Studi Implementasi dan Perbandingan DES, TDES, AES pada J2ME

Krisantus Sembiring – NIM : 13503121

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13121@students.if.itb.ac.id

Abstrak

J2ME memiliki kelemahan dari sisi keamanan terutama pada level aplikasi. Oleh karena itu, kriptografi adalah salah satu solusi untuk mengatasi masalah tersebut. Pada makalah ini akan dijelaskan implementasi kriptografi (algoritma kunci simetri) untuk aplikasi yang berjalan pada *J2ME*. Untuk itu akan dibuat program sederhana (pocketCrypto) yang menggunakan Bouncy Castle Cryptography API untuk membandingkan performansi dari beberapa algoritma kunci simetri seperti DES, TDES dan AES untuk menyandikan data berupa teks. Implementasi ketiga algoritma ini akan dilakukan pada tiga mode operasi yaitu *cipher block chaining* (*CBC*), *cipher feedback* (*CFB*), dan *output feedback* (*OFB*). Performansi yang akan dibandingkan adalah waktu dan jumlah *resource* yang dibutuhkan untuk melakukan enkripsi dan dekripsi, dan juga tingkat keamanan dari ketiga algoritma tersebut. Program pocketCrypto kemudian dijalankan dengan menggunakan emulator Java Wireless Toolkit 2.2.

Kata kunci: J2ME, Bouncy Castle Cryptography API, Advanced Encryption Standard, Data Encryption Standard, Triple DES, CBC, CFB, OFB.

1. Pendahuluan

Seiring dengan pesatnya perkembangan teknologi *mobile,* aplikasi yang berjalan pada perangkat *mobile* seperti telepon seluler menjadi lebih kompleks dengan berbagai fitur dan layanan baru seperti *mobile banking, mobile commerce* dan lain sebagainya. Untuk aplikasi seperti ini, tentu diperlukan jaminan keamanan dan keutuhan dari data yang dikirimkan. Contohnya pada aplikasi *mobile commerce* pengguna dapat melakukan pembayaran dengan mengirimkan informasi kartu kredit melalui aplikasi yang berjalan di telepon seluler. Untuk mencuri informasi ini, seorang penyadap (*eavesdropper*) dapat menggunakan radio penerima yang dapat mengintersepsi aliran data dari telepon seluler ke BTS. Oleh karena itu, untuk melindungi informasi yang sensitif diperlukan kriptografi pada aplikasi yang berjalan pada perangkat *mobile*.

Untuk mengimplementasikan algoritma enkripsi yang aman dibutuhkan *resource* yang jumlahnya tidak sedikit. Sementara itu, pengguna aplikasi *mobile* tetap mengharapkan agar telepon seluler miliknya dapat bertahan lebih lama sehingga aplikasi *mobile* yang menggunakan kriptografi diharapkan hanya menggunakan resource dalam jumlah yang seminimum mungkin, tetapi kemanan dan keutuhan data yang dikirimkan tetap terjamin.

Salah satu *platform* yang banyak digunakan untuk membuat aplikasi *mobile* adalah J2ME (*Java 2 Platform, Micro Edition*).

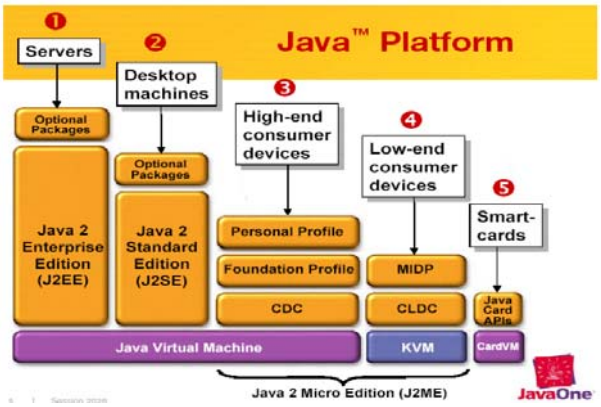
2. J2ME dan Keamanannya

J2ME merupakan edisi Java yang dibuat untuk perangkat yang mempunyai memori, CPU, dan *display* terbatas. Edisi ini merupakan salah satu dari 3 edisi java yang tersedia yaitu:

a. *Standard Edition* (J2SE)
Edisi java yang didesain untuk berjalan pada aplikasi *desktop* dan komputer-komputer *workstation.*

b. *Enterprise Edition* (J2EE)
Edisi java yang mendukung Servlet, JSP, dan XML dan ditujukan sebagai *platform* untuk aplikasi yang berjalan diserver.

c. *Micro Edition* (J2ME)



Gambar 1 Arsitektur Java

J2ME merupakan sebuah *runtime environment*. J2ME meliputi beberapa *virtual machine* yang spesifik dengan berbagai konfigurasi dan *profile* yang sesuai dengan berbagai jenis *environment* dan kebutuhan.

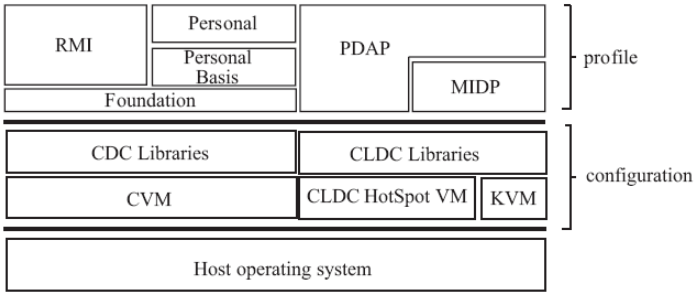
Sebuah konfigurasi mendefinisikan satu set *library* minimum dan kemampuan *virtual machine* minimum yang dimiliki sebuah *device.* Hal ini berarti *device* yang memiliki kemampuan pemrosesan yang sama dan batasan memori yang sama memiliki konfigurasi yang sama. *Profile* adalah API yang diimplementasikan si atas sebuah konfigurasi yang ditujukan untuk *device* dengan jenis/kegunaan yang serupa misalnya telepon seluler. Pada konfigurasi dan *profile* yang berbeda terdapat mekanisme keamanan yang berbeda sehingga menawarkan tingkat keamanan yang berbeda pada level aplikasi.

Terdapat dua konfigurasi pada J2ME yaitu :

a. *Connected Device Configuration* (CDC). *Device* dengan konfigurasi ini memiliki *C Virtual Machine* (CVM) yang kompatibel dengan *platform* java 2 (versi 1.3 dengan API dan kelas terbatas). Alat yang memiliki konfigurasi CDC biasanya memiliki memori minimal 2MB dan terhubung dengan jaringan tertentu.

b. *Connected Limited Device Configuration* (CLDC). *Device* yang mendukung konfigurasi ini memiliki *K Virtual Machine* (KVM) atau *HotSpot Virtual Machine* (*HotSpot* VM). KVM cocok dengan alat yang memiliki *microprocessor* 16/32-bit RISC/CISC dengan memori yang tersedia untuk *platform* java sekitar 160-512 kb. *Hotspot* VM adalah *virtual machine* yang memiliki performansi lebih baik dari KVM dan dipersiapkan untuk telepon seluler di masa yang akan datang. Alat yang mendukung konfigurasi ini biasanya memiliki hubungan ke jaringan *mobile* dengan *bandwith* terbatas.

Pembahasan mengenai penerapan kriptografi pada makalah ini akan ditekankan pada CLDC karena konfirgurasi ini memiliki *resource* yang lebih terbatas dibandingkan dengan CDC. *Profile* utama pada CLDC adalah *Mobile Information Device Profile* (MIDP) yang diperuntukkan bagi telepon seluler dan *pager* dua arah. Untuk lebih jelasnya konfigurasi dan *profile* J2ME dapat dilihat pada gambar 2.



