BAB II

LANDASAN TEORI

2.1 Android

2.1.1 Definisi Android

Android adalah sebuah tumpukan *software open-source* untuk berbagai macam jenis perangkat *mobile* dan berhubungan langsung dengan proyek *open-source* yang dipimpin oleh Google (Android, 2012).

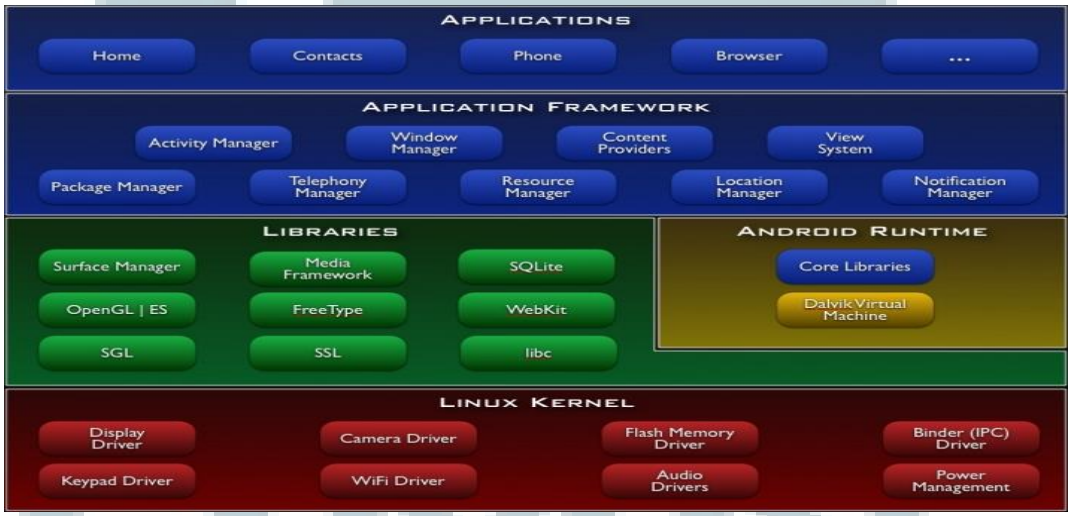
2.1.2 Sejarah Android

Android dikembangkan oleh Android Inc., Android Inc. pertama kali didirikan pada tahun 2003 oleh Andy Rubin, Rich Miner, dan Chris White. Pada awal pengembangan sistem operasi, Android adalah sistem operasi yang ditujukan sebagai sistem operasi untuk kamera digital. Tetapi pada pengembangan selanjutnya, sistem operasi Android lebih dikembangkan untuk menjadi sistem operasi perangkat *mobile* dan *embedded system*. Pada Tahun 2005 Android Inc. dibeli seluruhnya oleh Google Inc. dan menjadi bagian dari perusahaan Google. Pada tanggal 5 November 2007, *Open Handset Alliance* yang dipimpin oleh Google dan terdiri dari perusahaan-perusahaan besar seperti HTC, Intel, dan Samsung menampakkan diri kepada media. *Open Handset Alliance* merupakan aliansi yang mengembangkan standar terbuka bagi perangkat *mobile*, para anggota yang tergabung ke dalam aliansi berkomitmen untuk membuat perangkat dan servis dengan menggunakan sistem operasi Android (Open Handset Alliance, 2007). Setelah pengembangan sistem operasi yang cukup lama, akhirnya pada tanggal 12 November 2007 *Software Development Kit* versi beta

untuk Android diluncurkan. Setelah itu, pada tanggal 23 September 2008, Google dan HTC merilis perangkat Android untuk pertama kalinya yang bernama "HTC Dream G1". Sampai pada saat tulisan ini dibuat, Android masih terus berkembang dan menjadi salah satu sistem operasi yang diminati oleh konsumen telepon pintar.

2.1.3 Arsitektur Android

Menurut Bellosa 2010, sistem operasi Android berbeda dari sistem operasi lain seperti iOS milik Apple, webOS milik Palm atau Symbian karena aplikasi Android ditulis dengan bahasa pemograman Java dan berjalan di mesin virtual. Android memiliki lapisan-lapisan penting untuk mendukung kesuluruhan sistem operasi, seperti yang terlihat pada gambar di bawah ini:



Gambar 2.1. Arsitektur Sistem Operasi Android

(Sumber: *Analysis of the Android Architecture*, 2010)

Berikut penjelasan dari lapisan-lapisan yang ada pada arsitektur sistem operasi Android.

1. *Application*

   Lapisan *application* adalah lapisan paling luar dari arsitektur Android dan dapat dilihat oleh pengguna perangkat Android. Pada lapisan ini terdapat semua aplikasi yang berjalan pada perangkat Android, seperti kontak, *browser*, *phone*, dan lain-lain. Semua aplikasi berjalan di *sandbox* yang terpisah pada mesin *virtual* Dalvik dan dapat terbentuk dari beberapa komponen seperti: *activities, services, broadcast receivers,* dan *content providers*.

2. *Application Framework*

   Framework pada lapisan ini ditulis dalam bahasa pemograman Java dan menyediakan abstraksi dan menjadi dasar untuk mendukung *native libraries* dan mesin virtual Dalvik dalam menjalankan aplikasi.

3. *Android Runtime* dan *Libraries*

   Android runtime terdiri dari mesin virtual Dalvik dan Java *core libraries*. Mesin virtual Dalvik adalah penerjemah kode *byte* yang sudah ditransformasi dari kode *byte* Java menjadi kode *byte* Dalvik. *Core libraries* dari Java sepenuhnya akan diterjemahkan oleh mesin virtual Dalvik yang kemudian dikompilasi bersama dengan mesin virtual Dalvik menjadi *native code*. *Libraries* sendiri pada lapisan ini berfungsi sebagai pendukung untuk menjalankan tugas-tugas di dalam sistem operasi Android.

4. Linux Kernel

   Sistem operasi Android memakai Linux kernel yang sudah dimodifikasi sebagai otak dari sistem operasi. Modifikasi pada kernel Linux ditujukan untuk berbagai kebutuhan seperti manajemen daya, manajemen memori dan *runtime environment*.

2.1.4 Dukungan Jaringan

Untuk koneksi kepada jaringan, pada Android tersedia paket dan *classes* yang sangat beragam dari Java. Berikut adalah contoh paket dan *classes* yang disediakan untuk koneksi kepada jaringan di Android.

1. Android.net.*

   Merupakan paket yang sama dengan java.net, tetapi dengan fitur yang lebih beragam.

