暗号化/復号化 - JNI 呼び出しによる OpenSSL* API の使用

この記事は、インテル® デベロッパー・ゾーンに公開されている「Encryption/Decryption - invoking OpenSSL API through JNI calls」の日本語参考訳です。

ここでは、OpenSSL* ライブラリーを使用して Android* アプリケーションにインテル® AES-NI 命令を統合する手順を紹介します。手順に従うことで、インテル® AES-NI 命令によるアクセラレーションを利用する JNI アプリケーションをビルドできます。

インテル® AES New Instructions (インテル® AES-NI)

2008 年 3 月に発表されたインテル® AES-NI は、インテル® マイクロプロセッサー向けの x86 命令セット・アーキテクチャーの拡張です。この命令セットは、Advanced Encryption Standard (AES) を使用して暗号化/復号化を行うアプリケーションのパフォーマンス、セキュリティー、電力効率を向上することを目的としています。

Android* でインテル® AES-NI を使用する

OpenSSL* ライブラリーの AES アルゴリズムは、ネイティブ Java* プロバイダーと比べてパフォーマンスを大幅に向上します。これは、OpenSSL* ライブラリーはインテル® プロセッサー向けに最適化されており、インテル® AES-NI 命令を使用するためです。以下に、OpenSSL* プロバイダーを使用してファイルを暗号化する手順を示します。

Android* 4.3 から、Android* オープンソース・プロジェクト (AOSP) の OpenSSL* はインテル® AES-NI をサポートするようになったため、適切な設定でコンパイルするだけで済みます。また、公式 Web サイトからソースをダウンロードしてコンパイルすると、プロジェクト・ディレクトリーで ***.a/*.so** を利用できるようになります。暗号化ライブラリーを入手する方法は 2 つあります。

AOSP ソースを保持している場合は、http://www.openssl.org/source/ (英語) から OpenSSL* をダウンロードできます。最新のバージョンを使用することで、openssl の以前のバージョンで見つかった脆弱性を回避することができます。AOSP には openssl ライブラリーが含まれており、このライブラリーをアプリケーションの jni フォルダーに配置することで、インクルード・ディレクトリーにアクセスできるようになります。

openssl ソースをダウンロードしてクロスコンパイルし、ライブラリーを作成する場合は、次の操作を行います。

1. ソースコードをダウンロードします。
   ```
   wget https://www.openssl.org/source/openssl-1.0.1j.tar.gz
   ```

2. コンソールで次のコマンドを実行して、コンパイルします (NDK 変数は、使用しているディストリビューションのフルパスに設定します)。
   ```
   export NDK=~/android-ndk-r9d
   export TOOL=arm-linux-androideabi
   export NDK_TOOLCHAIN_BASENAME=${TOOLCHAIN_PATH}/${TOOL}
   export CC=$NDK_TOOLCHAIN_BASE-gcc
   export CXX=$NDK_TOOLCHAIN_BASENAME-g++
   export LINK=${CXX}
   export LD=$NDK_TOOLCHAIN_BASENAME-ld
   export AR=$NDK_TOOLCHAIN_BASENAME-ar
   export STRIP=$NDK_TOOLCHAIN_BASENAME-strip
   export ARCH_FLAGS="-march=armv7-a –mfloat-abi=softfp –mfpu=vfpv3-d16"
   export ARCH_LINK="-march=armv7-a –Wl, --flx-cortex-a"
   export CPPFLAGS="${ARCH_FLAGS} –fpic –ffunction-sections –funwind-tables –fstack-
        protector –fno-strict-aliasing –finline-limited=64"
   export LDFLAGS="${ARCH_LINK"}
   export CXXFLAGS="${ ARCH_FLAGS} –fpic –ffunction-sections –funwind-tables –fstack-
        protector –fno-strict-aliasing –finline-limited=64 –frtti –fexceptions"
   ```

```
cd $OPENSSL_SRC_PATH
export CC="$STANDALONE_TOOCHAIN_PATH/bin/i686-linux-android-gcc –mtune=atome –
     march=atom –sysroot=$STANDALONE_TOOCHAIN_PATH/sysroot"
export AR=$STANDALONE_TOOCHAIN_PATH/bin/i686-linux-android-ar
export RANLIB=$STANDALONE_TOOCHAIN_PATH/bin/i686-linux-android-ranlib
./Configure android-x86 –DOPENSSL_IA32_SSE2 –DAES_ASM –DVPAES_ASM
make
```

トップ・ディレクトリーに **libcrypto.a** が作成されます。***.so** ファイルを使用する場合は、"**Configure shared android-x86 *****" と入力します。

AOSP ソースコードを保持している場合、ndk ツールチェーンは不要です。

```
source build/envsetiup.sh
lunch <options>
make –j8
cd external/openssl
mm
```

