_DIGESTSIZE      32
#define SLC_RIPEMD256_HEXDIGESTSIZE   ( 2 * ( SLC_RIPEMD256_DIGESTSIZE ) + 1 )
#define SLC_RIPEMD256_CONTEXTSIZE     104
#define SLC_RIPEMD256_BLOCKSIZE             64
#define SLC_RIPEMD256_ALGID                 13

#define SLC_RIPEMD320_DIGESTSIZE      40
#define SLC_RIPEMD320_HEXDIGESTSIZE   ( 2 * ( SLC_RIPEMD320_DIGESTSIZE ) + 1 )
#define SLC_RIPEMD320_CONTEXTSIZE     112
#define SLC_RIPEMD320_BLOCKSIZE             64
#define SLC_RIPEMD320_ALGID                 14

#define SLC_PANAMA_DIGESTSIZE         32
#define SLC_PANAMA_HEXDIGESTSIZE      ( 2 * ( SLC_PANAMA_DIGESTSIZE ) + 1 )
#define SLC_PANAMA_CONTEXTSIZE        1280
#define SLC_PANAMA_BLOCKSIZE          32
#define SLC_PANAMA_ALGID                    6

#define SLC_TIGER_DIGESTSIZE          24
#define SLC_TIGER_HEXDIGESTSIZE             ( 2 * ( SLC_TIGER_DIGESTSIZE ) + 1 )
#define SLC_TIGER_CONTEXTSIZE         96
#define SLC_TIGER_BLOCKSIZE                 64
#define SLC_TIGER_ALGID                            5

#define SLC_MD2_DIGESTSIZE                  16
#define SLC_MD2_HEXDIGESTSIZE         ( 2 * ( SLC_MD2_DIGESTSIZE ) + 1 )
#define SLC_MD2_CONTEXTSIZE                 68
#define SLC_MD2_ALGID                       9

#define SLC_CRC32_DIGESTSIZE          4
#define SLC_CRC32_HEXDIGESTSIZE             ( 2 * ( SLC_CRC32_DIGESTSIZE ) + 1 )
#define SLC_CRC32_CONTEXTSIZE         4
#define SLC_CRC32_ALGID             10

#define SLC_CRC16_DIGESTSIZE          2
#define SLC_CRC16_HEXDIGESTSIZE             ( 2 * ( SLC_CRC32_DIGESTSIZE ) + 1 )
#define SLC_CRC16_CONTEXTSIZE         2
#define SLC_CRC16_ALGID             15

#define SLC_CRC16C_DIGESTSIZE         2
#define SLC_CRC16C_HEXDIGESTSIZE            ( 2 * ( SLC_CRC32_DIGESTSIZE ) + 1 )
#define SLC_CRC16C_CONTEXTSIZE              2
#define SLC_CRC16C_ALGID            16

#define SLC_ADLER32_DIGESTSIZE              4
#define SLC_ADLER32_HEXDIGESTSIZE     ( 2 * ( SLC_ADLER32_DIGESTSIZE ) + 1 )
#define SLC_ADLER32_CONTEXTSIZE             4
#define SLC_ADLER32_ALGID           11


#define SLC_HMAC_CONTEXTSIZE( CONTEXTSIZE, BLOCKSIZE ) ( CONTEXTSIZE + 2 * BLOCKSIZE +
sizeof( unsigned int ) )



#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

void                          SL_HASHCALL SL_MD4_Init( void* pContext );
void                          SL_HASHCALL SL_MD4_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
```