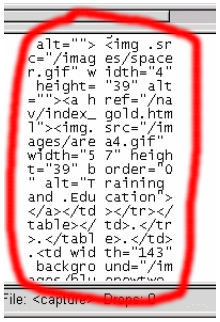
Gambar 2 Arsitektur J2ME

Alat yang mendukung *profile* MIDP biasanya terhubung dengan jaringan seperti GSM dan GPRS. Oleh karena itu, aplikasi dapat menggunakan potokol internet seperti HTTP atau WAP. Ketika aplikasi terhubung ke jaringan internet melalui protokol ini, maka semakin banyak gangguan yang dapat mengancam keamanan dan keutuhan data. Pihak lain dapat memperoleh atau bahkan mengubah data yang dikirim tanpa diketahui, dengan hanya *listening* pada data koneksi jaringan. Dengan demikian, informasi penting seperti *password* dan informasi bisnis yang penting dapat diperoleh dengan mudah. Selain itu, dapat juga terjadi *man-in-the-middle-attack*, dimana penyerang dapat mengintersepsi komunikasi antar dua pihak kemudian ”menyerupai” salah satu pihak dengan cara bersikap seolah-olah ia adalah salah satu pihak yang berkomunikasi (pihak yang lainnya tidak menyadari kalau dia berkomunikasi dengan pihak yang salah). Hal ini juga dimungkinkan karena protokol HTTP tidak

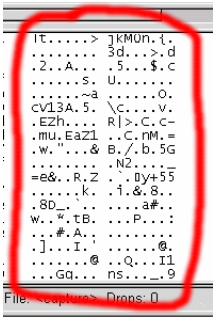
memiliki mekanisme yang dapat menjamin integritas koneksi, sehingga perubahan data yang dikirimkan mungkin terjadi.

Pada perangkat *mobile*, protokol HTTP biasanya dijalankan diatas koneksi lain seperti GSM atau GPRS. Jaringan *mobile* ini juga memiliki kelemahan yang dapat dieksploitasi oleh pihak lain. Misalnya jaringan GSM yang tidak menyediakan mekanisme kemamanan yang cukup penting seperti *mutual authentication, end-to-end security*, *non-repudiation* (penerima atau pengirim tidak bisa menyangkal bahswa dia telah mengirim atau menerima sebuah pesan) atau *user anonymity*.

Beberapa dari masalah ini sudah dapat diatasi melalui HTTPS yang tersedia pada MIDP versi 2.0. Sebelum terdapat HTTPS pada J2ME seoarang penyusup yang memiliki *laptop* dapat menggunakan aplikasi *protocol analyzer* (dapat diperoleh dengan mudah di internet) untuk memperoleh data yang lewat pada jaringan misalnya seperti pada gambar 3.



Gambar 3 Data koneksi jaringan yang tidak dienkripsi



Gambar 4 Data koneksi jaringan yang dienkripsi

Namun setelah menggunakan HTTPS penyusup akan memperoleh data seperti pada gambar 4. HTTPS merupakan protokol keamanan yang didasarkan pada koneksi. Ide dasar dari protokol ini adalah dengan mengamankan saluran komunikasi maka semua data yang mengalir melalui saluran tersebut akan aman. Akan tetapi, HTTPS tidak dapat digunakan sebagai solusi kemanan yang dapat diadaptasi untuk berbagai fungsi seperti pada J2SE, misalnya untuk model aplikasi yang dijelaskan pada [1]. Selain itu, tidak selalu aplikasi yang mengirimkan data lewat jaringan menggunakan HTTPS, misalnya menggunakan SMS.

Meskipun HTTPS dapat memberikan solusi untuk sebagian masalah kemanan data yang dikirim lewat jaringan, masih terdapat masalah keamanan data yang tersimpan pada perangkat *mobile* yang sering kali dicuri atau hilang. Oleh karena itu, pengembang aplikasi pada J2ME harus dapat mencegah akses yang tidak terotorisasi terhadap data sensitif yang tersimpan pada perangkat *mobile.* Solusi untuk masalah ini adalah mengenkripsi informasi tersebut.

Untuk mengimplementasikan sebuah mekanisme keamanan seperti enkripsi, biasanya diperlukan CPU dan *resource* memori tambahan. Menulis sendiri kode enkripsi dan dekripsi pun tidak disarankan karena terdapatnya kesulitan yang kompleks dalam memilih dan mengimplementasikan algoritma kriptografi pada J2ME. Kesulitan ini timbul karena pada J2ME tidak tersedia API untuk melakukan enkripsi seperti pada J2SE. Pada J2SE, enkripsi dapat digunakan pada aplikasi dengan memanfaatkan *Java Cryptographic Extension* (JCE). JCE menyediakan *framework* dan implementasi enkripsi, *key generation and agreement*, dan algoritma MAC. JCE mendukung algoritma enkripsi kunci simetri, asimetri, *block cipher* dan *stream cipher*. Namun, JCE tidak dimasukkan sebagai bagian dari J2ME karena selain ukurannya besar, JCE juga memerlukan memori yang tidak sedikit.

Jadi, untuk dapat menerapkan solusi keamanan pada aplikasi *mobile* pada J2ME, pengembang masih bergantung pada *toolkit* yang dikembangkan oleh vendor lain. *Toolkit* ini cukup penting karena CLDC/MIDP standar tidak menyediakan API kriptografi.

Pada CDC sebagian kecil dari paket pada JCA (*Java Cryptography Architecture*) dapat didukung secara opsional. Namun, beberapa API penting tidak tersedia sehingga beberapa algoritma tidak dapat diimplementasikan dan

kalaupun bisa, kemungkinan besar kurang efisien.

Menurut Yuan, ada beberapa hal yang harus diperhatikan terkait dengan *toolkit* yang akan digunakan pada aplikasi *mobile* untuk mengimplementasikan kriptografi yaitu:

a. Membutuhkan waktu yang relatif singkat karena perangkat *mobile* yang harus responsif walapun memiliki CPU yang lambat dan *resource* yang terbatas.
b. Membutuhkan *footprint* (media penyimpanan) yang kecil. Kebanyakan API kriptografi berukuran sampai beberapa MB, sementara perangkat *mobile* (khususnya MIDP) sebagian besar memiliki memori terbatas.
c. Mendukung berbagai algoritma penting baik algoritma simetri, algoritma kunci publik dan *digital signature*.
d. Vendor dapat dipercaya dan responsif terhadap kelemahan (*bug*).
e. Kemudahan identifikasi dan serialisasi kunci.

Salah satu API yang memenuhi beberapa faktor diatas adalah Bouncy Castle Cryptography API. *Toolkit* ini adalah sebuah API *open source* untuk mengimplementasikan kriptografi pada platform java. API ini mendukung banyak algoritma dan mengimplementasikan JCE versi 1.2.1. API ini didesain sehingga dapat dijalankan pada J2SE (minimal versi 1.4) dan pada J2ME (termasuk MIDP). API ini juga merupakan API pertama yang paling lengkap yang dapat dijalankan pada J2ME. Meskipun demikian, terdapat sebuah kekurangan yang cukup vital dari API ini yaitu kurangnya dokumentasi dan JavaDoc dari kode *source code*-nya pun tidak ditulis dengan baik.

Pad API ini didukung banyak algoritma enkripsi. Salah satu hal yang perlu diperhatikan adalah penambahan ukuran aplikasi akibat penggunaan kriptografi. Ukuran aplikasi, perlu dijaga karena pada beberapa telepon seluler jenis lama, seperti Nokia seri 40, ukuran aplikasi maksimal 100 kb. Namun, pada seri 60 ukuran aplikasi lebih fleksibel sebanyak memori yang tersisa pada telepon seluler.

3. Tipe dan Mode Algoritma Simetri

Algoritma kriptografi (*cipher*) simetri dapat dikelompokkan menjadi dua kategori, yaitu:

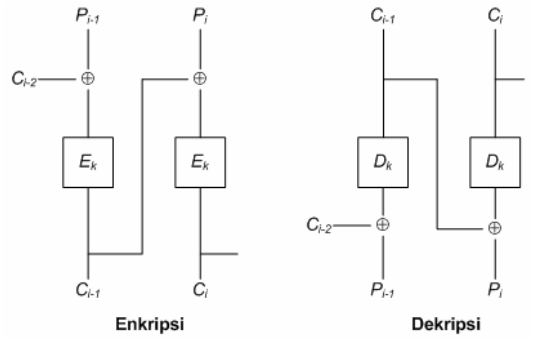
1. *Cipher* aliran (s*tream cipher*)
   Algoritma kriptografi beroperasi pada plainteks/cipherteks dalam bentuk bit tunggal, yang dalam hal ini rangkaian bit dienkripsikan/didekripsikan bit per bit.
2. *Cipher* blok (*block cipher*)
   Algoritma kriptografi beroperasi pada plainteks/cipherteks dalam bentuk blok bit, yang dalam hal ini rangkaian bit dibagi menjadi blok-blok bit yang panjangnya sudah ditentukan sebelumnya.

Baik *stream cipher* maupun *blok cipher* didukung oleh Bouncy Castle Crypto API. Namun pada makalah ini algoritma yang akan dibandingkan tergolong dalam kelompok *block cipher*. Adapun mode enkripsi dan dekripsi yang didukung pada API ini adalah sebagai berikut:

3.1 Cipher-Block-Chaining (CBC)

Mode ini menerapkan mekanisme umpan balik (*feedback*) pada sebuah blok, yang dalam hal ini hasil enkripsi blok sebelumnya diumpanbalikkan ke dalam enkripsi blok yang *current*. Caranya, blok plainteks yang *current* di-*XOR*-kan terlebih dahulu dengan blok cipherteks hasil enkripsi sebelumnya, selanjutnya hasil peng-*XOR*-an ini masuk ke dalam fungsi enkripsi. Dengan mode *CBC*, setiap blok cipherteks bergantung tidak hanya pada blok plainteksnya tetapi juga pada seluruh blok plainteks sebelumnya.

Dekripsi dilakukan dengan memasukkan blok cipherteks yang *current* ke fungsi dekripsi, kemudia meng-*XOR*-kan hasilnya dengan blok cipherteks sebelumnya. Dalam hal ini, blok cipherteks sebelumnya berfungsi sebagai umpan maju (*feedforward*) pada akhir proses dekripsi. Skema enkripsi dan dekripsi dengan mode *CBC* dapat dilihat pada Gambar 5.



Gambar 5 Mode CBC

Secara matematis, enkripsi dengan mode *CBC* dinyatakan sebagai

$C_i = E_k(P_1 \oplus C_{i-1})$

dan dekripsi sebagai

$P_i = D_k(C_i) \oplus C_{i-1}$

Yang dalam hal ini, $C_0 = IV$ (*initialization vector*). *IV* dapat diberikan oleh pengguna atau dibangkitkan secara acak oleh program. Jadi, untuk menghasilkan blok cipherteks pertama (C_1), *IV* digunakan untuk menggantikan blok cipherteks sebelumnya, C_0. Sebaliknya pada dekripsi, blok plainteks diperoleh dengan cara meng-*XOR*-kan *IV* dengan hasil dekripsi terhadap blok cipherteks pertama.

Pada mode *CBC*, blok plainteks yang sama menghasilkan blok cipherteks yang berbeda hanya jika blok-blok plainteks sebelumnya berbeda.

3.2 Cipher-FeedBack (CFB)

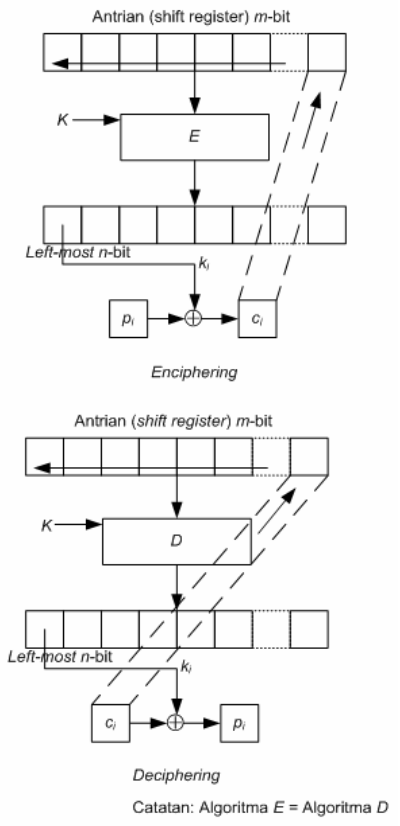
Pada mode *CFB*, data dienkripsikan dalam unit yang lebih kecil daripada ukuran blok. Unit yang dienkripsikan dapat berupa bit per bit, 2 bit, 3 bit, dan seterusnya. Bila unit yang dienkripsikan satu karakter setiap kalinya, maka mode *CFB*-nya disebut *CFB* 8-bit. Secara umum *CFB n*-bit mengenkripsi plainteks sebanyak *n* bit setiap kalinya, yang mana $n \leq m$ (m = ukuran blok). Mode *CFB* membutuhkan sebuah antrian (*queue*) yang berukuran sama dengan ukuran blok masukan.

Tinjau mode *CFB n*-bit yang bekerja pada blok berukuran *m*-bit. Algoritma enkripsi dengan mode *CFB* adalah sebagai berikut:

1. Antrian diisi dengan *IV* (*initialization vector*).
2. Enkripsikan antrian dengan kunci *K*. *n* bit paling kiri dari hasil enkripsi berlaku sebagai *keystream* (k_i) yang kemudian di-*XOR*-kan dengan *n*-bit dari plainteks menjadi *n*-bit pertama dari cipherteks. Salinan (*copy*) *n*-bit dari cipherteks ini dimasukkan ke dalam antrian (menempati *n* posisi bit paling kanan antrian), dan semua *m*-*n* bit lainnya di dalam antrian digeser ke kiri menggantikan *n* bit pertama yang sudah digunakan.
3. *m*-*n* bit plainteks berikutnya dienkripsikan dengan cara yang sama seperti pada langkah 2.

Sedangkan, algoritma dekripsi dengan mode *CFB* adalah sebagai berikut:

1. Antrian diisi dengan *IV* (*initialization vector*).
2. Dekripsikan antrian dengan kunci *K*. *n* bit paling kiri dari hasil dekripsi berlaku sebagai *keystream* (k_i) yang kemudian di-*XOR*-kan dengan *n*-bit dari cipherteks menjadi *n*-bit pertama dari plainteks. Salinan (*copy*) *n*-bit dari cipherteks dimasukkan ke dalam antrian (menempati *n* posisi bit paling kanan antrian), dan semua *m*-*n* lainnya di dalam antrian digeser ke kiri menggantikan *n* bit pertama yang sudah digunakan.
3. *m*-*n* bit cipherteks berikutnya dienkripsikan dengan cara yang sama seperti pada langkah 2.



Gambar 6 Mode *CFB* n-bit

Baik enkripsi maupun dekripsi, algoritma *E* dan *D* yang digunakan sama. Mode *CFB n*-bit yang bekerja pada blok berukuran *m*-bit dapat dilihat pada Gambar 6.

Secara formal, mode *CFB n*-bit dapat dinyatakan sebagai:

Proses Enkripsi:

$C_i = P_i \oplus MSB_m(E_k(X_i))$

$X_{i+1} = LSB_{m-n}(X_i) \,||\, C_i$

Proses Dekripsi:

$$P_i = C_i \oplus MSB_m(D_k(X_i))$$
$$X_{i+1} = LSB_{m-n}(X_i) \,||\, C_i$$

yang dalam hal ini:

X_i = isi antrian dengan X_i adalah *IV*
E = fungsi enkripsi dengan algoritma *cipher* blok
D = fungsi dekripsi dengan algoritma *cipher* blok
K = kunci
m = panjang blok enkripsi/dekripsi
n = panjang unit enkripsi/dekripsi
|| = operator penyambungan (*concatenation*)
MSB = Most Significant Byte
LSB = Least Significant Byte

Gambar 7 Enkripsi dan Dekripsi Mode *CFB* *n*-bit untuk blok *n*-bit

Jika *m* = *n*, maka mode *CFB* *n*-bit adalah seperti pada Gambar 7. *CFB* menggunakan skema umpan balik dengan mengaitkan blok plainteks bersama-sama sedemikian sehingga cipherteks bergantung pada semua blok plainteks sebelumnya. Skema enkripsi dan dekripsi dengan mode *CFB* dapat dilihat pada Gambar 7.
Dari Gambar 7 dapat dilihat bahwa:

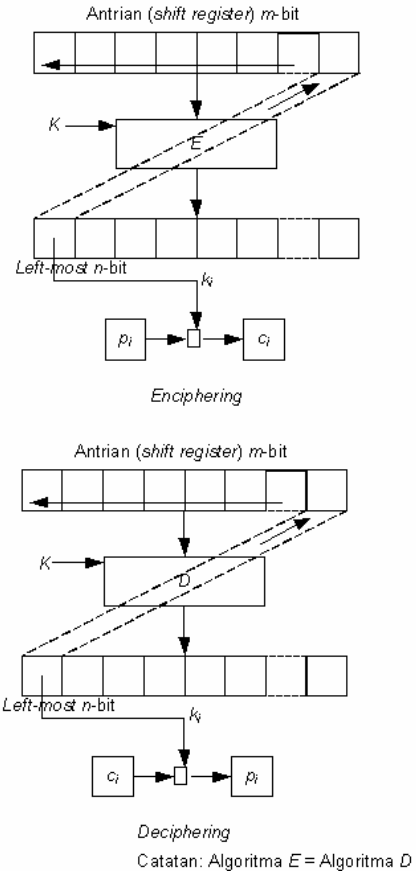
$$C_i = P_i \oplus E_k(C_{i-1})$$
$$P_i = C_i \oplus D_k(C_{i-1})$$

IV pada *CFB* tidak perlu dirahasiakan. *IV* harus unik untuk setiap pesan, sebab *IV* yang sama untuk setiap pesan yang berbeda akan menghasilkan *keystream* k_i yang sama.