2. Android.http.net

   Paket untuk menangani koneksi SSL.

3. Java.net.*

   Merupakan paket asli dari Java *network classes* seperti soket, HTTP sederhana dan *packets*.

4. Org.apache.*

   Paket untuk menangani berbagai macam kebutuhan HTTP dengan fitur yang lebih luas.

2.1.5 Dukungan Enkripsi

Android memakai Java sebagai bahasa untuk pengembangan program, maka secara otomatis pada Android terdapat paket dan *classes* untuk mendukung enkripsi data. Berikut adalah contoh dari beberapa paket tersebut:

1. Java.security.*

   Merupakan paket berupa *service provider infrastructure* (SPI) yang dapat diperpanjang. Paket ini diperlukan untuk menggunakan dan mendefinisikan *services* seperti *Certificates*, *keys*, *KeyStores*, *MessageDigests*, dan *Signatures*.

2. Javax.crypto.*

Paket ini menyediakan *class* dan *interface* untuk aplikasi yang menggunakan algoritma untuk enkripsi, dekripsi atau persetujuan kunci.

Pada package java.security teradpat beberapa class yang dapat digunakan untuk melakukan enkripsi dan dekripsi data. Untuk menunjang penggunaan enkripsi pada penelitian ini, maka digunakan *class* java.security.KeyGenerator, java.security.KeyPairGenerator, dan java.security.Messadigest. Berikut adalah daftar dari algoritma enrkipsi yang dapat digunakan di dalam class java.security.KeyGenerator:

Tabel 2.1 Tabel Daftar Algoritma Enkripsi Dari Class *KeyGenerator*

Algorithm Name	Description
AES	*Key generator for use with the AES algorithm*
ARCFOUR	*Key generator for use with the ARCFOUR* (RC4) *algorithm*
Blowfish	*Key generator for use with the Blowfish algorithm*
DES	*Key generator for use with the DES algorithm*
DESede	*Key generator for use with the DESede algorithm*
DESede	*Key generator for use with the DESede algorithm*
HmacMD5	*Key generator for use with the HmacMD5 algorithm*
HmacSHA1 HmacSHA256 HmacSHA384 HmacSHA512	*Key generator for use with the various flavors of the HmacSHA algorithm*
RC2	*Key generator for use with the RC2 algorithm*

Berikut adalah daftar algoritma enkripsi yang dapat digunakan pada *Class KeyPair Generator*.

Tabel 2.2 Tabel Daftar Algoritma Enkrips dari *Class KeyPairGenerator*

Algorithm Name	Description
Diffie Hellman	*Generates keypairs for the* Diffie-Hellman *KeyAgreement algorithm*.
DSA	*Generates keypairs for the Digital Signature Algorithm*
RSA	*Generates keypairs for the RSA algorithm* (*Signature/Chiper*)
EC	*Generates keypairs for the Elliptic Curve Algorithm*

Pada algoritma enkripsi terdapat kunci enkripsi yang digunakan untuk melakukan enkripsi. Berikut adalah batasan panjang kunci dari algoritma enkripsi yang disediakan oleh bahasa pemograman Java:

Tabel 2.3 Tabel Batas Panjang Kunci

Algorithm	Maximum Keysize
DES	64
DESede	*
RC2	128
RC4	128
RC5	128
RSA	*
All Others	128

Berikut adalah daftar algoritma enkripsi yang dapat digunakan pada *class MessageDigest*:

Tabel 2.4 Tabel Daftar Algoritma Message Digest dari *Class* MessageDigest

Algorithm Name	Description
MD2	*The* MD2 *message digest algorithm as defined in* RFC 1319
MD5	*The* MD5 *message digest algorithm as defined in* RFC 1321

Tabel 2.4 Tabel Daftar Algoritma Message Digest dari *Class* MessageDigest (Lanjutan)

Algorithm	Maximum Keysize
SHA-1 SHA-256 SHA-384 SHA-512	*Has algorithms defined in the* FIPS PUB-180-2. SHA-256 *is* 256-*bit hash function intended to provide* 128 *bits of security againts collision attacks, while* SHA-512 *is a* 512-*bit hash function.*

2.1.6 Enkripsi AES pada Android

Untuk melakukan enkripsi AES pada Android, dibutuhkan beberapa *package* atau *libraries* yang disediakan oleh bahasa pemograman Java:

1. Javax.crypto.Cipher

   Class ini menyediakan akses untuk implementasi *cryptographic chipers* untuk proses enkripsi dan dekripsi.

2. Javax.crypto.KeyGenerator

   Class ini menyediakan *application programming interface* (API) yang bersifat publik untuk membuat kunci *symmetric cryptographic*.

3. Javax.crypto.SecretKey

   Interface ini adalah pembuat *interface* untuk mengelompokkan *secret keys* dan menyediakan keamanan untuk pengimplementasian *interface*, sehingga komparasi dilakukan dengan menggunakan kunci yang sesungguhnya dan tidak memakai *object reference*.

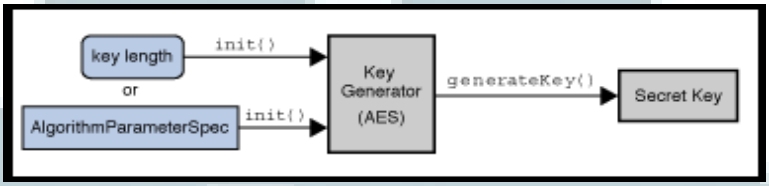
4. Javax.crypto.spec.SecretKeySpec

   Merupakan sebuah spesifikasi kunci untuk *class secretKey* dan juga sebuah implementasi *secret key* yang *provider-independent*.

5. Java.security.SecureRandom

Class ini menghasilkan angka *pseudo-random* yang aman untuk *crypthography*.

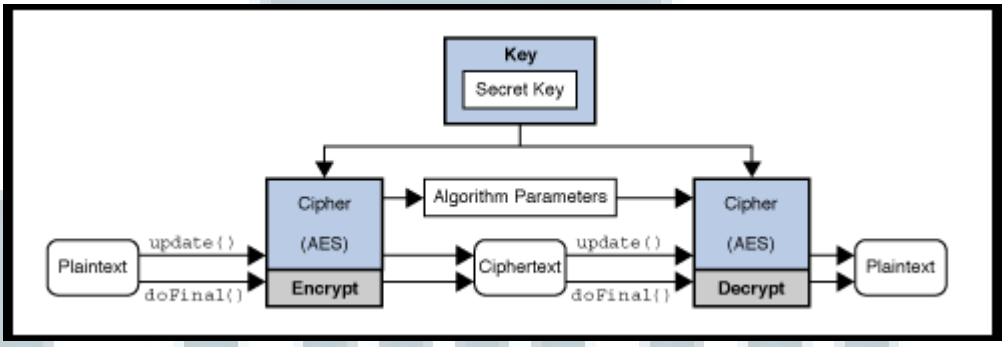
Untuk menghasilkan *symmetric key* dari algoritma enkripsi AES digunakan *class KeyGenerator*. *Class* ini harus dinisialisasi terlebih dahulu dengan menentukan spesifikasi algoritma dan panjang kunci yang dipakai, seperti gambar di bawah ini.