```
void                            SL_HASHCALL SL_MD4_UpdateStr( void* pContext, const char*
pSrc );
void                            SL_HASHCALL SL_MD4_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_MD4_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_MD4_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_MD4_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_MD4_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_MD4_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_MD5_Init( void* pContext );
void                            SL_HASHCALL SL_MD5_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_MD5_UpdateStr( void* pContext, const char*
pSrc );
void                            SL_HASHCALL SL_MD5_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_MD5_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_MD5_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_MD5_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_MD5_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_MD5_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_SHA1_Init( void* pContext );
void                            SL_HASHCALL SL_SHA1_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_SHA1_UpdateStr( void* pContext, const char*
pSrc);
void                            SL_HASHCALL SL_SHA1_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_SHA1_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_SHA1_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_SHA1_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_SHA1_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_SHA1_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_SHA256_Init( void* pContext );
void                            SL_HASHCALL SL_SHA256_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_SHA256_UpdateStr( void* pContext, const
char* pSrc );
void                            SL_HASHCALL SL_SHA256_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_SHA256_FinalHex( void* pContext, void*
pDest, int bUpper );
void                            SL_HASHCALL SL_SHA256_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_SHA256_CalculateStr( void* pDest, const
char* pSrc );
void                            SL_HASHCALL SL_SHA256_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_SHA256_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_SHA512_Init( void* pContext );
void                            SL_HASHCALL SL_SHA512_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_SHA512_UpdateStr( void* pContext, const
char* pSrc );
void                            SL_HASHCALL SL_SHA512_Final( void* pContext, void* pDest );
```

```
void                        SL_HASHCALL SL_SHA512_FinalHex( void* pContext, void*
pDest, int bUpper );
void                        SL_HASHCALL SL_SHA512_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_SHA512_CalculateStr( void* pDest, const
char* pSrc );
void                        SL_HASHCALL SL_SHA512_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                        SL_HASHCALL SL_SHA512_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                        SL_HASHCALL SL_SHA384_Init( void* pContext );
void                        SL_HASHCALL SL_SHA384_Final( void* pContext, void* pDest );
void                        SL_HASHCALL SL_SHA384_FinalHex( void* pContext, void*
pDest, int bUpper );
void                        SL_HASHCALL SL_SHA384_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_SHA384_CalculateStr( void* pDest, const
char* pSrc );
void                        SL_HASHCALL SL_SHA384_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                        SL_HASHCALL SL_SHA384_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );
#define                     SL_SHA384_Update              SL_SHA512_Update
#define             SL_SHA384_UpdateStr             SL_SHA512_UpdateStr

void                        SL_HASHCALL SL_RIPEMD160_Init( void* pContext );
void                        SL_HASHCALL SL_RIPEMD160_Update( void* pContext, const
void* pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_RIPEMD160_UpdateStr( void* pContext, const
char* pSrc );
void                        SL_HASHCALL SL_RIPEMD160_Final( void* pContext, void* pDest
);
void                        SL_HASHCALL SL_RIPEMD160_FinalHex( void* pContext, void*
pDest, int bUpper );
void                        SL_HASHCALL SL_RIPEMD160_Calculate( void* pDest, const
void* pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_RIPEMD160_CalculateStr( void* pDest, const
char* pSrc );
void                        SL_HASHCALL SL_RIPEMD160_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                        SL_HASHCALL SL_RIPEMD160_CalculateStrHex( void* pDest,
const char* pSrc, int bUpper );

void                        SL_HASHCALL SL_RIPEMD128_Init( void* pContext );
void                        SL_HASHCALL SL_RIPEMD128_Update( void* pContext, const
void* pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_RIPEMD128_UpdateStr( void* pContext, const
char* pSrc );
void                        SL_HASHCALL SL_RIPEMD128_Final( void* pContext, void* pDest
);
void                        SL_HASHCALL SL_RIPEMD128_FinalHex( void* pContext, void*
pDest, int bUpper );
void                        SL_HASHCALL SL_RIPEMD128_Calculate( void* pDest, const
void* pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_RIPEMD128_CalculateStr( void* pDest, const
char* pSrc );
void                        SL_HASHCALL SL_RIPEMD128_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSize, int bUpper );
void                        SL_HASHCALL SL_RIPEMD128_CalculateStrHex( void* pDest,
const char* pSrc, int bUpper );

void                        SL_HASHCALL SL_RIPEMD256_Init( void* pContext );
void                        SL_HASHCALL SL_RIPEMD256_Update( void* pContext, const
void* pSrc, unsigned int nSrcLength );
void                        SL_HASHCALL SL_RIPEMD256_UpdateStr( void* pContext, const
char* pSrc );
void                        SL_HASHCALL SL_RIPEMD256_Final( void* pContext, void* pDest
);
void                        SL_HASHCALL SL_RIPEMD256_FinalHex( void* pContext, void*
pDest, int bUpper );
```