3.3 Output-FeedBack (OFB)

Pada mode *OFB*, data dienkripsikan dalam unit yang lebih kecil daripada ukuran blok. Unit yang dienkripsikan dapat berupa bit per bit, 2 bit, 3 bit, dan seterusnya. Bila unit yang dienkripsikan satu karakter setiap kalinya, maka mode *OFB*-nya disebut *OFB* 8-bit. Secara umum *OFB* *n*-bit mengenkripsi plainteks sebanyak *n* bit setiap kalinya, yang mana $n \leq m$ (*m* = ukuran blok). Mode *OFB* membutuhkan sebuah antrian (*queue*) yang berukuran sama dengan ukuran blok masukan.
Tinjau mode *OFB* *n*-bit yang bekerja pada blok berukuran *m*-bit. Algoritma enkripsi dengan mode *OFB* adalah sebagai berikut (lihat Gambar 6):

1. Antrian diisi dengan *IV* (*initialization vector*).
2. Enkripsikan antrian dengan kunci *K*. *n* bit paling kiri dari hasil enkripsi dimasukkan ke dalam antrian (menempati *n* posisi bit paling kanan antrian), dan *m*-*n* bit lainnya di dalam antrian digeser ke kiri menggantikan *n* bit pertama yang sudah digunakan. *n* bit paling kiri dari hasil enkripsi juga berlaku sebagai *keystream* (k_i) yang kemudian di-*XOR*-kan dengan *n*-bit dari plainteks menjadi *n*-bit pertama dari cipherteks.
3. *m*-*n* bit plainteks berikutnya dienkripsikan dengan cara yang sama seperti pada langkah 2.



Gambar 8 Mode *OFB* n-bit

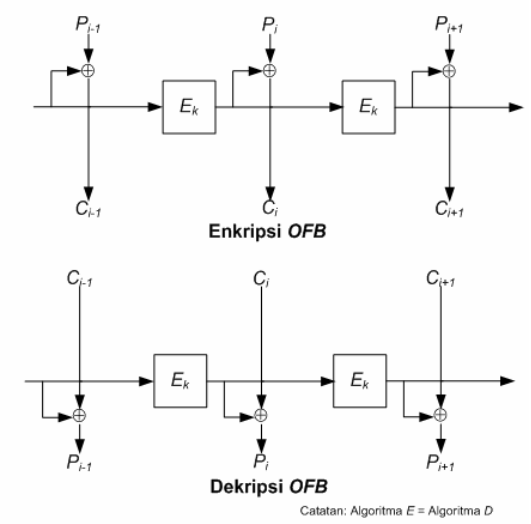
Sedangkan, algoritma dekripsi dengan mode *OFB* adalah sebagai berikut (lihat Gambar 8):

1. Antrian diisi dengan *IV* (*initialization vector*).
2. Dekripsikan antrian dengan kunci *K*. *n* bit paling kiri dari hasil dekripsi dimasukkan ke dalam antrian

(menempati *n* posisi bit paling kanan antrian), dan *m*-*n* bit lainnya di dalam antrian digeser ke kiri menggantikan *n* bit pertama yang sudah digunakan. *n* bit paling kiri dari hasil dekripsi juga berlaku sebagai *keystream* (k_i) yang kemudian di-*XOR*-kan dengan *n*-bit dari cipherteks menjadi *n*-bit pertama dari plainteks.

3. *m*-*n* bit cipherteks berikutnya dienkripsikan dengan cara yang sama seperti pada langkah 2.

Baik enkripsi maupun dekripsi, algoritma *E* dan *D* yang digunakan sama. Mode *OFB n*-bit yang bekerja pada blok berukuran *m*-bit dapat dilihat pada Gambar 8.



Gambar 9 Enkripsi dan Dekripsi *OFB n*-bit untuk blok n-bit

Secara formal, mode *OFB n*-bit dapat dinyatakan sebagai:

Proses Enkripsi:

$$C_i = P_i \oplus MSB_m(E_k(X_i))$$
$$X_{i+1} = LSB_{m-n}(X_i) \,||\, LSB_n(E_k(X_i))$$

Proses Dekripsi:

$$P_i = C_i \oplus MSB_m(D_k(X_i))$$
$$X_{i+1} = LSB_{m-n}(X_i) \,||\, LSB_n(E_k(X_i))$$

yang dalam hal ini:

X_i = isi antrian dengan X_i adalah *IV*
E = fungsi enkripsi dengan algoritma *cipher* blok
D = fungsi dekripsi dengan algoritma *cipher* blok
K = kunci
m = panjang blok enkripsi/dekripsi
n = panjang unit enkripsi/dekripsi
|| = operator penyambungan (*concatenation*)
MSB = Most Significant Byte
LSB = Least Significant Byte

OFB menggunakan skema umpan balik dengan mengaitkan blok plainteks bersama-sama sedemikian sehingga cipherteks bergantung pada semua blok plainteks sebelumnya. Skema enkripsi dan dekripsi dengan mode *OFB* dapat dilihat pada Gambar 9.

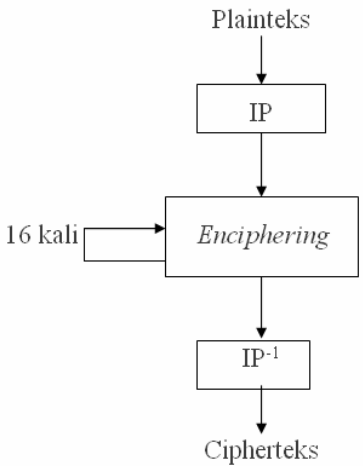
4. Algoritma Simetri

Dalam makalah ini akan dibahas implementasi dan perbandingan beberapa implementasi algoritma kunci simetri dengan menggunakan Bouncy Castle Lightweight API. Adapun, algoritma yang akan dibandingkan adalah DES, TDES dan AES.

4.1 DES (*Data **Encryption Standard***)

DES adalah salah satu algoritma cipher blok yang populer karena dijadikan standar algoritma enkripsi kunci-simetri, meski pun saat ini standard tersebut sudah diganti dengan algoritma yang baru, AES, karena DES sudah dianggap tidak aman lagi. Algoritma DES Dikembangkan di IBM pada tahun 1972 dibawah kepemimpinan W. L.Tuchman. Algorima ini didasarkan pada algoritma Lucifer yang dibuat oleh Horst Feistel. Algorima ini telah disetujui oleh National Bureau of Standard (NBS) setelah penilaian kekuatannya oleh *National Security Agency (NSA)* Amerika Serikat.

DES beroperasi pada ukuran blok 64 bit. DES mengenkripsinya 64 bit plainteks menjadi 64 bit cipherteks dengan menggunakan 56 bit kunci internal (*internal key*) atau upda-kunci(*subkey*). Kunci internal dibangkitkan dari kunci eksternal (*external key*) yang panjangnya 64 bit. Skema global dari algoritma DES dapat dilihat pada Gambar 10.



Gambar 10 Skema global DES

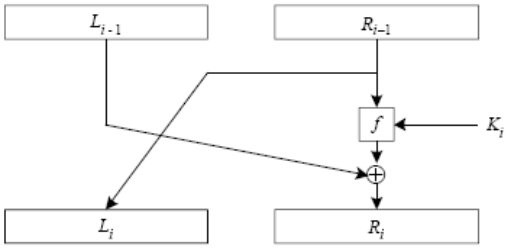
1. Blok plainteks dipermutasi dengan matriks permutasi awal (*initial permutation* atau IP).
2. Hasil permutasi awal kemudian di-*enciphering*- sebanyak 16 kali (16 putaran). Setiap putaran menggunakan kunci internal yang berbeda.
3. Hasil *enciphering* kemudian dipermutasi dengan matriks permutasi balikan (*invers initial permutation* atau IP-1) menjadi blok cipherteks.