Gambar 2.2 *The* KeyGenerator *Class*

(Sumber*: Java Cryptography Architecture* (JCA) *Reference Guide*, 2013)

Setelah *secret key* sudah didapatkan, *secret key* dapat digunakan untuk meng-enkripsi data dengan menggunakan *class chiper*. Pada saat penggunaannya *class chiper* harus diinisialisasi terlebih dahulu penggunaannya, apakah untuk mengenkripsi data atau mengdekripsi data. Proses enkripsi dan dekripsi menggunakan *chiper class* dapat dilihat pada gambar berikut ini.



Gambar 2.3 *The* Chiper *Class*

(Sumber: *Java Cryptography Architecture* (JCA) *Reference Guide*, 2013)

2.1.7 Enkripsi RSA pada Android

Untuk dapat melakukan enkripsi menggunakan algoritma enkripsi RSA, diperlukan beberapa *class* atau *libraries* yang sudah tersedia pada *Java Development Kit* (JDK):

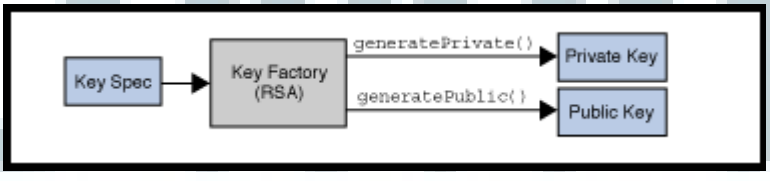
1. Java.security.KeyFactory

   Class KeyFactory adalah sebuah *engine class* yang didesain untuk melakukan konversi antara *opaque cryptographic keys* dan *key spesification*.

2. Java.security.KeyPair

   Class KeyPair adalah sebuah tempat penyimpanan sederhana untuk sebuah *key pair* (*public key* dan *private key*).

Pada penggunaan *class KeyPairGenerator*, pertama-tama dilakukan penentuan spesifikasi kunci algoritma enkripsi apa yang akan digunakan. Setelah spesifikasi kunci ditentukan, maka selanjutnya dilakukan initialisasi terhadap *object* dari *KeyPairGenerator* yang sudah terbentuk. Setelah selesai di-inisialisasi, *public key* dan *private key* dapat dibentuk dan disimpan di dalam *object* yang dibentuk oleh *class KeyPair*. Selanjutnya *private key* dan *public key* dapat diambil dari object *object KeyPair* dan dipisah. Berikut adalah gambar pembentukan *public key* dan *private key* oleh *KeyFactory class*:

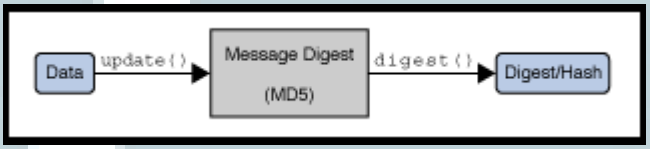


Gambar 2.4 *The* KeyFactory *Class*

(Sumber: *Java Cryptography Architecture* (JCA) *Reference Guide*, 2013)

2.1.8 Message-diggest MD5 pada Android

Untuk mendapatkan *Message Authentication Code* (MAC) dengan menggunakan algoritma *message-digest* MD5 pada Android diperlukan *class* java.security.MessageDigest. *Class* java.security.MessageDigest adalah sebuah *engine class* yang didesain untuk menyediakan *cryptographical secure message digests* seperti SHA-1 atau MD5. Dalam penggunaannya, pertama-tama harus ditentukan dahulu algoritma *message digest* apa yang akan digunakan. Setelah itu, data yang sudah diubah bentuknya menjadi *byte* akan diproses di dalam object dari *class* MessageDigest seperti gambar di bawah ini:



Gambar 2.5 *The* MessageDigest *Class*

(Sumber: *Java Cryptography Architecture* (JCA) *Reference Guide*, 2013)

2.2 Google Cloud Messaging untuk Android

2.2.1 Definisi Google Cloud Messaging untuk Android

Google Cloud Messaging (GCM) untuk Android adalah sebuah *service* gratis untuk membantu para pengembang mengirim data dari *server* ke aplikasi pada perangkat Android, dan dari aplikasi pada perangkat Android ke *cloud* (Android, 2013). Data yang dikirimkan dapat berupa pesan pemberitahuan ringan dengan maksimal besar data 4kb. GCM untuk Android mempunyai karakteristik yang berbeda dari GCM untuk Chrome (*web browser* Goolge). Karakteristik GCM untuk Android adalah sebagai berikut:

1. GCM memperbolehkan sebuah *server* untuk mengirimkan pesan kepada perangkat berbasis Android.
2. Dengan menggunakan GCM *Cloud Connection Server*, kita dapat menerima pesan secara *upstream* dari perangkat berbasis Android.
3. GCM hanya mentransfer data mentah berupa pesan ke perangkat berbasis Android, pesan itu kemudian baru akan diproses pada perangkat berbasis Android untuk kegunaan lainnya.
4. Memerlukan perangkat berbasis Android yang mempunyai versi Android di atas versi 2.2 atau sebuah emulator Android dengan *Google* APIs.
5. GCM menggunakan koneksi yang sudah ada untuk Google *Services*. Untuk Android versi 4.0.4 ke bawah, para pengguna harus melakukan konfigurasi terhadap Google *Account* terlebih dahulu.

2.2.2 Konsep Google Cloud Messaging untuk Android

Konsep GCM secara garis besar dapat dibagi menjadi dua, yaitu komponen dan *credentials*. Komponen adalah entitas yang menjalankan tugas utama dari GCM, sedangkan *credentials* adalah IDs dan *token* yang digunakan di dalam GCM. GCM mempunyai tiga komponen utama, yaitu:

1. *Client App*

   Aplikasi yang berada pada perangkat Android dan memiliki fitur GCM di dalamnya.

2. *3rd-party Application Server*

   Sebuah *server* aplikasi milik pengembang aplikasi yang mengirimkan data yang diterima dari perangkat Android ke GCM *Connection Servers*.

3. GCM *Connection Servers*

   Server yang disediakan oleh Google untuk menerima pesan dari *server* aplikasi milik pengembang aplikasi.

   Berikut adalah daftar *credentials* yang dipakai di dalam sistem GCM :

1. *Sender* ID

   Sender ID adalah nomor proyek yang didapat dari *Goolge* API *console*.

2. *Application* ID

   Application ID adalah aplikasi Android yang telah di-*setting* pada file *application_manifest* untuk dapat menerima pesan dari GCM.

3. *Registration* ID

   Registration ID adalah identifikasi yang diberikan oleh *server* GCM kepada aplikasi. Setiap perangkat Android akan mendapatkan ID yang unik, sehingga server GCM dapat membedakan perangkat Android mana yang dituju ketika mengirim pesan.

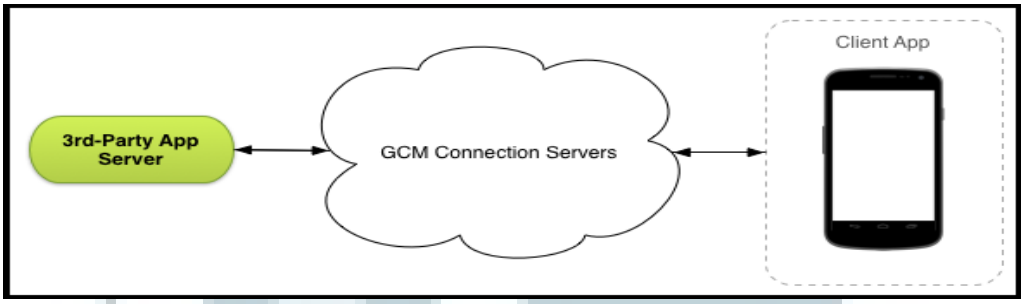
4. *Goolge User Account*

   Untuk menjalankan fitur GCM, perangkat Android yang memiliki sistem operasi Android di bawah versi 4.0.4 harus memiliki akun Google.

5. *Sender Auth Token*

   API yang ada pada 3rd-*party application server* mempunyai *token* yang berfungsi untuk membuka akses ke Google *Services.*

2.2.3 Arsitektur Google Cloud Messaging untuk Android



Gambar 2.6. Arsitektur GCM
(Sumber: Google *Cloud Messaging for* Android, 2013)

Di dalam sistem GCM, komponen-komponen yang ada berinteraksi satu sama lain sehingga pesan dapat sampai kepada tempat yang dituju. Berikut adalah penjelasan peran dari setiap komponen yang ada pada arsitektur GCM.

1. Server aplikasi merupakan server dari pengembang aplikasi yang diperbolehkan untuk mengirimkan pesan kepada GCM *Connection Servers*.
2. Google memperbolehkan GCM *Connection Servers* untuk menerima pesan dari server aplikasi. Setelah mendapatkan pesan dari server aplikasi, GCM *Connection Servers* akan mengirimkan pesan tersebut kepada aplikasi Android yang dituju.
3. Aplikasi pada Android *client* merupakan aplikasi dengan fitur GCM, dimana aplikasi tersebut mempunyai Registration ID yang unik sehingga dapat menerima pesan dari GCM *Connection Servers*.

2.3 Enkripsi

2.3.1 Definisi Enkripsi

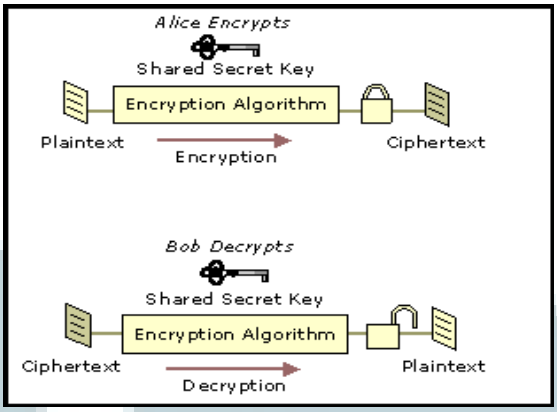
Enkripsi adalah alat yang mendasar untuk melindungi informasi yang sensitif (Abrams & Podell, 1995). Tujuan dari enkripsi adalah untuk mencegah pihak yang

tidak mempunyai akses untuk melihat atau mengubah data (Freeman, Neely, & Megalo, 1998). Pesan yang dienkripsi akan menjadi tidak dapat dibaca atau tersembunyi dan dapat dikembalikan menjadi pesan asli seperti semula dengan dekripsi. Menurut Mahapatra dan Dash (2007), sekarang ini enkripsi lebih dari sekedar enkripsi dan dekripsi. Otentikasi sekarang ini menjadi bagian dari kehidupan kita sebagai privasi dan sistem enkripsi dapat dibagi menjadi tiga dimensi yang berbeda:

1. Tipe dari operasi yang men-transformasi pesan asli menjadi pesan yang dienkripsi. Kebanyakan sistem enkripsi disebut sebagai sistem produk yang mempunyai banyak tingkatan substitusi dan transposisi.
2. Jumlah kunci enkripsi yang digunakan. Jika pengirim dan penerima menggunakan kunci yang sama, maka sistem enkripsi yang digunakan adalah *symmetric key encryption*. Tetapi apabila pengirim dan penerima menggunakan kunci yang berbda, maka yang digunakan adalah *public key encryption*.
3. Cara pesan diproses. Sebuah *block chiper process* hanya menerima satu blok inputan dan menghasilkan tepat satu blok keluaran juga. Tetapi sebuah *stream chiper process* akan memproses inputan secara kontinu dan menghasilkan keluaran berupa satu elemen tiap satu waktu pada saat proses enkripsi berlangsung.

2.3.2 Definisi Symmetric Key Encryption

Di dalam *Symmetric key encryption, encryption key* yang sama akan digunakan untuk mengenkripsi dan mengdekripsi data. *Symmetric key* dapat disebut *secret key* karena berbagi rahasia yang sama antara pengiriman dan penerima data.