```
void                            SL_HASHCALL SL_RIPEMD256_Calculate( void* pDest, const
void* pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_RIPEMD256_CalculateStr( void* pDest, const
char* pSrc );
void                            SL_HASHCALL SL_RIPEMD256_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSize, int bUpper );
void                            SL_HASHCALL SL_RIPEMD256_CalculateStrHex( void* pDest,
const char* pSrc, int bUpper );

void                            SL_HASHCALL SL_RIPEMD320_Init( void* pContext );
void                            SL_HASHCALL SL_RIPEMD320_Update( void* pContext, const
void* pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_RIPEMD320_UpdateStr( void* pContext, const
char* pSrc );
void                            SL_HASHCALL SL_RIPEMD320_Final( void* pContext, void* pDest
);
void                            SL_HASHCALL SL_RIPEMD320_FinalHex( void* pContext, void*
pDest, int bUpper );
void                            SL_HASHCALL SL_RIPEMD320_Calculate( void* pDest, const
void* pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_RIPEMD320_CalculateStr( void* pDest, const
char* pSrc );
void                            SL_HASHCALL SL_RIPEMD320_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSize, int bUpper );
void                            SL_HASHCALL SL_RIPEMD320_CalculateStrHex( void* pDest,
const char* pSrc, int bUpper );

void                            SL_HASHCALL SL_PANAMA_Init( void* pContext );
void                            SL_HASHCALL SL_PANAMA_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_PANAMA_UpdateStr( void* pContext, const
char* pSrc );
void                            SL_HASHCALL SL_PANAMA_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_PANAMA_FinalHex( void* pContext, void*
pDest, int bUpper );
void                            SL_HASHCALL SL_PANAMA_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_PANAMA_CalculateStr( void* pDest, const
char* pSrc );
void                            SL_HASHCALL SL_PANAMA_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_PANAMA_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_TIGER_Init( void* pContext );
void                            SL_HASHCALL SL_TIGER_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_TIGER_UpdateStr( void* pContext, const char*
pSrc );
void                            SL_HASHCALL SL_TIGER_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_TIGER_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_TIGER_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_TIGER_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_TIGER_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_TIGER_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_MD2_Init( void* pContext );
void                            SL_HASHCALL SL_MD2_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_MD2_UpdateStr( void* pContext, const char*
pSrc );
void                            SL_HASHCALL SL_MD2_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_MD2_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_MD2_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
```

```
void                            SL_HASHCALL SL_MD2_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_MD2_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_MD2_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_CRC32_Init( void* pContext );
void                            SL_HASHCALL SL_CRC32_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_CRC32_UpdateStr( void* pContext, const char*
pSrc );
void                            SL_HASHCALL SL_CRC32_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_CRC32_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_CRC32_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_CRC32_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_CRC32_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_CRC32_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_CRC16_Init( void* pContext );
void                            SL_HASHCALL SL_CRC16_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_CRC16_UpdateStr( void* pContext, const char*
pSrc );
void                            SL_HASHCALL SL_CRC16_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_CRC16_FinalHex( void* pContext, void* pDest,
int bUpper );
void                            SL_HASHCALL SL_CRC16_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_CRC16_CalculateStr( void* pDest, const char*
pSrc );
void                            SL_HASHCALL SL_CRC16_CalculateHex( void* pDest, const void*
pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_CRC16_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_CRC16C_Init( void* pContext );
void                            SL_HASHCALL SL_CRC16C_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_CRC16C_UpdateStr( void* pContext, const
char* pSrc );
void                            SL_HASHCALL SL_CRC16C_Final( void* pContext, void* pDest );
void                            SL_HASHCALL SL_CRC16C_FinalHex( void* pContext, void*
pDest, int bUpper );
void                            SL_HASHCALL SL_CRC16C_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_CRC16C_CalculateStr( void* pDest, const
char* pSrc );
void                            SL_HASHCALL SL_CRC16C_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_CRC16C_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