Di dalam proses *enciphering*, blok plainteks terbagi menjadi dua bagian, kiri (*L*) dan kanan (*R*), yang masing-masing panjangnya 32 bit. Kedua bagian ini masuk ke dalam 16 putaran DES. Pada setiap putaran *i*, blok *R* merupakan masukan untuk fungsi transformasi yang disebut *f*. Pada fungsi *f*, blok *R* dikombinasikan dengan kunci internal *Ki*. Keluaran dari fungsi *f* di-*XOR*-kan dengan blok *L* untuk mendapatkan blok *R* yang baru. Sedangkan blok *L* yang baru langsung diambil dari blok *R* sebelumnya. Ini adalah satu putaran DES.

Secara matematis, satu putaran DES dinyatakan sebagai
$Li = Ri - 1$
$Ri = Li - 1$ Å $f(Ri - 1, Ki)$

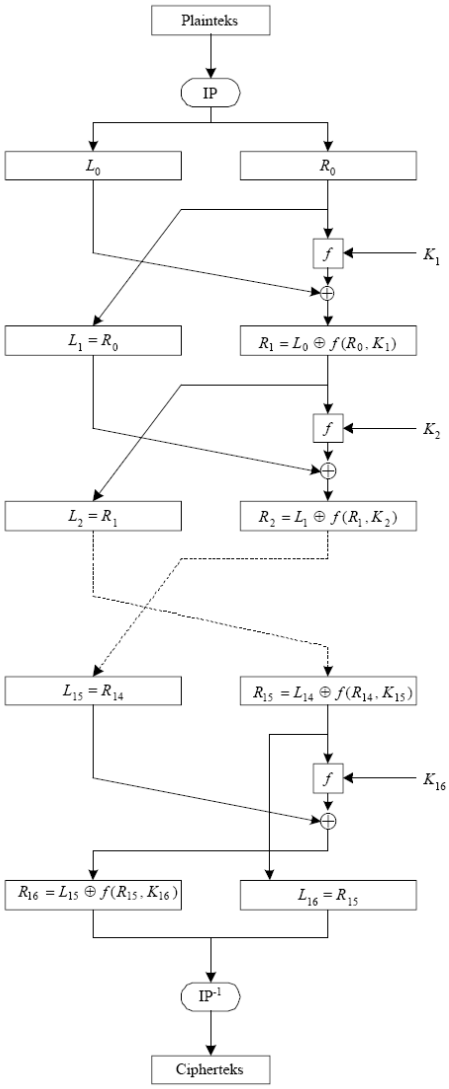
Gambar 12 memperlihatkan skema algoritma DES yang lebih rinci.

Satu putaran DES merupakan model jaringan *Feistel* (lihat Gambar 11).



Gambar 11 Jaringan Feistel untuk satu putaran DES

Perlu dicatat dari Gambar 12 bahwa jika (*L*16, *R*16) merupakan keluaran dari putaran ke-16, maka (*R*16, *L*16) merupakan pra-cipherteks (*pre-ciphertext*) dari *enciphering* ini. Cipherteks yang sebenarnya diperoleh dengan melakukan permutasi awal balikan, IP-1, terhadap blok pra-cipherteks.



Gambar 12 Algoritma Enkripsi dengan DES

DES saat ini sudah dianggap tidak aman lagi, karena panjang kuncinya yang pendek. Panjang kunci eksternal DES hanya 64 bit atau 8 karakter, itupun yang dipakai hanya 56 bit. Pada rancangan awal, panjang kunci yang diusulkan IBM adalah 128 bit, tetapi atas permintaan NSA, panjang kunci diperkecil menjadi 56 bit.

Dengan panjang kunci 56 bit akan terdapat 2^{56} atau 72.057.594.037.927.936 kemungkinan kunci. Jika serangan *exhaustive key search* dengan menggunakan prosesor paralel, maka dalam satu detik dapat dikerjakan satu juta serangan. Jadi seluruhnya diperlukan 1142 tahun untuk menemukan kunci yang benar. Namun, dari penelitian, DES sudah dapat dipecahkan.

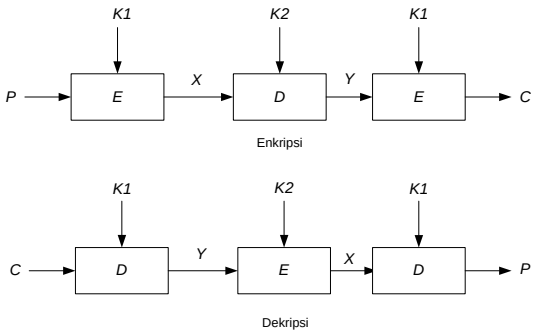
Meskipun demikian, algoritma ini masih banyak digunakan pada aplikasi J2ME karena dianggap informasi yang dienkripsi tidak sebanding nilainya dengan usaha yang diperlukan untuk memecahkannya. DES pada Bouncy Castle Cryptography API diimplementasikan pada kelas DESEngine besar filenya adalah 15 kb.

4.2 TDES (*Triple DES*)

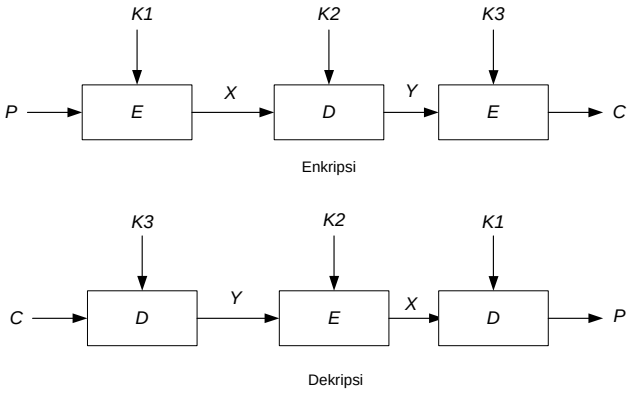
Triple DES atau TDES atau 3DES atau DESede menggunakan DES tiga kali. Penggunaan tiga langkah ini penting untuk mencegah terjadinya *meet-in-the-middle-attack* pada *double DES*. Bentuk umum TDES (mode EEE):

Enkripsi: C = EK3(EK2(EK1 (P)))
Dekripsi: P = DK1(DK2 (DK3 (C)))

Untuk menyederhanakan TDES, maka langkah di tengah diganti dengan D (mode EDE). Ada dua versi TDES dengan mode EDE, versi pertama menggunakan 2 kunci dan versi kedua menggunakan 3 kunci. Kedua mode ini didukung pada Bouncy Castle Cryptography API.



Gambar 13 *Triple* DES dengan 2 kunci



Gambar 14 Triple DES dengan 3 kunci

Triple DES dengan dua kunci memiliki panjang kunci 112 bit (2 X 56 bit) sedangkan Triple DES dengan tiga kunci memiliki panjang 168 bit.

TDES pada Bouncy Castle Cryptography API diimplementasikan pada kelas DESedeEngine dan merupakan hasil implementasi dari buku Applied Cryptography oleh Bruce Schneier. Adapun mode DES yang akan digunakan bergantung dari panjang kunci. Apabila panjang kunci lebih kecil dari 24 byte (192 bit) maka akan digunakan mode TDES dengan dua kunci, sedangkan jika panjang kunci sebesar 24 byte maka akan digunakan mode TDES dengan tiga kunci. Besar file kelas DESedeEngine ini adalah 4 kb. Namun, tetap memerlukan file DESEngine.

4.3 AES (*Advanced Encryption Standard*)

Seperti pada *DES*, *Rijndael* menggunakan substitusi dan permutasi, dan sejumlah putaran. Untuk setiap putarannya, *Rijndael* menggunakan kunci yang berbeda. Kunci setiap putaran disebut *round key*. Tetapi tidak seperti *DES* yang berorientasi bit, *Rijndael* beroperasi dalam orientasi *byte* sehingga memungkinkan untuk implementasi algoritma yang efisien ke dalam *software* dan *hardware* [1].

Garis besar algoritma *Rijndael* yang beroperasi blok 128-bit dengan kunci 128-bit adalah sebagai berikut:

1. *AddRoundKey*: melakukan *XOR* antara *state* awal (plainteks) dengan *cipher key*. Tahap ini disebut juga *initial round*.
2. Putaran sebanyak *Nr* – 1 kali. Proses yang dilakukan pada setiap putaran adalah:
   a. *ByteSub*: substitusi byte dengan menggunakan tabel substitusi (*S-box*).

Tabel substitusi dapat dilihat pada tabel 2, sedangkan ilustrasi *ByteSub* dapat dilihat pada gambar 16.

b. *ShiftRow:* pergeseran baris-baris *array state* secara *wrapping*. Ilustarsi *ShiftRow* dapat dilihat pada gambar 17.

c. *MixColumn*: mengacak data di masing-masing kolom *array state*. Ilustarsi *MixColumn* dapat dilihat pada gambar 18.

d. *AddRoundKey:* melakukan *XOR* antara *state* sekarang dengan *round key*. Ilustarsi *AddRoundKey* dapat dilihat pada gambar 19.

3. *Final round*: proses untuk putaran terakhir:
   a. *ByteSub*.
   b. *ShiftRow*.
   c. *AddRoundKey*.

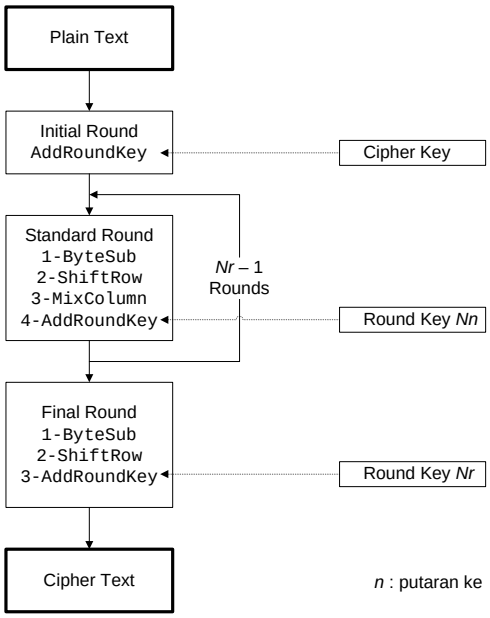
Diagram proses enkripsi *AES* dapat dilihat pada Gambar 15.

Algoritma *Rijndael* mempunyai 3 parameter sebagai berikut:

1. `plainteks` : *array* yang berukuran 16 *byte*, yang berisi data masukan.
2. `cipherteks` : *array* yang berukuran 16 *byte*, yang berisi hasil enkripsi.
3. `key` : *array* yang berukuran 16 *byte*, yang berisi kunci ciphering (disebut juga *cipher key*).

Dengan 16 *byte*, maka baik blok data dan kunci yang berukuran 128-bit dapat disimpan di dalam ketiga array tersebut (128 = 16 x 8).

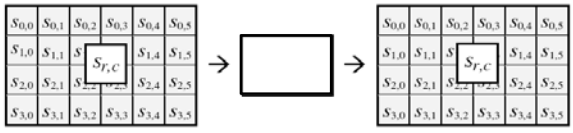
Selama kalkulasi plainteks menjadi cipherteks, status sekarang dari data disimpan di dalam *array of byte* dua dimensi, `state`, yang berukuran `NROWS x NCOLS`. Elemen *array* `state` diacu sebagai $S[r,c]$, dengan $0 \le r < 4$ dan $0 \le c < Nc$ (*Nc* adalah panjang blok dibagi 32). Pada *AES*, $Nc = 128/32 = 4$.



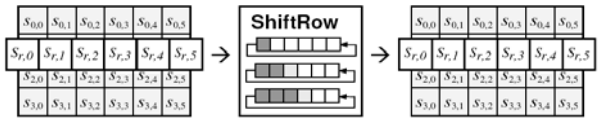
Gambar 15 Diagram Proses Enkripsi *AES*

Tabel 1 Tabel *S-box* yang digunakan dalam transformasi *ByteSub() AES*

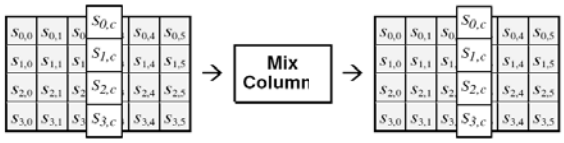
hex	y=0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x=0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16



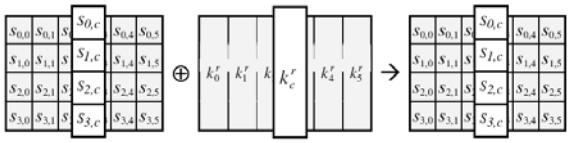
Gambar 16 Ilustrasi Transformasi *ByteSub() AES*



Gambar 17 Ilustrasi Transformasi *ShiftRow() AES*



Gambar 18 Ilustrasi Transformasi *MixColumn() AES*



Gambar 19 Ilustrasi Transformasi *AddRoundKey() AES*

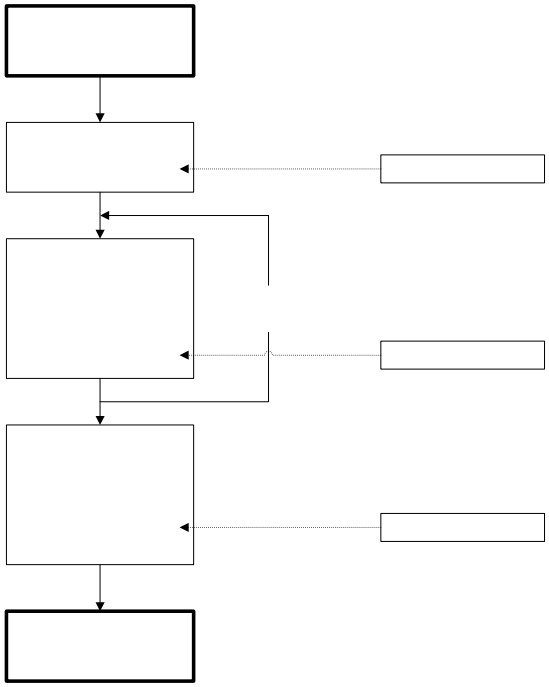
Cipher kebalikan merupakan algoritma kriptografi *AES* yang digunakan untuk melakukan proses dekripsi cipherteks menjadi plainteksnya. Secara garis besar, *cipher* kebalikan yang beroperasi blok 128-bit dengan kunci 128-bit adalah sebagai berikut:

1. *AddRoundKey*: melakukan *XOR* antara *state* awal (cipherteks) dengan *cipher key*. Tahap ini disebut juga *initial round*.
2. Putaran sebanyak *Nr* – 1 kali. Proses yang dilakukan pada setiap putaran adalah:
    a. *InvShiftRow:* pergeseran baris-baris *array state* secara *wrapping*.
    b. *InvByteSub*: substitusi byte dengan menggunakan tabel substitusi kebalikan (*inverse S-box*). Tabel substitusi dapat dilihat pada tabel 3.
    c. *AddRoundKey:* melakukan *XOR* antara *state* sekarang dengan *round key*.
    d. *InvMixColumn*: mengacak data di masing-masing kolom *array state*.
3. *Final round*: proses untuk putaran terakhir:
    a. *InvShiftRow*.
    b. *InvByteSub*.
    c. *AddRoundKey*.

Tabel 2 Tabel *S-box* yang digunakan dalam transformasi *InvByteSub() AES*

hex	y 0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x 0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Diagram proses dekripsi *AES* dapat dilihat pada Gambar 20.



Gambar 20 Diagram Proses Dekripsi *AES*

AES pada Bouncy Castle Cryptography API diimplementasikan sebagai hasil optimasi dari AES dalam bahasa C pada *paper* yang ditulis oleh Dr. Brian Gladman. Terdapat tigas kelas hasil implementasi AES. Ketiganya dibuat untuk menyeimbangkan antara kecepatan dan jumlah memori yang digunakan. Ketiga kelas ini adalah:

1. AESFastEngine, mengunakan tabel statik (berukuran 8 kb) yang berisi hasil perhitungan *round*, 4 buah tabel buffer untuk enkripsi dan 4 buah lagi untuk dekripsi (masing-masing berukuran 256 word). Dengan demikian enkripsi dan dekripsi dapat dilakukan dalam waktu yang

lebih cepat dibandingkan dengan dua kelas lainnya. Kelas AESFastEngine ini besar filenya adalah 46 kb.