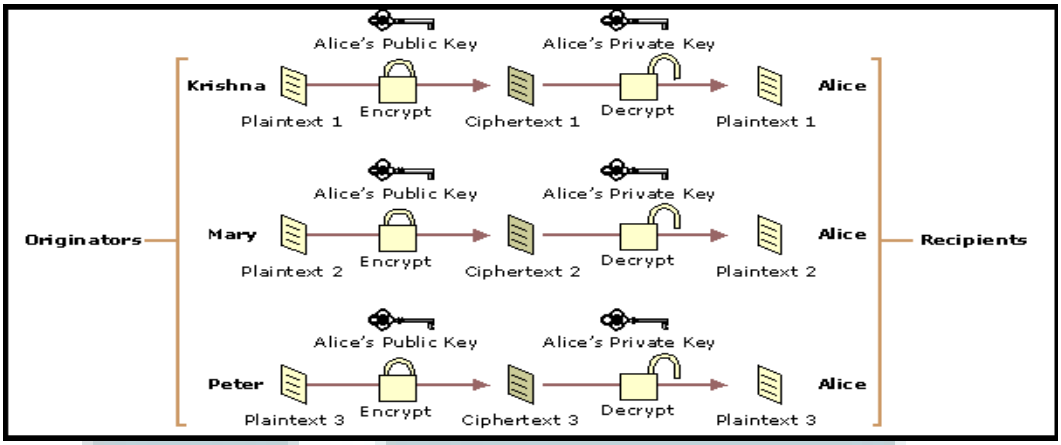


Gambar 2.7. Enkripsi dan Dekripsi Menggunakan *Symmetric Key Encryption*
(Sumber: Microsoft Technet, 2013)

Symmetric key encryption biasanya digunakan pada protokol keamanan seperti *Transport Layer Securiy* (TLS) sebagai *session key* atau pada teknologi yang menyediakan enkripsi untuk data yang *persistent* seperti surat elektronik atau *file* dokumen (Microsoft Technet, 2013).

2.3.3 Definisi Public Key Encryption

Berbeda dengan *symmetric key encryption* yang hanya menggunakan satu kunci enkripsi, *public key encryption* menggunakan dua kunci enkripsi yang berbeda dan memakai komputasi yang lebih berat , hal ini membuat enkripsi dan dekripsi yang dilakukan menjadi lebih lama (Dash & Mahapatra, 2007). *Public key encryption* memiliki *public key* dan *private key* untuk mengenkripsi dan mengdekripsi data. *Public key* berfungsi sebagai kunci untuk mengenkripsi data, sedangkan *private key* berfungsi sebagai kunci untuk mengdekripsi data. Karena mempunyai sepasang kunci untuk enkripsi dan dekripsi, maka pasangan *public key* dan *private key* tidak boleh berbeda.



Gambar 2.8 Enkripsi dan Dekripsi Menggunakan *Public Key Encryption*
(Sumber: Microsoft Technet, 2013)

Pada pemakaian *public key encryption*, sumber harus mengetahui terlebih dahulu *public key* tujuan yang ingin dituju karena dengan melakukan enkripsi dengan *public key* tujuan, data dapat di-dekripsi sesudah sampai pada tujuan dengan *private key* yang ada. *Public key encryption* dipakai untuk berbagai tujuan, misalnya untuk mengenkripsi *symmetric key* pada saat proses pengiriman, sebagai otentikasi atau sebagai *digital signatures* untuk menjaga integritas dan keaslian data (Microsoft Technet, 2013).

2.4 Secure Socket Layer

2.4.1 Definisi Secure Socket Layer

Tujuan utama dari protokol *Secure Socket Layer* (SSL) adalah untuk menyediakan privasi dan kehandalan pada dua aplikasi yang saling berkomunikasi (Kocher, Karlton, & Freier, 2011). SSL sendiri terdiri dari dua protokol utama:

1. SSL *record protocol*, protokol ini berfungsi untuk melakukan *encapsulation* terhadap protokol-protokol yang berada di *level* lebih tinggi dari SSL *record protocol*.

2. SSL *handshake protocol*, protokol ini adalah protokol yang memperbolehkan *server* dan *client* untuk melakukan otentikasi dan melakukan negosiasi. Negosiasi yang dilakukan oleh *server* dan *client* adalah menentukan algoritma enkripsi apa yang akan dipakai dan apa kuncinya sebelum *application protocol* mengirim atau menerima *byte* data.

Keuntungan dari pemakaian protokol SSL adalah, sebuah protokol dengan level yang lebih tinggi dapat berada di atas protokol SSL secara transparan. Protokol SSL mempunyai tiga sifat dasar, yaitu:

1. Koneksi bersifat pribadi atau *private*. Enkripsi dilakukan setelah sebuah *handshake* dilakukan untuk mendefinisikan sebuah *secret key*. *Symmetric crypthography* yang digunakan untuk data enkripsi antara lain adalah DES, 3DES, RC4.
2. Identifikasi rekan dapat di otentikasi dengan menggunakan *asymmetric* atau *public key crypthography* seperti RSA, DSS, dan sebagainya.
3. Koneksinya dapat diandalkan. Di dalam transportasi pesan terdapat pengecekan integritas dari pesan itu sendiri dengan menggunakan *Message Authentication Code* (MAC). Contoh *secure hash functions* yang digunakan untuk komputasi MAC adalah MD5 dan SHA.

2.5 End to End Encryption

2.5.1 Definisi End to End Encryption

End to end encryption adalah sebuah konsep yang mendeskripsikan metode untuk mengamankan data pada proses pengiriman dari satu perangkat ke perangkat lainnya, contoh yang paling sederhana adalah *Secure Socket Layer* (SSL) yang ada

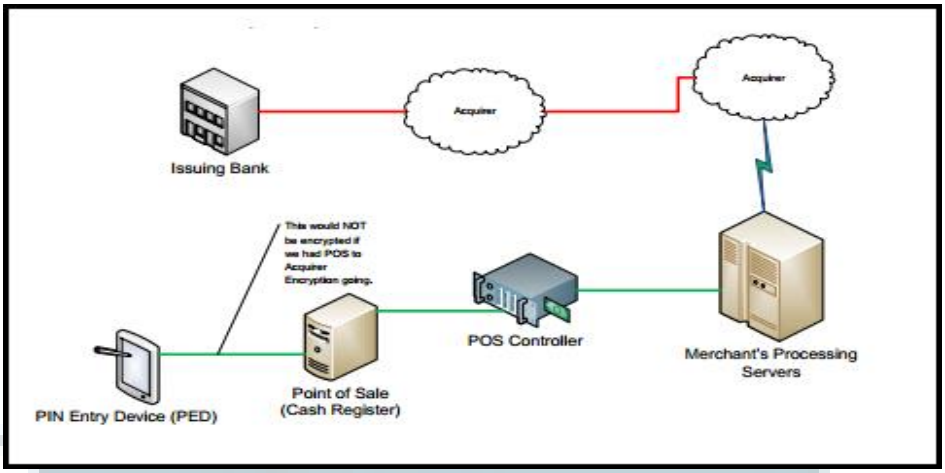
pada *web browser*. Dengan menggunakan *end to end encryption*, setidaknya menghapus kekhawatiran bahwa akan ada orang yang dapat membaca data ketika data meninggalkan sebuah *endpoint* (Williams, 2010). Jika sebuah aplikasi melakukan *end to end encryption*, maka aplikasi tersebut memerlukan pengecekan otentikasi, aplikasi tersebut juga harus dapat manangani *key management* yang diperlukan, dan data tidak boleh sampai terungkap di luar aplikasi (Saltzer, Reed, & Clark, 1984). Secara garis besar, dengan dilakukannya *end to end encryption*, maka diharapkan data yang dikirim oleh pengirim hanya dapat dibaca oleh tujuan yang sudah dikenal dan ditunjuk oleh pengirim.