void                            SL_HASHCALL SL_ADLER32_Init( void* pContext );
void                            SL_HASHCALL SL_ADLER32_Update( void* pContext, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_ADLER32_UpdateStr( void* pContext, const
char* pSrc );
void                            SL_HASHCALL SL_ADLER32_Final( void* pContext, void* pDest
);
void                            SL_HASHCALL SL_ADLER32_FinalHex( void* pContext, void*
pDest, int bUpper );
void                            SL_HASHCALL SL_ADLER32_Calculate( void* pDest, const void*
pSrc, unsigned int nSrcLength );
void                            SL_HASHCALL SL_ADLER32_CalculateStr( void* pDest, const
char* pSrc );
```

```
void                            SL_HASHCALL SL_ADLER32_CalculateHex( void* pDest, const
void* pSrc, unsigned int nSrcLength, int bUpper );
void                            SL_HASHCALL SL_ADLER32_CalculateStrHex( void* pDest, const
char* pSrc, int bUpper );

int                                     SL_HASHCALL SL_HMAC_Init( void* pContext, unsigned
int nAlgID, const void* pKey, unsigned int nKeyLength );
int                                     SL_HASHCALL SL_HMAC_InitKeyStr( void* pContext,
unsigned int nAlgID, const char* pKey );
int                                     SL_HASHCALL SL_HMAC_Update( void* pContext, const
void* pSrc, unsigned int nSrcLength );
int                                     SL_HASHCALL SL_HMAC_UpdateStr( void* pContext, const
char* pSrc );
int                                     SL_HASHCALL    SL_HMAC_Final( void* pContext, void*
pDest );
int                                     SL_HASHCALL SL_HMAC_FinalHex( void* pContext, void*
pDest, int bUpper );
int                                     SL_HASHCALL SL_HMAC_Calculate( unsigned int nAlgID,
void* pDest, const void* pSrc, unsigned int nSrcLength, const void* pKey, unsigned int
nKeyLength );
int                                     SL_HASHCALL SL_HMAC_CalculateStr( unsigned int
nAlgID, void* pDest, const char* pSrc, const char* pKey );
int                                     SL_HASHCALL SL_HMAC_CalculateHex( unsigned int
nAlgID, void* pDest, const void* pSrc, unsigned int nSrcLength, const void* pKey,
unsigned int nKeyLength, int bUpper );
int                                     SL_HASHCALL SL_HMAC_CalculateStrHex( unsigned int
nAlgID, void* pDest, const char* pSrc, const char* pKey, int bUpper );
unsigned int          SL_HASHCALL SL_HMAC_GetDigestSize( const void* pContext );

#ifdef __cplusplus
}

#ifdef NO_NAMESPACE

#ifndef USING_NAMESPACE
#define USING_NAMESPACE( X )
#endif

#ifndef NAMESPACE_BEGIN
#define NAMESPACE_BEGIN( X )
#endif

#ifndef NAMESPACE_END
#define NAMESPACE_END
#endif

#else

#ifndef USING_NAMESPACE
#define USING_NAMESPACE( X ) using namespace X;
#endif

#ifndef NAMESPACE_BEGIN
#define NAMESPACE_BEGIN( X ) namespace X {
#endif

#ifndef NAMESPACE_END
#define NAMESPACE_END }
#endif

#endif


NAMESPACE_BEGIN( QuickHash )


class CMD4
{
public:
        enum{ DIGESTSIZE = SLC_MD4_DIGESTSIZE, HEXDIGESTSIZE = SLC_MD4_HEXDIGESTSIZE,
BLOCKSIZE = SLC_MD4_BLOCKSIZE, ID = SLC_MD4_ALGID, CONTEXTSIZE = SLC_MD4_CONTEXTSIZE };
```

```
        CMD4(){ SL_MD4_Init( m_context ); }
        ~CMD4(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){ SL_MD4_Update(
m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_MD4_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_MD4_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_MD4_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_MD4_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_MD4_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_MD4_CalculateHex( pDest , pSrc, nSrcLength, bUpper
? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_MD4_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CMD5
{
public:
        enum{ DIGESTSIZE = SLC_MD5_DIGESTSIZE, HEXDIGESTSIZE = SLC_MD5_HEXDIGESTSIZE,
BLOCKSIZE = SLC_MD5_BLOCKSIZE, ID = SLC_MD5_ALGID, CONTEXTSIZE = SLC_MD5_CONTEXTSIZE };