2. AESEngine, menggunakan satu buah tabel untuk enkripsi dan satu buah tabel buffer untuk dekripsi (masing-masing berukuran 256 *word*). Kelas AESEngine ini besar filenya adalah sebesar 27 kb.

3. AESLightEngine, merupakan implementasi AES yang membutuhkan waktu paling lama karena tidak ada tabel statik yang disimpan, sehinnga semua nilai dihitung pada setiap *round*. Kelas AESLightEngine ini besar filenya adalah 20 kb.

Untuk pengujian pada makalah ini akan digunakan kelas AESEngine.

4. Percobaan dan Hasil

Perbandingan performansi algoritma DES, TDES dan AES pada J2ME dilakukan dengan cara membuat program sederhana yang melakukan enkripsi dan dekripsi pesan teks dengan panjang yang bervariasi. Kakas yang digunakan untuk menjalan progran sederhana ini adalah emulator Java Wireless Toolkit 2.2 yang dijalankan pada komputer dengan prosesor Pentium(R) 4 CPU 2,26 GHz, RAM 256 MB. Perbandingan ketiga algoritma di atas dilakukan terhadap 3 mode yaitu CBC, CFB (8 bit), dan OFB (8 bit). Ketiga algoritma dijalankan secara terpisah dan dalam sekali eksekusi program digunakan sebuah mode

Dengan menggunakan emulator ini dapat disimulasikan *device* dengan jumlah *run time* memori (heap), jumlah *storage*, dan dengan kecepatan eksekusi *virtual machine* tertentu. Jadi, kakas ini dapat digunakan untuk mengetahui performansi ketiga algoritma tersebut pada *device* dengan spesifikasi yang berbeda-beda.

Untuk mengetahui penggunaan *resource* (memori) untuk melakukan enkripsi dan dekripsi digunakan kelas `Runtime`, sedgkan untuk menghitung jumlah total waktu yang diperlukan digunakan kelas `System`. Untuk lebih jelasnya dapat dilihat pada bagian lampiran.

Adapun mekanisme dan hasil pengujian yang dilakukan adalah sebagai berkut.

Tabel 3 Spesifikasi kasus uji 1

Kecepatan VM	500 bytecodes/ms
Storage	1024 kb
Heap	512 kb

Tabel 4 Hasil pengujian kasus uji 1

Algoritma/ Mode	Enkripsi	Dekripsi
	Waktu (ms) Memori (byte)	Waktu (ms) Memori (byte)
Plainteks =160 karakter		
DES/ CBC	180 19586	165 1588
DES/ CFB	956 20559	1013 1565
DES/OFB	859 21000	825 1565
TDES/CBC	406 20184	375 1744
TDES/CFB	2203 20204	2343 1744
TDES/OFB	2187 23784	2171 1744
AES/CBC	297 21872	344 2284
AES/CFB	4609 21808	4390 2284
AES/OFB	4391 23812	4687 2284
Plainteks = 500 karakter		
DES/ CBC	356 2000	340 2100
DES/CFB	2676 2010	2550 2100
DEC/OFB	2560 2056	2550 2100
TDES/CBC	1172	906

	2604	3464
TDES/CFB	6594 2684	7203 3456
TDES/OFB	6516 2676	6547 3456
AES/CBC	937 3316	1172 4044
AESCFB	13688 3304	13688 3996
AESOFB	14141 3300	13828 3996
Plainteks = 1000 karakter		
DES/ CBC	856 3200	876 3100
DES/CFB	4200 3298	4168 3456
DES/OFB	3996 3300	3998 3245
TDES/CBC	1750 3604	1907 5984
TDES/CFB	13407 3680	13188 5964
TDES/OFB	12968 3676	12985 3676
AES/CBC	1844 4308	2156 4404
AES/CFB	27328 4304	27657 6504
AES/OFB	27593 4296	27703 6504

5. Kesimpulan

Dari hasil studi dan percobaan maka dapat ditarik kesimpilan sebagai berikut:

a. Aplikasi J2ME yang melibatkan data sensitif baik yang dikirim melalui jaringan maupun yang disimpan pada *device* memerlukan kriptografi untuk menjadmin kemanan dan keutuhan data dari pihak lain.

b. Sulit untuk mengimplementasikan sendiri kode algoritma enkripsi pada J2ME standar karena tidak ada API yang tersedia. Selain itu, *resource* pada alat yang mendukung J2ME sangat terbatas. Oleh karena itu pengembang dapat memanfaatkan *toolkit* kriptografi yang dikembangkan vendor lain seperti Bouncy Castle Cryptography API

c. Berdasarkan panjang kunci maka algoritma AES dan TDES jauh lebih aman daripada algoritma DES

d. Implementasi DES, TDES dan AES pada Bouncy Castle Cryptography API dapat digunakan pada J2ME karena membutuhkan waktu dan *resource* yang relatif kecil.

e. Terdapat tiga pilihan implementasi AES pada aplikasi J2ME dengan menggunakan Bouncy Castle Cryptography API yang masing-masing menggunakan *resource* dan CPU yang berbeda. Oleh karena itu, pemilihan kelas yang akan digunakan dapat disesuaikan dengan spesifikasi *device* tempat aplikasi akan dijalankan.

f. Kesalahan 1-bit pada blok plainteks/cipherteks dengan mode operasi *CFB* akan merambat pada blok-blok plainteks/cipherteks yang berkoresponden dan blok-blok plainteks/cipherteks selanjutnya pada proses enkripsi/dekripsi.

g. Urutan tingkat keamanan data algoritma kriptografi *AES* dengan mode operasi *CBC*, *CFB* 8-bit, dan *OFB* 8-bit terhadap pengubahan satu bit atau lebih blok cipherteks, penambahan blok cipherteks semu, dan penghilangan satu atau lebih blok cipherteks secara berturut-turut mulai dari yang teraman adalah sebagai berikut:
OFB 8-bit, *CBC*, *CFB* 8-bit

h. Perbandingan jumlah waktu yang dibutuhkan untuk enkripsi dan dekripsi dengan algoritma DES, TDES, AES pada mode CBC, CFB dan OFB adalah sebagai berikut:

Tabel 5 Perbandingan waktu

Algoritma	Total waktu
DES	CBC<OFB<CFB
TDES	CBC<OFB<CFB
AES	CBC<OFB~CFB

i. Waktu yang dibutuhkan algoritma DES, TDES dan AES untuk melakukan enkripsi hampir sama dengan waktu yang dibutuhkan untuk melakukan dekripsi. Sedangkan perbandingan resource dari hasil percobaan tidak dapat dibandingkan karena enkripsi dan dekripsi dijalankan secara sekuensial pada satu kali eksekusi program.

j. Perbandingan jumlah *resource* yang dibutuhkan untuk enkripsi dan dekripsi dengan algoritma DES, TDES, AES pada mode CBC, CFB dan OFB adalah sebagai berikut:

Tabel 6 Perbandingan jumlah *resource*

Algoritma	Jumlah *Resource*
DES	CBC<OFB~CFB
TDES	CBC<OFB~CFB
AES	CBC<OFB~CFB

6. Daftar Pustaka

[1] Bouncy Castle Crypto Package v 1.34 http://www.bouncycastle.org/ Diakses tanggal 5 Oktober 2006.

[2] Cervera, Anders.2002. Analysis of J2ME™ for developing Mobile Payment Systems. Master Thesis, IT University Of Copenhagen.

[3] Debabbi, Mourad, et al. Security Analysis of Wireless Java. Concordia Institute for Information Systems Engineering, Concordia University.

[4] Knudsen, Jonathan. 2002. MIDP Application Security 1: Design Concerns and Cryptography. *http://developers.sun.com/techtopics/mobility/midp/articles/security1/*. Diakses tanggal 1 September 2006

[5] Knudsen, Jonathan. 2002. MIDP Application Security 3: Authentication in MIDP. *http://developers.sun.com/techtopics/mobility/midp/articles/security3/*. Diakses tanggal 1 September 2006

[6] Knudsen, Jonathan. 2002. MIDP Application Security 4: Encryption in MIDP. *http://developers.sun.com/techtopics/mobility/midp/articles/security4/*. Diakses tanggal 1 September 2006

[7] Knudsen, Jonathan. 2002. Wireless Java-Chapter 12. *http://developers.sun.com/techtopics/mobility/midp/chapters/j2meknudsen/Chap12.pdf*. Diakses tanggal 1 September 2006.

[8] Kolski, Otto et al. 2004. MIDP 2.0 Security Enhancements. IEEE-*Proceedings of the 37th Hawaii International Conference on System Sciences* .

[9] Li, Johnny et al. Component-based Interchangeable Cryptographic Architecture for SecuringWireless Connectivity in Java™ Applications. Department of Computer Science. University of Pretoria

[10] Matsuoka, Yusuke, et al. Java Cryptography on KVM and its Performance and Security Optimization using HW/SW Co-design Techniques. 2004. Electrical Engineering Department, University of California.

[11] Munir, Rinaldi. 2006. Diktat Kuliah IF5054 Kriptografi. Program Studi Teknik Informatika, STEI, Institut Teknologi Bandung.

http://developers.sun.com/techtopics/mobility/midp/chapters/j2meknudsen/Chap12.pdf Diakses tanggal 1 Oktober 2006.s

[12] Yuan, Michael.2002**.** Data Security in Mobile Java. *http://www.javaworld.com/javaworld/jw-12-2002/jw-1220-wireless.html?page=1* Diakses tanggal 1 September 2006

7. Lampiran Source Code

```
import java.io.*;
import java.lang.*;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import org.bouncycastle.util.test.*;
import org.bouncycastle.util.encoders.*;
import org.bouncycastle.crypto.*;
import org.bouncycastle.crypto.paddings.*;
import org.bouncycastle.crypto.engines.*;
import org.bouncycastle.crypto.modes.*;
import org.bouncycastle.crypto.params.*;
public class PocketCrypto extends MIDlet
{
    private Display             d           = null;
    private boolean             doneEncrypt = false;
    private String              key         = "01234567";
    private String              plainText   = "";
    private byte[]              keyBytes    = null;
    private byte[]              cipherText  = null;
    private BufferedBlockCipher cipher      = null;
    private String[]            cipherNames = {"DES", "DESede", "IDEA", "Rijndael", "Twofish"};
    private Form                output      = null;
    private int selCipher=1;
    public void startApp()
    {
        Display.getDisplay(this).setCurrent(output);
    }

    public void pauseApp() { }

    public void destroyApp(boolean unconditional) {  }

    public PocketCrypto()
    {
        selCipher=0; //DES
        //selCipher=1; //TDES
        //selCipher=2; //AES
        output = new Form("BouncyCastle");
    //160 karakter
```

```
        plainText= "Mode ini menerapkan mekanisme umpan balik (feedback) pada sebuah blok, yang dalam hal ini hasil enkripsi
blok sebelumnya diumpanbalikkan ke dalam enkripsi blok";
        compare();

        //500 karakter
        plainText= "Mode ini menerapkan mekanisme umpan balik (feedback) pada sebuah blok, yang dalam hal ini hasil enkripsi
blok sebelumnya diumpanbalikkan ke dalam enkripsi blok yang current. Caranya, blok plainteks yang current di-XOR-kan terlebih
dahulu dengan blok cipherteks hasil enkripsi sebelumnya, selanjutnya hasil peng-XOR-an ini masuk ke dalam fungsi enkripsi.
Dengan mode CBC, setiap blok cipherteks bergantung tidak hanya pada blok plainteksnya tetapi juga pada seluruh blok plainteks
sebelumnya. Dekripsi ";
        compare();

        //1003 karakter
        plainText="Mode ini menerapkan mekanisme umpan balik (feedback) pada sebuah blok, yang dalam hal ini hasil enkripsi blo
sebelumnya diumpanbalikkan ke dalam enkripsi blok yang current. Caranya, blok plainteks yang current di-XOR-kan terlebih dahulu
dengan blok cipherteks hasil enkripsi sebelumnya, selanjutnya hasil peng-XOR-an ini masuk ke dalam fungsi enkripsi. Dengan mode
CBC, setiap blok cipherteks bergantung tidak hanya pada blok plainteksnya tetapi juga pada seluruh blok plainteks sebelumnya.
Dekripsi dilakukan dengan memasukkan blok cipherteks yang current ke fungsi dekripsi, kemudia meng-XOR-kan hasilnya dengan blok
cipherteks sebelumnya. Dalam hal ini, blok cipherteks sebelumnya berfungsi sebagai umpan maju (feedforward) pada akhir proses
dekripsi. Skema enkripsi dan dekripsi dengan mode CBC dapat dilihat pada Gambar 3. Secara matematis, enkripsi dengan mode CBC
dinyatakan sebagai   Ci = Ek(P1 ??Ci??) dan dekripsi sebagai  Pi = Dk(Ci)  ? Ci-1 Yang dalam hal ini, CO = IV (initialization
vector)";
        compare();


    }

    private void compare(){
        long d1,d2,e1,e2;
        output.append("Key: " + key.substring(0, 7) + "...\n");
        output.append("In : " + plainText.substring(0, 7) + "...\n");
        message("Plain text "+plainText.length()+"karakter");
        Runtime r=Runtime.getRuntime();
        //for(selCipher=1;selCipher<3;selCipher++){
        //ENKRIPSI
        final long enc=System.currentTimeMillis();
        e1=r.freeMemory();
        cipherText = performEncrypt(Hex.decode(key.getBytes()), plainText);
        e2=r.freeMemory();
        message((System.currentTimeMillis() - enc)+ " encryption elapsed time \n "+(e1-e2)+" memory used");
        String ctS = new String(Hex.encode(cipherText));
        output.append("\nCT : " + ctS.substring(0, 7) + "...\n");

        //DEKRIPSI
        final long dec=System.currentTimeMillis();
        d1=r.freeMemory();
        String decryptText = performDecrypt(Hex.decode(key.getBytes()), cipherText);
        d2=r.freeMemory();
        message((System.currentTimeMillis() - dec)+ " decryption elapsed time\n "+(d1-d2)+" memory used");
        output.append("PT : " + decryptText.substring(0, 7) + "...\n");
            System.gc();
        //}

    }
    private final byte[] performEncrypt(byte[] key, String plainText)
    {
        byte[] ptBytes = plainText.getBytes();

        cipher = new PaddedBufferedBlockCipher(new CBCBlockCipher(getEngineInstance()));
        //cipher = new PaddedBufferedBlockCipher(new CFBBlockCipher(getEngineInstance(),8));
        //cipher = new PaddedBufferedBlockCipher(new OFBBlockCipher(getEngineInstance(),8));
        String name = cipher.getUnderlyingCipher().getAlgorithmName();
        message("Using " + name);

        cipher.init(true, new KeyParameter(key));

        byte[] rv = new byte[cipher.getOutputSize(ptBytes.length)];

        int oLen = cipher.processBytes(ptBytes, 0, ptBytes.length, rv, 0);
        try
        {
            cipher.doFinal(rv, oLen);
        }
        catch (CryptoException ce)
        {
            message("Ooops, encrypt exception");
            status(ce.toString());
        }
```

```
        return rv;
    }

    private final String performDecrypt(byte[] key, byte[] cipherText)
    {
        cipher.init(false, new KeyParameter(key));
        byte[] rv = new byte[cipher.getOutputSize(cipherText.length)];
        int oLen = cipher.processBytes(cipherText, 0, cipherText.length, rv, 0);
        try
        {
            cipher.doFinal(rv, oLen);
        }
        catch (CryptoException ce)
        {
            message("Ooops, decrypt exception");
            status(ce.toString());
        }
        return new String(rv).trim();
    }

    private final BlockCipher getEngineInstance()
    {
        BlockCipher rv = null;
        switch (selCipher)
        {
            case 0 :
                //key="01234567";
                rv = new DESEngine();
                break;
            case 1 :
                //key="0123456789abcdef0123456789abcdef";
                rv = new DESedeEngine();
                break;
            case 3 :
               // key="01234567";
                rv = new IDEAEngine();
                break;
            case 2 :
                //key="0123456789abcdef0123456789abcdef";
                rv = new RijndaelEngine();
                break;
             case 4 :
                //key="01234567";
                rv = new TwofishEngine();
                 break;
             default :
                 //key="01234567";
                 rv = new DESEngine();
                 break;
        }
        return rv;
    }

    public void message(String s)
    {
        System.out.println("M:" + s);
    }

    public void status(String s)
    {
        System.out.println("S:" + s);
    }

}
```