2.5.2 Penerapan End to End Encryption

End to end encryption tidak hanya diterapkan pada *secure socket layer*, tetapi konsep ini juga diterapkan pada *payment card industry data security standard* (PCI-DSS) dan keperluan keamanan lainnya. Salah satu tantangan terbesar di dalam *end to end encryption* adalah istilah yang berbeda-beda dari *end to end encryption* itu sendiri. Di dalam aspek jaringan, *end to end encryption* merupakan istilah untuk mendeskripsikan proteksi data antara dua *endpoints*. Pengguna masuk ke dalam masalah ketika digunakan istilah *end to end encryption* tanpa mengerti ataupun mengetahui apa *end* atau akhiran yang akan dituju. Tanpa mendefenisikan dengan jelas antara *ends* mana enkripsi akan dilakukan, maka dampak dari mengamankan jaringan tidak akan dapat ditentukan (Williams, 2010).

Berikut adalah beberapa jenis *end to end encryption* yang pada umumnya digunakan. Contoh-contoh ini diambil dari *end to end encryption* pada PCI-DSS:

1. POS/PED *to Acquirer Encryption* (P2AE)

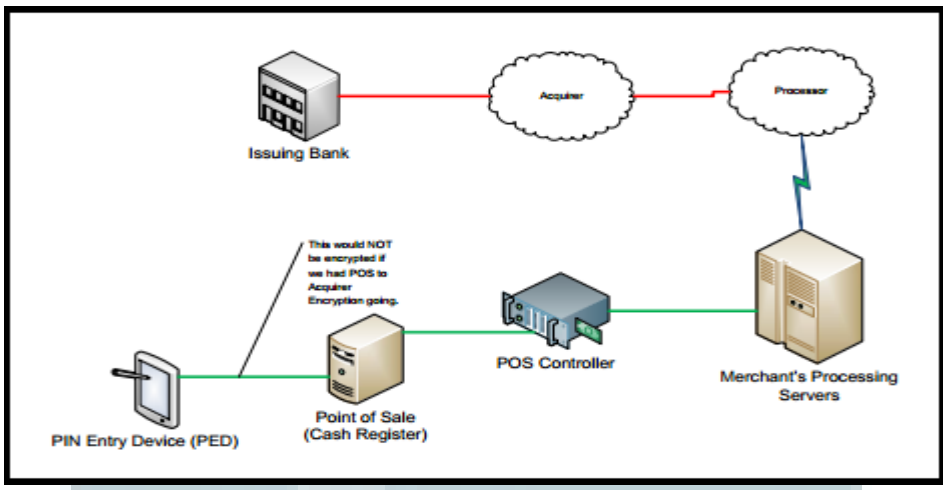


Gambar 2.9 POS/PED *to Acquirer Encryption*
(Sumber: *Will End to End Encryption Save Us All*, 2010)

P2AE adalah bentuk yang paling komplit dari *end to end encryption*. Secara teori, konsep ini memberikan rasa aman kepada pedagang ketika berurusan dengan PCI-DSS. Pada P2AE, PIN *entry device* (PED) akan mengenkripsi data pembayaran sebelum mengirimkan data tersebut ke *acquirer* untuk diproses melalui jaringan yang dimiliki oleh pedagang.

2. POS/PED *to Processor Encryption* (P2PE)

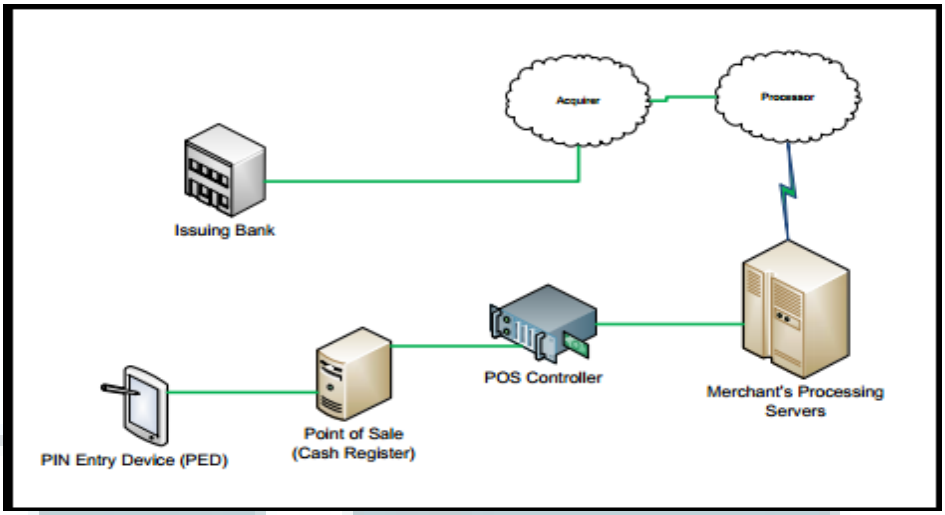
   Beberapa pedagang tidak langsung memproses transaksi mereka dengan sebuah *acquiring* bank, melainkan pedagang bekerjasama dengan *interim processing houses* atau *Independent Sales Organizations* (ISOs) yang besar. Ketika sebuah transaksi masuk ke dalam *processor*, transaksi tersebut sudah menjadi tanggung jawab pemegang *processor* untuk memastikan bahwa transaksi yang akan diteruskan ke *acquiring* bank akan aman.