        CMD5(){ SL_MD5_Init( m_context ); }
        ~CMD5(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){ SL_MD5_Update(
m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_MD5_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_MD5_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_MD5_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_MD5_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_MD5_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_MD5_CalculateHex( pDest , pSrc, nSrcLength, bUpper
? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_MD5_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CSHA1
{
public:
        enum{ DIGESTSIZE = SLC_SHA1_DIGESTSIZE, HEXDIGESTSIZE = SLC_SHA1_HEXDIGESTSIZE,
BLOCKSIZE = SLC_SHA1_BLOCKSIZE, ID = SLC_SHA1_ALGID, CONTEXTSIZE = SLC_SHA1_CONTEXTSIZE
};

        CSHA1(){ SL_SHA1_Init( m_context ); }
        ~CSHA1(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){ SL_SHA1_Update(
m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_SHA1_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_SHA1_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_SHA1_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }
```

```
        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_SHA1_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_SHA1_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_SHA1_CalculateHex( pDest , pSrc, nSrcLength, bUpper
? 1 : 0 ); }
        static void CalculateStrHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_SHA1_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CSHA256
{
public:
        enum{ DIGESTSIZE = SLC_SHA256_DIGESTSIZE, HEXDIGESTSIZE =
SLC_SHA256_HEXDIGESTSIZE, BLOCKSIZE = SLC_SHA256_BLOCKSIZE, ID = SLC_SHA256_ALGID,
CONTEXTSIZE = SLC_SHA256_CONTEXTSIZE };

        CSHA256(){ SL_SHA256_Init( m_context ); }
        ~CSHA256(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_SHA256_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_SHA256_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_SHA256_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_SHA256_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_SHA256_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_SHA256_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_SHA256_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_SHA256_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CSHA512
{
public:
        enum{ DIGESTSIZE = SLC_SHA512_DIGESTSIZE, HEXDIGESTSIZE =
SLC_SHA512_HEXDIGESTSIZE, BLOCKSIZE = SLC_SHA512_BLOCKSIZE, ID = SLC_SHA512_ALGID,
CONTEXTSIZE = SLC_SHA512_CONTEXTSIZE };

        CSHA512(){ SL_SHA512_Init( m_context ); }
        ~CSHA512(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_SHA512_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_SHA512_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_SHA512_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_SHA512_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_SHA512_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_SHA512_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_SHA512_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
```

```
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_SHA512_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CSHA384
{
public:
        enum{ DIGESTSIZE = SLC_SHA384_DIGESTSIZE, HEXDIGESTSIZE =
SLC_SHA384_HEXDIGESTSIZE, BLOCKSIZE = SLC_SHA384_BLOCKSIZE, ID = SLC_SHA384_ALGID,
CONTEXTSIZE = SLC_SHA384_CONTEXTSIZE };

        CSHA384(){ SL_SHA384_Init( m_context ); }
        ~CSHA384(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_SHA384_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_SHA384_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_SHA384_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_SHA384_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_SHA384_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_SHA384_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_SHA384_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_SHA384_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CRIPEMD160
{
public:
        enum{ DIGESTSIZE = SLC_RIPEMD160_DIGESTSIZE, HEXDIGESTSIZE =
SLC_RIPEMD160_HEXDIGESTSIZE, BLOCKSIZE = SLC_RIPEMD160_BLOCKSIZE, ID =
SLC_RIPEMD160_ALGID, CONTEXTSIZE = SLC_RIPEMD160_CONTEXTSIZE };

        CRIPEMD160(){ SL_RIPEMD160_Init( m_context ); }
        ~CRIPEMD160(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_RIPEMD160_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_RIPEMD160_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_RIPEMD160_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_RIPEMD160_FinalHex(
m_context, pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_RIPEMD160_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_RIPEMD160_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_RIPEMD160_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_RIPEMD160_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CRIPEMD128
{
```

