**DSP LAB 2016-17 Java Secure Native Interface using intel-SGX**

By Clindo Devassy and Subhadeep Manna.

Guidance by: Marcel Blöcher and Malte Viering

Arithmetic and Logic parsers for expressions are the backbone of several stand alone and distributed software applications. It can be used for evaluating essential and sensitive expressions for scientific, engineering, banking and many other such platforms. This expression is divided into tokens using tokenizer and the parser is used to evaluate the value of each token internally. Finally giving us the final output.

Enclave code

When the application that uses this parser highly depends on the genuine and precise result of the expression. It is highly important that the parser is secure from attacks and tampering. So, by using intel SGX we plan to reduce the attack surface on the arithmetic and logical parser which would be used by the application.



Fig 1: Trusted and untrusted part of an SGX application.

The untrusted application component here is the application presentation layer. Which would be the GUI in this case.

The trusted component or the **enclave code** would be the core parser module. This would be written under the ECALL section of an intel SGX enabled application.

The untrusted component would be the user interface and other related module. This would be written under OCALL.

*<Enclave\_Name>.edl* file would be used to define the trusted part of the application which is the arithmetic and logic parser module.

Below is the enclave code that we plan to implement in our project:

*Code Fragment 1: Shows functions to be called from inside an enclave*

enable\_enclave()

{

enclave {

trusted {

/\* defining ECALLs here\*/

//Input functions to Java application

//for the arithmetic expression input module

arithmetic\_parser\_moduleJNICall();

//for the logical expression input module

logical\_parser\_moduleJNICall();

};

untrusted {

/\* defining OCALLs here \*/

//call to the application user interface

application\_userinterfaceJNIcall();

};

}

}

JNI code (and C non-enclave code)

The most fundamental part of our project is to create a SGX enabled application using JNI in java. Two possible ways to approach this:

1. Link the Java application to another SGX enabled C/C++ application. (*The disadvantage being performance issue and programmer efforts*).
2. The second way to approach the problem is to call just the SGX implemented function written in native C/C++ through JNI in java. This overcomes the problems faced in the first technique.

The steps how we plan to achieve this is as follows:

1. Java code

Java module and class declaring the native methods

public class JNI\_ImplSGX {

static {

// loading the C-library

System.loadLibrary("C\_Encalve\_Module");

}

// declaration of native method

private native void enableSGX();

public static void main(String[] args) {

new JNI\_ImplSGX ().enableSGX ();

}

}

loadLibrary(String path) loads an external C library. The variable path is the path to the library.

2. Generate a C-header file (.h) which has the native function declaration using javah tool.

javah -jni JNI\_ImplSGX

3. Header which is generated contains a declaration of C function linked with the Java native method:

JNIEXPORT void JNICALL Java\_enableSGX\_call (JNIEnv \*, jobject);

4. The C code

#include <jni.h>

#include <stdio.h>

#include " JNI\_ImplSGX.h"

JNIEXPORT void JNICALL

Java\_enableSGX\_call (JNIEnv \*env, jobject obj)

{

enable\_enclave(); //enclave call to the java application modules

}

5. The above can be compiled into a (dynamic link libraries) .dll file and can be used from the Java application by copying the .dll file generated to the java application and specifying the path in *System.Loadlibrary(String path)*.

Phases:

We are using a Client-Server model for our application with following assumptions:

1. Both machines must be SGX enabled.
2. They communicate through via WebSockets.

Initial setup

The client app and server app must be started on respective hosts as separate SGX enabled applications.

Initial key exchange

We plan to check the availability of server and client on the network initially by public key exchanges matched against their private keys.

Authentication client/server

We plan to carry out the authentication of client and server using SGX Remote attestation. The flow diagram is as follows:

C:\Users\manna\AppData\Local\Microsoft\Windows\INetCacheContent.Word\Untitled Diagram.png

Fig:2 Showing Authentication of Client App using SGX remote Attestation feature.

We are following this tutorial section : <https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example#introduction>

Communication client-server-enclave

C:\Users\manna\AppData\Local\Microsoft\Windows\INetCacheContent.Word\flow_diagram.png

Fig:3 Showing Client and Server Interaction in Java SGX application

Based on the assumption that the Client and Server both are running on sgx enabled machine. The flow diagram for the project is shown above.

Interface/API of your arithmetic expression interpreter

The Arithmetic expression interpreter for our project is shown in the following diagram:

Raw string expression

Output Value

Node Evaluate and process

Grammar Evaluation

Tokenizer

Fig: 4. Arithmetic Interpreter

The main section of Arithmetic interpreter API is as follows:

1. *Parser\_Arithmetic* : This class is used to take an input string expression. The method parse() is used to pass it to the tokenizer.
2. *Core\_Tokenizer*: It splits the stream into tokens which is then returned to the parser. The tokens are defined in a list class called as Tokens.
3. *Exp\_Node*: This class gets the type of terminal or non terminal symbol. Then passed it to the respective class(to which the token belongs) for the evaluation of the expression.
4. The parser Arithmetic class returns the value of the expression as the Output Value.