Intel DAL Extended Storage Mechanism

By Shai Falach and Ron Keinan
Instructor – Mr. Barak Einav

Lev Academic Center
Computer Science faculty
5781 semester B

Table of Contents

Introduction:

Intel's DAL is a TEE (Trusted Execution Environments) that has a very small amount of flash storage.
There is a need for an infrastructure that can be used by other TAs to store data with integrity, confidentiality and replay protection without the size limitation of the physical storage.

Purpose – why, what, and how:

To create a java class that will connect to host and handle all the extended memory including encryption and decryption of the data. This class will be used by the applet.
In advance, to create a c# class that will connect to the applet and handle all the communication to get, change and send files. This class will be used by the host.

The classes should supply a similar interface to the regular flash memory used by applet, and jhi session handle used by host, to simplify the use of the extended storage for all users.

This solution is better than the flash storage memory that can save only few bytes (can go to few KB in extreme cases), and our solution is open this limitation. (This current version is set to a buffer of 16 KB, but it can be extended by demand(see future work).

The memory is saved in the Hard Disk only when it is encrypted – and every action made on the DB is only in the applet – it is the only process that decrypt & encrypt, write, change, or delete from the file.

Architecture and design:

The project is divided into 2 sections – the applet zone and the host zone.

Java Trusted Application:

- XFlashStorageSample – this is the class that is used by user and inherits the abstract class XFlashStorageApplet. It opens a session in front of the host, creates an instance of the extended flash storage, and handles it according to commands from host – reading and writing from the DB.

- XFlashStorageApplet – abstract class that inherits the regular IntelApplet class. Used to extend the applet module so the user can use the applet in the regular and simple way of use. Mainly it sets an instance of the xFlashStorage on initiation of the session and set a response buffer to return to an invoke command.

- AppletResponseModel – class to set main fields of a response from applet to host.

- XFlashStorage – The class that extends the flash secured memory of the applet. It initializes an AES symmetric encryption keys that are HW built into the platform, and an IV that is randomized in the Initialization process, and then saves the IV to the original flash memory, and then used to receive a buffer from the host that contains the secured memory DB, and in charge of reading and writing to it, and updates the host to changes in the DB. The DB itself is saved on the HARD DISK of the host and can be read and changed only inside the applet.

C# Host Application:

- Program –this is the class that is used by user. It installs a JhiClientWithExtendedStorage, opens a session in front of the applet, creates an instance of the extended flash storage, send to it commands – reading and writing from the DB.

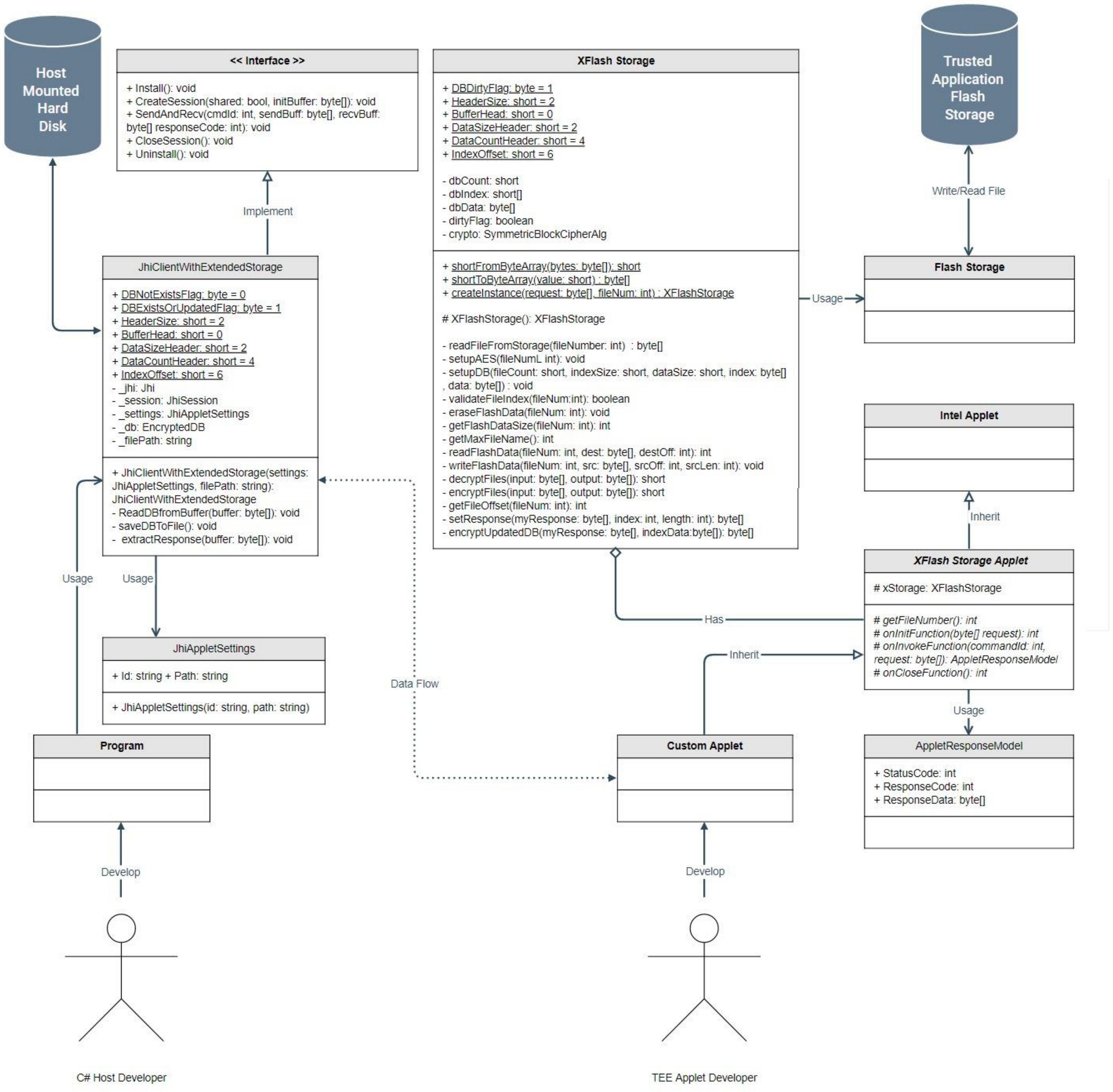
- JhiClient – interface that contains main JHI functions to be re implemented by the client with extended storage.

- JhiAppletSettings – class with main fields for an extended storage Jhi install – path of the DB and ID of the applet dalp file.

- EncryptedDB – class to represent the secured memory – DB – including buffers for chained data, indexes of files in data.

- JhiClientWithExtendedStorage - The class that helps to extend the flash secured memory of the applet. It initializes a binary file that will contain the secured decrypted data handled by the applet. For every change or read of the Db – the file id sent to the applet through the session (regular SendAndRecieve) together with a known buffer that contains information about the files number and order. It receives the buffer from applet after use and updates the DB if needed.

UML Diagram:


Host Mounted Hard Disk
<< Interface >>
+ Install(): void
+ CreateSession(shared: bool, initBuffer: byte[]): void
+ SendAndRecv(cmdId: int, sendBuff: byte[], recvBuff: byte[] responseCode: int): void
+ CloseSession(): void
+ Uninstall(): void
Implement
JhiClientWithExtendedStorage
+ DBNotExistsFlag: byte = 0
+ DBExistsOrUpdatedFlag: byte = 1
+ HeaderSize: short = 2
+ BufferHead: short = 0
+ DataSizeHeader: short = 2
+ DataCountHeader: short = 4
+ IndexOffset: short = 6
- _jhi: Jhi
- _session: JhiSession
- _settings: JhiAppletSettings
- _db: EncryptedDB
- _filePath: string
+ JhiClientWithExtendedStorage(settings: JhiAppletSettings, filePath: string): JhiClientWithExtendedStorage
- ReadDBfromBuffer(buffer: byte[]): void
- saveDBToFile(): void
- extractResponse(buffer: byte[]): void
Usage
Usage
JhiAppletSettings
+ Id: string + Path: string
+ JhiAppletSettings(id: string, path: string)
Program
Develop
C# Host Developer
XFlash Storage
+ DBDirtyFlag: byte = 1
+ HeaderSize: short = 2
+ BufferHead: short = 0
+ DataSizeHeader: short = 2
+ DataCountHeader: short = 4
+ IndexOffset: short = 6
- dbCount: short
- dbIndex: short[]
- dbData: byte[]
- dirtyFlag: boolean
- crypto: SymmetricBlockCipherAlg
+ shortFromByteArray(bytes: byte[]): short
+ shortToByteArray(value: short) : byte[]
+ createInstance(request: byte[], fileNum: int) : XFlashStorage
# XFlashStorage(): XFlashStorage
- readFileFromStorage(fileNumber: int) : byte[]
- setupAES(fileNumL int): void
- setupDB(fileCount: short, indexSize: short, dataSize: short, index: byte[], data: byte[]) : void
- validateFileIndex(fileNum:int): boolean
- eraseFlashData(fileNum: int): void
- getFlashDataSize(fileNum: int): int
- getMaxFileName(): int
- readFlashData(fileNum: int, dest: byte[], destOff: int): int
- writeFlashData(fileNum: int, src: byte[], srcOff: int, srcLen: int): void
- decryptFiles(input: byte[], output: byte[]): short
- encryptFiles(input: byte[], output: byte[]): short
- getFileOffset(fileNum: int): int
- setResponse(myResponse: byte[], index: int, length: int): byte[]
- encryptUpdatedDB(myResponse: byte[], indexData:byte[]): byte[]
Data Flow
Has
Inherit
Custom Applet
Develop
TEE Applet Developer
Trusted Application Flash Storage
Write/Read File
Flash Storage
Usage
Intel Applet
Inherit
XFlash Storage Applet
# xStorage: XFlashStorage
# getFileNumber(): int
# onInitFunction(byte[] request): int
# onInvokeFunction(commandId: int, request: byte[]): AppletResponseModel
# onCloseFunction(): int
Usage
AppletResponseModel
+ StatusCode: int
+ ResponseCode: int
+ ResponseData: byte[]

Security analysis – threat modeling:

XFlashStorage_Threa
tModel.htm

Mode of work:

The entire project was written together, in pair programming, through the following steps:

- At first, we wrote an initial report including our goals.
- From the report we created a prototype API for the applet and host - including the structure and functionality of the main class for extended storage.
- After discussing the API with Barak, the instructor, we understood our faults and updated the API. mainly – we decided to design the class so that will be like the regular modules – the flash library in tha applet and the Jhi interface in the host – to simplify the use in the extended storage and make it familiar to all users.
- we finished writing all the classes needed to the project (can be seen in Architecture).
- Finally, this report and its threat analysis were written.

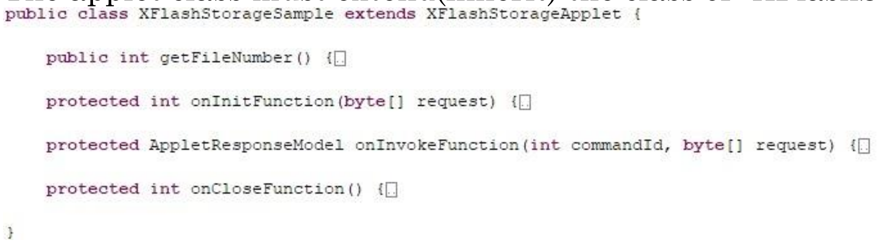
Future work:

- Working with directory instead of single binary file as DB – need to invent protocol scheme that will know how to read from several files, and send them through the session in an iterative way (16kb is the limit for sending). For now the extended memory allows 16KB, by working with a file system, the memory size will be 16 * number of files.
- Implement reply attack mitigation – currently protected only if the trusted application is protected.
- Working with API over https – the host will provide REST API for handling data in a remote server. Can provide more stability and availability.

How to use the project:

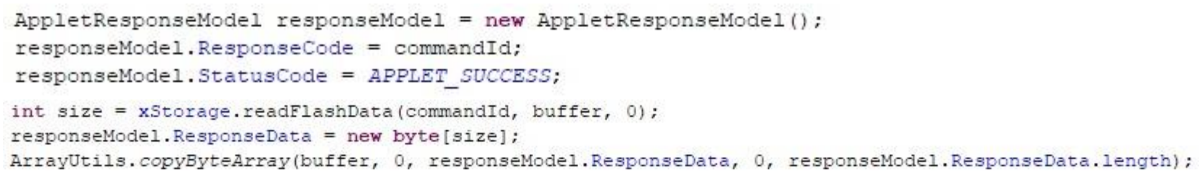
Applet side:

The applet class must extend(inherit) the class of "XFlashStorageApplet":



```
public class XFlashStorageSample extends XFlashStorageApplet {

    public int getFileNumber() {

    protected int onInitFunction(byte[] request) {

    protected AppletResponseModel onInvokeFunction(int commandId, byte[] request) {

    protected int onCloseFunction() {

}
```