```
public:
	enum{ DIGESTSIZE = SLC_RIPEMD128_DIGESTSIZE, HEXDIGESTSIZE =
SLC_RIPEMD128_HEXDIGESTSIZE, BLOCKSIZE = SLC_RIPEMD128_BLOCKSIZE, ID =
SLC_RIPEMD128_ALGID, CONTEXTSIZE = SLC_RIPEMD128_CONTEXTSIZE };

	CRIPEMD128(){ SL_RIPEMD128_Init( m_context ); }
	~CRIPEMD128(){}

	void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_RIPEMD128_Update( m_context, pSrc, nSrcLength ); }
	void Update( const char* pSrc ){ SL_RIPEMD128_UpdateStr( m_context, pSrc ); }
	void Final( unsigned char* pDest ){ SL_RIPEMD128_Final( m_context, pDest ); }
	void FinalHex( char* pDest, bool bUpper = false ){ SL_RIPEMD128_FinalHex(
m_context, pDest, bUpper ? 1 : 0 ); }

	static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_RIPEMD128_Calculate( pDest, pSrc, nSrcLength ); }
	static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_RIPEMD128_CalculateStr( pDest, pSrc ); }
	static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_RIPEMD128_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
	static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_RIPEMD128_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
	unsigned char m_context[ CONTEXTSIZE ];
};

class CRIPEMD256
{
public:
	enum{ DIGESTSIZE = SLC_RIPEMD256_DIGESTSIZE, HEXDIGESTSIZE =
SLC_RIPEMD256_HEXDIGESTSIZE, BLOCKSIZE = SLC_RIPEMD256_BLOCKSIZE, ID =
SLC_RIPEMD256_ALGID, CONTEXTSIZE = SLC_RIPEMD256_CONTEXTSIZE };

	CRIPEMD256(){ SL_RIPEMD256_Init( m_context ); }
	~CRIPEMD256(){}

	void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_RIPEMD256_Update( m_context, pSrc, nSrcLength ); }
	void Update( const char* pSrc ){ SL_RIPEMD256_UpdateStr( m_context, pSrc ); }
	void Final( unsigned char* pDest ){ SL_RIPEMD256_Final( m_context, pDest ); }
	void FinalHex( char* pDest, bool bUpper = false ){ SL_RIPEMD256_FinalHex(
m_context, pDest, bUpper ? 1 : 0 ); }

	static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_RIPEMD256_Calculate( pDest, pSrc, nSrcLength ); }
	static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_RIPEMD256_CalculateStr( pDest, pSrc ); }
	static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_RIPEMD256_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
	static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_RIPEMD256_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
	unsigned char m_context[ CONTEXTSIZE ];
};

class CRIPEMD320
{
public:
	enum{ DIGESTSIZE = SLC_RIPEMD320_DIGESTSIZE, HEXDIGESTSIZE =
SLC_RIPEMD320_HEXDIGESTSIZE, BLOCKSIZE = SLC_RIPEMD320_BLOCKSIZE, ID =
SLC_RIPEMD320_ALGID, CONTEXTSIZE = SLC_RIPEMD320_CONTEXTSIZE };

	CRIPEMD320(){ SL_RIPEMD320_Init( m_context ); }
	~CRIPEMD320(){}
```

```
        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_RIPEMD320_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_RIPEMD320_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_RIPEMD320_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_RIPEMD320_FinalHex(
m_context, pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_RIPEMD320_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_RIPEMD320_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_RIPEMD320_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_RIPEMD320_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CPanama
{
public:
        enum{ DIGESTSIZE = SLC_PANAMA_DIGESTSIZE, HEXDIGESTSIZE =
SLC_PANAMA_HEXDIGESTSIZE, BLOCKSIZE = SLC_PANAMA_BLOCKSIZE, ID = SLC_PANAMA_ALGID,
CONTEXTSIZE = SLC_PANAMA_CONTEXTSIZE };