Gambar 2.10 POS/PED *to Processor Encryption*
(Sumber: *Will End to End Encryption Save Us All*, 2010)

3. PED *to Issuer Encryption* (P2IE)

P2IE adalah bentuk ideal dalam mengamankan informasi ketika melakukan proses pembayaran atau transaksi. Sayangnya P2IE dapat menjadi keliru di dalam beberapa hal dan tidak akan dapat terjadi apabila tidak ada investasi yang besar dari semua pihak yang berada pada wilayah *payment data*. Pada P2IE, data transaksi akan dienkripsi sampai pada *issuer* atau bank yang bersangkutan, yang membedakan adalah setiap kali transaksi berjalan ke *provider* berikutnya, maka transksi akan di-dekripsi dan dienkripsi ulang dengan *working key* berikutnya. Dengan kata lain, setiap transaksi berada di suatu *provider*, bisa saja data transaksi tidaklah aman karena *provider* yang bersangkutan bermasalah.

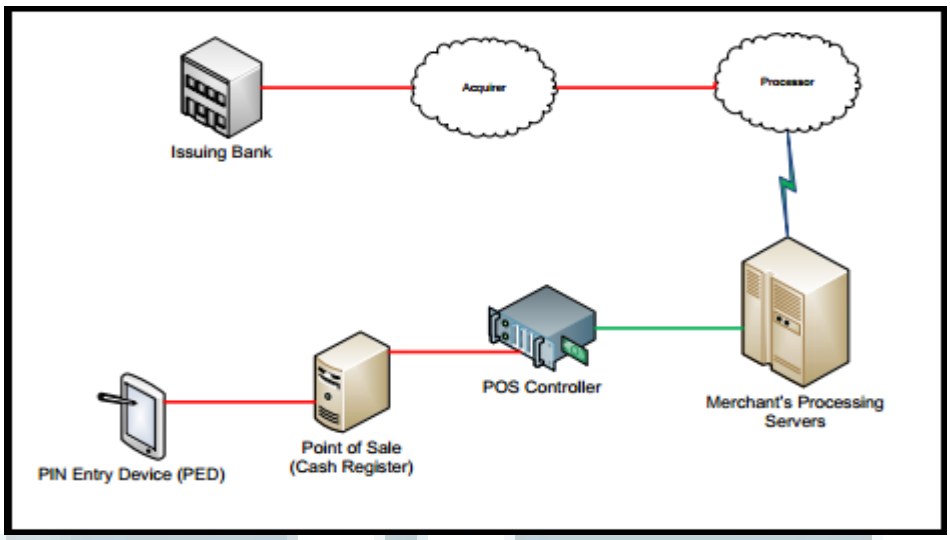


Gambar 2.11 PED *to Issuer Encryption*

(Sumber: *Will End to End Encryption Save Us All*, 2010)

4. *Controller to Processor Encryption*

   Pada *controller to processor encryption*, transaksi yang datang dari POS *controller* tidak terenkripsi. Enkripsi akan dilakukan oleh *controller* dengan melakukan sebuah *encryption routine* yang akan melindungi data sampai ke *processor*. Pengesetan seperti ini cukup efektif, tergantung dari arsitektur jaringan POS yang dimiliki oleh pedagang. Kelemahan terbesar dari *controller to processor encryption* adalah, *controller* dapat menjadi sasaran empuk penyerang yang ingin mencuri data transaksi.



Gambar 2.12 *Controller to Processor Encryption*
(Sumber: *Will End to End Encryption Save Us All*, 2010)

2.6 Algoritma Enkripsi AES

2.6.1 Definisi Algortima Enkripsi AES

Advanved Encryption Standard (AES) adalah sebuah standar enkripsi untuk data elektronik yang merupakan *symmetric encryption* dan dikeluarkan oleh National Institute of Standards and Technology (NIST) Amerika Serikat pada tahun 2001. Standard spesifikasi dari algoritma AES ini diambil dari algoritma Rijndael, sebuah algoritma yang memakai sebuah *chiper* blok yang simetris dan dapat memproses blok data sepanjang 128 *bits* dengan memanfaatkan kunci 128, 192 dan 256 *bits* (Federal Information Processing Standards Publication 197, 2001). Dari jenis-jenis kunci tersebut maka terdapat tiga jenis enkrips11i AES, yaitu "AES-128", "AES-192", dan "AES-256". Masukan dan keluaran dari algoritma AES merupakan barisan yang tersusun dari 128 *bits*. Barisan yang terbentuk biasanya disebut sebagai sebuah blok dan jumlah *bit* yang dimiliki data tersebut biasanya disebut panjang.

Karena algoritma enkripsi AES mempunyai tiga jenis kunci yang berbeda, maka spesifikasi dari ketiga jenis kunci algoritma ini juga berbeda. Untuk semua jenis kunci pada algoritma AES, panjang dari setiap block(input maupun output)-nya mempunyai panjang yang sama, yaitu 128 *bits*. Panjang *block* 128 *bits* ini direpresentasikan oleh Nb = 4, yang merefleksikan angka dari 32-*bit words*(jumlah kolom) di dalam *state*. Setiap panjang kunci yang ada direpresentasikan oleh Nk = 4, 6 atau 8 yang merefleksikan 32-*bit words* (jumlah kolom) di dalam kunci *chiper*. Sedangkan jumlah *round* yang dieksekusi tergantung kepada panjang kunci yang dipakai.

	Key Length *(Nk words)*	**Block Size** *(Nb words)*	**Number of Rounds** *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Gambar 2.13. Kombinasi *Key-Block-Round*

(Sumber: *Federal Information Processing Standards Publication* 197, 2001)

Algoritma AES menggunakan *round function* untuk enkripsi dan dekripsi. Round function ini dibentuk oleh empat jenis transformasi yang berorientasi kepada *byte*. Empat transformasi itu adalah :

1. Substitusi *byte* menggunakan tabel substitusi.
2. Penukaran baris dari *state array* dengan *offsets* yang berbeda.
3. Mencampurkan data di dalam setiap kolom dari *state array*.
4. Menambahkan *Round key* ke dalam *State*.

2.6.1 Keamanan Algoritma Enkripsi AES

Yacoumis (2005), melakukan penelitian tentang "*On The Security Of The Advanced Encryption Standard*"dengan melakukan penyerangan terhadap algoritma enkripsi AES. Di dalam penelitian ini, didapatkan bahwa serangan *cryptanalysis* paling kuat terhadap algoritma enkripsi AES adalah serangan *multiset*, tetapi algoritma enkripsi AES dapat bertahan dari serangan klasik pada *block chipers* seperti *linear* dan *differential crypanalytsis, interpolation attacks,* dan *slide attacks*. Algoritma Enkripsi AES telah diprediksi untuk tetap aman paling tidak untuk 30 Tahun. Berdasarkan riset yang telah dilakukan, AES akan tetap aman untuk 10 dari 30 Tahun yang sudah diprediksi (Yacoumis, 2005).

2.7 Algoritma Enkripsi RSA

2.7.1 Definisi Algortima Enkripsi RSA

RSA merupakan algoritma enkripsi yang termasuk dalam golongan *public key encryption*. Dirilis pada tahun 1978 oleh Ron Rivest, Adi Shamir, dan Leonard Adleman yang kemudian menamai algoritma yang mereka temukan dengan memakai inisial nama keluarga mereka, yaitu RSA. Dari jurnal yang ditulis oleh Ron Rivest, Adi Shamir, dan Leonard Adleman (1978), diketahui bahwa *public key encryption* mempunyai prosedur enkripsi E yang sesuai dengan prosedur dekripsi D dan memiliki empat sifat:

1. Setelah melakukan enkripsi, dekripsi data akan mengembalikan data seperti semula, D(E(M)) = M.
2. Dengan membalilkan prosedur enkripsi, hasil yang sama masih dapat diperoleh, E(D(M)) = M.

3. Memiliki *public key* (E*)* dan *private key* (D).
4. Dengan menyebar *public key*, tidak berarti kerahasiaan *private key* akan dapat terbuka.

Untuk menentukan *public key* dan *private key* pada algoritma enkripsi RSA diperlukan beberapa tahap, yaitu :

1. Menentukan bilangan prima yang besar, dari dua bilangan *p* dan *q* yang disebut dengan *n*. Bilangan prima ini nantinya akan ditemukan pada *public key* dan *private key* yang dibentuk. Tetapi nantinya, pada *public key* maupun *private key* tidak akan tampak secara eksplisit dari mana bilangan prima ini berasal atau terbentuk.
2. Menentukan *private key*. Pada algoritma RSA, *private key* ditentukan dengan mengambil satu angka prima yang masih berelasi dengan angka prima yang dipilih pada saat awal proses algoritma (*n*).
3. Menemukan *public key* dari *private key* dan bilangan prima yang dihitung pada saat awal proses (*n*). Pada proses ini digunakan variasi dari algoritma Euclid's untuk menghitung pembagi terbesar hingga mendapatkan *public key*.

2.7.2 Keamanan Algoritma Enkripsi RSA

Menurut Ron Rivest, Adi Shamir, dan Leonard Adleman (1978), dengan memfaktorkan *n*, *cryptanalyst* musuh dapat membongkar metode enkripsi RSA. Berikut adalah hasil percobaan yang dilakukan oleh Richard Schroeppel, Ron Rivest, Adi Shamir, dan Leonard Adleman (1978) dengan memakai algoritma pemfaktoran yang paling cepat yang diciptakan oleh Richard Schroeppel.

Tabel 2.5 Tabel Percobaan Pemfaktoran *n.*

Jumlah Digit	Jumlah Operasi	Waktu
50	1.4×10^{10}	3.9 Jam
75	9.0×10^{12}	104 Hari
100	2.3×10^{15}	74 Tahun
200	1.2×10^{23}	3.8×10^{9} Tahun
300	1.5×10^{29}	4.9×10^{15} Tahun
500	1.3×10^{30}	4.2×10^{25} Tahun

Dari hasil percobaan pada tabel 2.7, maka direkomendasikan untuk memakai panjang kunci enkripsi RSA atau *n* lebih dari 200 digit, tetapi panjang pendek kunci tetap dapat disesuaikan dengan kepentingan dari enkripsi itu sendiri.

2.8 Algoritma Message-Digest MD5

2.8.1 Definisi Algortima Message-Digest MD5

Algoritma *message-digest* MD5 didesain oleh Professor Ronald Rivest dari MIT pada Tahun 1992. Algoritma ini memproduksi suatu *hash* dengan nilai atau panjang 128-*bit* yang dikemas di dalam angka hexadesimal sepanjang 32 digit dan digunakan untuk memeriksa integritas dari data. Berikut adalah langkah-langkah yang harus dilakukan untuk mendapatkan nilai *hash* dari suatu pesan menggunakan algoritma *message-digest* MD5:

1. *Append Padding Bits*

   Pada proses ini "1" *bit* akan disambungkan ke pesan, kemudian "0" *bit* menyusul sehingga panjang di dalam *bits* dari *message* yang telah disambung menjadi sama dengan 448 atau *modulo* 512. *Padding* akan selalu dilakukan meskipun panjang pesan di dalam *bits* sudah mencapai 448 atau *modulo* 512.

2. *Append Length*

   Sebuah 64-*bit* representasi dari b (panjang dari pesan sebelum *padding bits* dilakukan) disambungkan ke hasil dari langkah sebelumnya. Jika b lebih besar dari 2^64, maka hanya 64-*bits* dari *low-order* b yang akan digunakan.

3. *Initialize MD Buffer*

   Sebuah *four-word-buffer* (A,B,C,D) digunakan untuk mengkomputasi *message digest*. *Registers* ini diinisialisasikan kepada nilai yang ada di dalam hexadesimal dengan mengutamakan *low-order bytes*.

4. *Process Message In 16-Word Blocks*

   Pertama-tama empat fungsi pembantu harus didefinisikan untuk diambil sebagai masukan tiga 32-*bit words* dan memproduksi satu 32-*bit word*. Cara pada langkah ini menggunakan sebuah tabel berisikan 64 elemen yang dikonstruksi dari fungsi sin. Biarkan T[i] men-*denote* elemen i-th dari tabel, yang sama dengan bagian integer dari 4294967296 *times* abs(sin(i)), dimana i di dalam *radians*.

5. *Output*

   Message digest yang diproduksi sebagai *output* adalah A, B, C, D. *Message digest* dimulai dari *low-order byte*, yaitu A, dan diakhiri dengan *high-order byte*, yaitu D.

2.8.2 Keamanan Algoritma Message-Digest MD5

Pada penelitian yang dilakukan oleh Md. Alam Hossain, Md. Kamrul Islam, Subrata Kumar Das, dan Md. Asif Nashiry (2012), didapatkan hasil bahwa algoritma *message-digest* MD5 dapat bertahan cukup baik dari serangan yang dilakukan. Di

dalam algoritma MD5, tidak mungkin dua pesan yang hampir sama mempunyai *hash code* yang sama. Walaupun MD5 lebih lamban dari pelopornya, yaitu MD5, tetapi MD5 lebih aman dari MD4. *Hash algorithm* seperti MD4 dan MD5 sudah banyak digunakan di dalam komersial maupun aplikasi pertahanan.