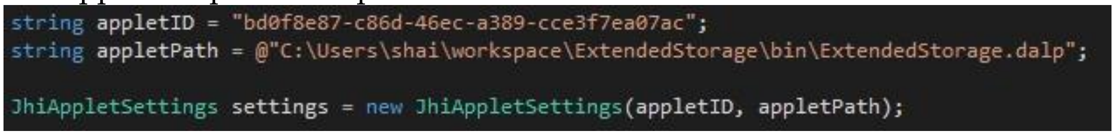
Now it contains an instance if the XFlashStorage and can call its different functions in the "invokeCommand" function – to read and write from DB. Decryption and encryption will happen automatically by the XFlashStorage instance.
The class receives form the host the updated DB as a buffer, handles it and sets a response to send back to host (with the DB after changes to be saved in HD by the host).
In invoke command function:



```
AppletResponseModel responseModel = new AppletResponseModel();
responseModel.ResponseCode = commandId;
responseModel.StatusCode = APPLET_SUCCESS;

int size = xStorage.readFlashData(commandId, buffer, 0);
responseModel.ResponseData = new byte[size];
ArrayUtils.copyByteArray(buffer, 0, responseModel.ResponseData, 0, responseModel.ResponseData.length);
```

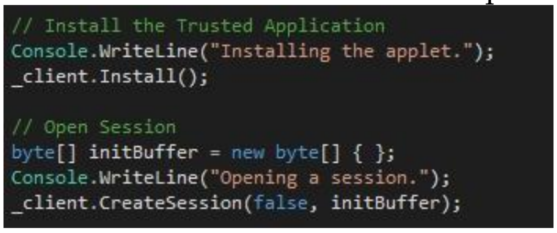
Host side:

The host class must create instance of the JhiClientWithExtendedStorage, set him settings of the applet dalp file and path for the DB:



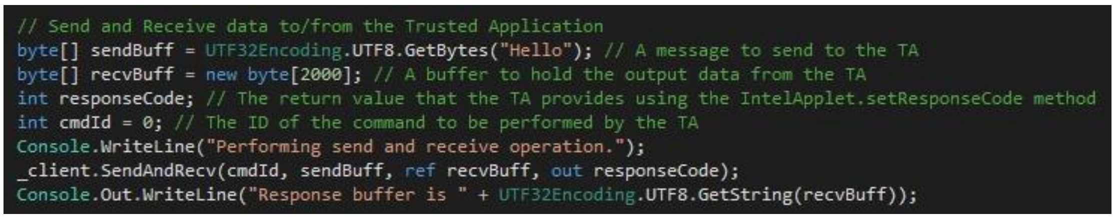
```
string appletID = "bd0f8e87-c86d-46ec-a389-cce3f7ea07ac";
string appletPath = @"C:\Users\shai\workspace\ExtendedStorage\bin\ExtendedStorage.dalp";

JhiAppletSettings settings = new JhiAppletSettings(appletID, appletPath);
```

Then to install a Jhi client and open a session:



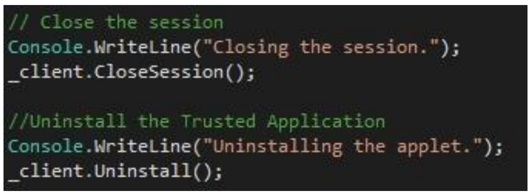
```
// Install the Trusted Application
Console.WriteLine("Installing the applet.");
_client.Install();

// Open Session
byte[] initBuffer = new byte[] { };
Console.WriteLine("Opening a session.");
_client.CreateSession(false, initBuffer);
```

Create buffers and use the SendAndRecv function in order to send and receive data:



```
// Send and Receive data to/from the Trusted Application
byte[] sendBuff = UTF32Encoding.UTF8.GetBytes("Hello"); // A message to send to the TA
byte[] recvBuff = new byte[2000]; // A buffer to hold the output data from the TA
int responseCode; // The return value that the TA provides using the IntelApplet.setResponseCode method
int cmdId = 0; // The ID of the command to be performed by the TA
Console.WriteLine("Performing send and receive operation.");
_client.SendAndRecv(cmdId, sendBuff, ref recvBuff, out responseCode);
Console.Out.WriteLine("Response buffer is " + UTF32Encoding.UTF8.GetString(recvBuff));
```

And at the end to Close and Uninstall:



```
// Close the session
Console.WriteLine("Closing the session.");
_client.CloseSession();

//Uninstall the Trusted Application
Console.WriteLine("Uninstalling the applet.");
_client.Uninstall();
```