このコマンドを実行すると、libcrypto.a が out/host/linux_x86/bin に生成されます。

Android* プロジェクトで NDK から OpenSSL* を使用する

1. **Android* プロジェクト**を作成し、任意の IDE でファイルを暗号化します。ここでは Eclipse* IDE を使用します。
2. Android.mk で、OpenSSL* 関連の関数を**ネイティブ関数**として宣言子します。
3. ソースの Android* プロジェクトで jni フォルダーを作成します。
4. jni フォルダー以下にプリコンパイル済みのインクルード・ディレクトリーを作成します。
5. jni フォルダー以下の <OpenSSL source/include/> にある openssl ライブラリー・フォルダーをインクルードします。
6. **jni/*.c** で暗号化を行う C 関数を記述し、暗号化を実装します。その後、***.a/*.so** とヘッダーファイルをプロジェクトにコピーします。
7. ステップ 1 でシステム・ライブラリーとして作成した Android* クラス関数で、jni フォルダーにあるライブラリーと C 実装をロードします。

以下に、アプリケーションに openssl ライブラリーをインクルードして、Java* クラスで呼び出す方法を示します。

新しいプロジェクトを作成します。ここでは、Eclipse* で EncryptFileOpenSSL というプロジェクトを作成します。Eclipse* (Project Explorer でプロジェクト名を右クリック) またはターミナルで、jni ディレクトリーを作成し、次に pre-compiled サブディレクトリーと include サブディレクトリーを作成します。

ターミナルを使用する場合:

```
cd <workspace/of/Project>
mkdir jni/pre-compiled/
mkdir jni/include
cp $OPENSSL_PATH/libcrypto.a jni/pre-compiled
cp –L -rf $OPENSSL_PATH/include/openssl jni/include
gedit jni/Android.mk
```

次の行を **jni/Android.mk** ファイルに追加します。

```
…
LOCAL_MODULE := static
LOCAL_SRC_FILES := pre-compiled/libcrypto.a
…
LOCAL_C_INCLUDES := include
LOCAL_STATIC_LIBRARIES := static –lcrypto
```

…

OpenSSL* により提供される関数を利用して **encrypt/decrypt/SSL** 関数を実装します。インテル® AES-NI を使用するには、次に示すように、一連の **EVP_*** 関数を利用します。そうすることで、CPU でサポートされる場合、自動的にインテル® AES-NI による AES 暗号化/復号化のアクセラレーションが行われます。例えば、OpenSSL* プロバイダーを利用してファイルを暗号化するクラスを記述する場合、*.java クラスの暗号化関数は次のようになります (このソースコードは、Christopher Bird の「サンプルコード: データ暗号化アプリケーション」からの引用です)。

```
public long encryptFile(String encFilepath, String origFilepath) {

    File fileIn = new File(origFilepath);
        if (fileIn.isFile()) {
            ret = encodeFileFromJNI(encFilepath, origFilepath);
        } else {
            Log.d(TAG, "ERROR*** File does not exist:" + origFilepath);
            seconds = -1;
        }

        if (ret == -1) {
            throw new IllegalArgumentException("encrypt file execution did not succeed.");
        }
    }

    /* encodeFile ライブラリーにあるネイティブ関数 */
    public native int encodeFileFromJNI(String fileOut, String fileIn);
    public native void setBlocksizeFromJNI(int blocksize);
    public native byte[] generateKeyFromJNI(int keysize);

    /* アプリケーションの起動時に暗号化を行うライブラリー (encodeFile) をロードします。
    * ライブラリーは、パッケージ・マネージャーにより、インストール時に
    * /data/data/com.example.opensssldataencryption/lib/libencodeFile.so
    * に展開されています。
    */
    static {
        System.loadLibrary("crypto");
        System.loadLibrary("encodeFile");
    }
```