        CPanama(){ SL_PANAMA_Init( m_context ); }
        ~CPanama(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_PANAMA_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_PANAMA_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_PANAMA_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_PANAMA_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_PANAMA_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_PANAMA_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_PANAMA_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_PANAMA_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CTiger
{
public:
        enum{ DIGESTSIZE = SLC_TIGER_DIGESTSIZE, HEXDIGESTSIZE = SLC_TIGER_HEXDIGESTSIZE,
BLOCKSIZE = SLC_TIGER_BLOCKSIZE, ID = SLC_TIGER_ALGID, CONTEXTSIZE =
SLC_TIGER_CONTEXTSIZE };

        CTiger(){ SL_TIGER_Init( m_context ); }
        ~CTiger(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_TIGER_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_TIGER_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_TIGER_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_TIGER_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_TIGER_Calculate( pDest, pSrc, nSrcLength ); }
```

```
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_TIGER_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_TIGER_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_TIGER_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CMD2
{
public:
        enum{ DIGESTSIZE = SLC_MD2_DIGESTSIZE, HEXDIGESTSIZE = SLC_MD2_HEXDIGESTSIZE,
CONTEXTSIZE = SLC_MD2_CONTEXTSIZE };

        CMD2(){ SL_MD2_Init( m_context ); }
        ~CMD2(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){ SL_MD2_Update(
m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_MD2_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_MD2_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_MD2_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_MD2_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_MD2_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_MD2_CalculateHex( pDest , pSrc, nSrcLength, bUpper
? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_MD2_CalculateStrHex( pDest , pSrc,bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

class CCRC32
{
public:
        enum{ DIGESTSIZE = SLC_CRC32_DIGESTSIZE, HEXDIGESTSIZE = SLC_CRC32_HEXDIGESTSIZE,
CONTEXTSIZE = SLC_CRC32_CONTEXTSIZE };

        CCRC32(){ SL_CRC32_Init( m_context ); }
        ~CCRC32(){}

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_CRC32_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_CRC32_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_CRC32_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_CRC32_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_CRC32_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_CRC32_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_CRC32_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_CRC32_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};
```

```
class CCRC16
{
public:
	enum{ DIGESTSIZE = SLC_CRC16_DIGESTSIZE, HEXDIGESTSIZE = SLC_CRC16_HEXDIGESTSIZE,
CONTEXTSIZE = SLC_CRC16_CONTEXTSIZE };

	CCRC16(){ SL_CRC16_Init( m_context ); }
	~CCRC16(){}

	void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_CRC16_Update( m_context, pSrc, nSrcLength ); }
	void Update( const char* pSrc ){ SL_CRC16_UpdateStr( m_context, pSrc ); }
	void Final( unsigned char* pDest ){ SL_CRC16_Final( m_context, pDest ); }
	void FinalHex( char* pDest, bool bUpper = false ){ SL_CRC16_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

	static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_CRC16_Calculate( pDest, pSrc, nSrcLength ); }
	static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_CRC16_CalculateStr( pDest, pSrc ); }
	static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_CRC16_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
	static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_CRC16_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
	unsigned char m_context[ CONTEXTSIZE ];
};

class CCRC16C
{
public:
	enum{ DIGESTSIZE = SLC_CRC16C_DIGESTSIZE, HEXDIGESTSIZE =
SLC_CRC16C_HEXDIGESTSIZE, CONTEXTSIZE = SLC_CRC16C_CONTEXTSIZE };

	CCRC16C(){ SL_CRC16C_Init( m_context ); }
	~CCRC16C(){}

	void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_CRC16C_Update( m_context, pSrc, nSrcLength ); }
	void Update( const char* pSrc ){ SL_CRC16C_UpdateStr( m_context, pSrc ); }
	void Final( unsigned char* pDest ){ SL_CRC16C_Final( m_context, pDest ); }
	void FinalHex( char* pDest, bool bUpper = false ){ SL_CRC16C_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

	static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_CRC16C_Calculate( pDest, pSrc, nSrcLength ); }
	static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_CRC16C_CalculateStr( pDest, pSrc ); }
	static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_CRC16C_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
	static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_CRC16C_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
	unsigned char m_context[ CONTEXTSIZE ];
};

class CAdler32
{
public:
	enum{ DIGESTSIZE = SLC_ADLER32_DIGESTSIZE, HEXDIGESTSIZE =
SLC_ADLER32_HEXDIGESTSIZE, CONTEXTSIZE = SLC_ADLER32_CONTEXTSIZE };

	CAdler32(){ SL_ADLER32_Init( m_context ); }
	~CAdler32(){}
```

```
        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){
SL_ADLER32_Update( m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_ADLER32_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_ADLER32_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_ADLER32_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength ) { SL_ADLER32_Calculate( pDest, pSrc, nSrcLength ); }
        static void Calculate( unsigned char* pDest, const char* pSrc ) {
SL_ADLER32_CalculateStr( pDest, pSrc ); }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, bool bUpper = false ){ SL_ADLER32_CalculateHex( pDest , pSrc, nSrcLength,
bUpper ? 1 : 0 ); }
        static void CalculateHex( char* pDest, const char* pSrc, bool bUpper = false ){
SL_ADLER32_CalculateStrHex( pDest , pSrc, bUpper ? 1 : 0 ); }

private:
        unsigned char m_context[ CONTEXTSIZE ];
};

template< class T >
class CHMAC
{
public:
        enum { DIGESTSIZE = T::DIGESTSIZE, HEXDIGESTSIZE = T::HEXDIGESTSIZE, BLOCKSIZE =
T::BLOCKSIZE, CONTEXTSIZE = SLC_HMAC_CONTEXTSIZE( T::CONTEXTSIZE, T::BLOCKSIZE ),
DEFAULTKEYLEN = 16 };
        CHMAC(){ SL_HMAC_Init( m_context, T::ID, NULL, 0 ); }
        explicit CHMAC( const unsigned char* pKey, unsigned int nKeyLength = DEFAULTKEYLEN
){ SL_HMAC_Init( m_context, T::ID, pKey, nKeyLength ); }
        explicit CHMAC( const char* pKey ){ SL_HMAC_InitKeyStr( m_context, T::ID, pKey );
}
        ~CHMAC(){}

        void Init( const unsigned char* pKey, unsigned int nKeyLength = DEFAULTKEYLEN ){
SL_HMAC_Init( m_context, T::ID, pKey, nKeyLength ); }
        void Init( const char* pKey ){ SL_HMAC_InitKeyStr( m_context, T::ID, pKey ); }

        void Update( const unsigned char* pSrc, unsigned int nSrcLength ){ SL_HMAC_Update(
m_context, pSrc, nSrcLength ); }
        void Update( const char* pSrc ){ SL_HMAC_UpdateStr( m_context, pSrc ); }
        void Final( unsigned char* pDest ){ SL_HMAC_Final( m_context, pDest ); }
        void FinalHex( char* pDest, bool bUpper = false ){ SL_HMAC_FinalHex( m_context,
pDest, bUpper ? 1 : 0 ); }

        static void Calculate( unsigned char* pDest, const unsigned char* pSrc, unsigned
int nSrcLength, const unsigned char* pKey, unsigned int nKeyLength = DEFAULTKEYLEN )
        {
                SL_HMAC_Calculate( T::ID, pDest, pSrc, nSrcLength, pKey, nKeyLength );
        }
        static void Calculate( unsigned char* pDest, const char* pSrc, const char* pKey )
        {
                SL_HMAC_CalculateStr( T::ID, pDest, pSrc, pKey );
        }
        static void CalculateHex( char* pDest, const unsigned char* pSrc, unsigned int
nSrcLength, const unsigned char* pKey, unsigned int nKeyLength = DEFAULTKEYLEN, bool
bUpper = false )
        {
                SL_HMAC_CalculateHex( T::ID, pDest, pSrc, nSrcLength, pKey, nKeyLength,
bUpper ? 1 : 0 );
        }

        static void CalculateHex( char* pDest, const char* pSrc, const char* pKey, bool
bUpper = false )
        {
                SL_HMAC_CalculateStrHex( T::ID, pDest, pSrc, pKey, bUpper ? 1 : 0 );
        }

private:
        unsigned char m_context[ CONTEXTSIZE ];
```

```
};

NAMESPACE_END

#endif /* __cplusplus */

#endif//#ifndef __QUICKHASH_H__
```