System.loadLibrary を使用してロードした encodeFile.cpp にある暗号化関数は、次のようになります。

```
int encodeFile(const char* filenameOut, const char* filenameIn) {
    int ret = 0;
    int filenameInSize = strlen(filenameIn)*sizeof(char)+1;
    int filenameOutSize = strlen(filenameOut)*sizeof(char)+1;

    char filename[filenameInSize];
    char encFilename[filenameOutSize];

    // 初期化されていない場合はキーを作成
    int seedbytes = 1024;

    memset(cKeyBuffer, 0, KEYSIZE );

    if (!opensslIsSeeded) {
        if (!RAND_load_file("/dev/urandom", seedbytes)) {
            //__android_log_print(ANDROID_LOG_ERROR, TAG, "Failed to seed OpenSSL RNG");
            return -1;
        }
        opensslIsSeeded = 1;
    }

    if (!RAND_bytes((unsigned char *)cKeyBuffer, KEYSIZE )) {
        //__android_log_print(ANDROID_LOG_ERROR, TAG,
        //    "Faled to create OpenSSSL random integers: %ul", ERR_get_error);
    }
```

```
strncpy(encFilename, filenameOut, filenameOutSize);
encFilename[filenameOutSize-1]=0;
strncpy(filename, filenameIn, filenameInSize);
filename[filenameInSize-1]=0;

EVP_CIPHER_CTX *e_ctx = EVP_CIPHER_CTX_new();

FILE *orig_file, *enc_file;

printf ("filename: %s\n" ,filename );
printf ("enc filename: %s\n" ,encFilename );
orig_file = fopen( filename, "rb" );
enc_file = fopen ( encFilename, "wb" );

unsigned char *encData, *origData;
int encData_len = 0;
int len = 0;
int bytesread = 0;

/**
* 暗号化
*/
//if (!(EVP_EncryptInit_ex(e_ctx, EVP_aes_256_cbc(), NULL, key, iv ))) {
if (!(EVP_EncryptInit_ex(e_ctx, EVP_aes_256_cbc(), NULL, cKeyBuffer, iv ))) {
    ret = -1;
    printf( "ERROR: EVP_ENCRYPTINIT_EX\n");
}
// 暗号化するファイルの準備
if ( orig_file != NULL ) {
    origData = new unsigned char[aes_blocksize];
    encData = new unsigned char[aes_blocksize+EVP_CIPHER_CTX_block_size(e_ctx)];
    // 暗号化でオリジナルよりも 16 バイト長くなる可能性がある

    printf( "Encoding file: %s\n", filename);

    bytesread = fread(origData, 1, aes_blocksize, orig_file);
    // ファイルからバイトを読み込んで暗号化ルーチンに渡す
    while ( bytesread ) {
        if (!(EVP_EncryptUpdate(e_ctx, encData, &len, origData, bytesread))) {
            ret = -1;
            printf( "ERROR: EVP_ENCRYPTUPDATE\n");
        }
        encData_len = len;

        fwrite(encData, 1, encData_len, enc_file );
        // 次のバイトを読み込む
        bytesread = fread(origData, 1, aes_blocksize, orig_file);
    }
    // 暗号化の最終ステップ
    if (!(EVP_EncryptFinal_ex(e_ctx, encData, &len))) {
        ret = -1;
        printf( "ERROR: EVP_ENCRYPTFINAL_EX\n");
    }
    encData_len = len;

    fwrite(encData, 1, encData_len, enc_file );

    // 暗号化ルーチンを解放
    EVP_CIPHER_CTX_free(e_ctx);

    // ファイルを閉じる
    printf( "\t>>\n");

    fclose(orig_file);
    fclose(enc_file);
} else {
```

```
        printf( "Unable to open files for encoding\n");
        ret = -1;
        return ret;
    }
    return ret;
}
```

次に、**ndk-build** を使用して <source of Application> でコンパイルします。
/<path to android-ndk>/ndk-build APP_ABI=x86

/<PATH\TO\OPENSSL>/include/openssl ディレクトリーを </PATH\to\PROJECT\workspace>/jni/ 以下にコピーします。

.so/.a は、/</PATH\to\PROJECT\workspace>/libs/x86/ または
/</PATH\to\PROJECT\workspace>/libs/armeabi/ に配置します。

暗号化/複合化に使用する encode.cpp ファイルは、</PATH\to\PROJECT\workspace>/jni/ に配置します。

パフォーマンス解析

次の関数を使用して、CPU 使用状況、メモリー使用状況、ファイルの暗号化にかかった時間を解析することができます。ここで使用するソースコードも Christopher Bird のブログからの引用です。

CPU 使用状況:

以下のコードは、/proc/stat にある情報から CPU 使用状況を読み取ります。

```
public float readCPUusage() {
    try {
        RandomAccessFile reader = new RandomAccessFile("/proc/stat", "r");
        String load = reader.readLine();
        String[] toks = load.split(" ");
        long idle1 = Long.parseLong(toks[5]);
        long cpu1 = Long.parseLong(toks[2]) + Long.parseLong(toks[3])
                    + Long.parseLong(toks[4]) + Long.parseLong(toks[6])
                    + Long.parseLong(toks[7]) +Long.parseLong(toks[8]);
        try {
            Thread.sleep(360);
        } catch (Exception e) {
        }

        reader.seek(0);
        load = reader.readLine();
        reader.close();
        toks = load.split(" ");
        long idle2 = Long.parseLong(toks[5]);
        long cpu2 = Long.parseLong(toks[2]) + Long.parseLong(toks[3])
                    + Long.parseLong(toks[4]) + Long.parseLong(toks[6])
                    + Long.parseLong(toks[7]) + ong.parseLong(toks[8]);

        return (float) (cpu2 - cpu1) / ((cpu2 + idle2) - (cpu1 + idle1));
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return 0;
}
```

メモリー使用状況:

以下のコードは、利用可能なシステムメモリーを読み取ります。

Memory Info は、利用可能なメモリーに関する情報を取得するための Android* API です。

1024B = 1 KB で、1024KB = 1MB なので、利用可能なメモリーは 1024*1024 == 1048576MB です。

```
public long readMem(ActivityManager am) {
    MemoryInfo mi = new MemoryInfo();
    am.getMemoryInfo(mi);
    long availableMegs = mi.availMem / 1048576L;
    return availableMegs;
}
```

実行時間の測定:

```
start = System.currentTimeMillis();
// 暗号化を実行
stop = System.currentTimeMillis();
seconds = (stop - start);
```

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください