An Integrated Testing and Debugging Environment for Java Card

JIN-HEE HAN*, SUNG-IK JUN*, SI-KWAN KIM**, KYO-IL JUNG***
IC Card Research Team, ETRI*, KUMOH national university of technology**
Dept. of Information Security Basic, ETRI***
ETRI, 161, Gajeong-Dong, Yuseong-Gu, Daejon, 305-350
KOREA
hanjh@etri.re.kr, sijun@etri.re.kr, sgkim99@naver.com, kyoil@etri.re.kr, http://www.etri.re.kr

Abstract: - This paper describes integrated testing and debugging environment for Java Card. An integrated testing and debugging environment is based on J-JCRE (Java Card Runtime Environment) and Java Card APIs (Application Programming Interface). And also, developed tool supports two kinds of cryptographic algorithms, automatic generation of client/server applet stub/skeleton, script execution, and source level debugging of system class etc. Therefore, by using development environment, application can be debugged and tested before being downloaded onto the Java Card.

Key-Words: - Java Card, Simulator, Testing, Debugging, Development tool

1 Introduction

To achieve a certain degree of confidence that a given program follows its specification, a testing phase must be included in the program development process, and also a complementary debugging phase is needed for locating the program's bugs.

There have been also commercial Java Card development kits, for instance *Odyssey lab*™ form Bull, *Cyberflex*™ form Schlumberger, *GemXpresso RAD*™ from Gemplus, *Sm@rtCafé Professional*™ from Gieseke & Devrient, *GalatIC*™ from Oberthur Card Systems. Most of these kits contain a card reader, cards and software tools [1].

Generally, simulation environment is slower than a real environment. However, in the view of a smart card, this characteristic is the exact opposite. It means that majority functins of application can be simulated and tested by using simulation environment because execution time of a real card is much slower than that of a simulator. Therefore, application developers can use a simulator as a real card and check whether their application is error-free state before downloading it onto a real card. However, like as general simulation environment, time-dependent functions of a real card should be considered .

In this paper, we introduce our integrated testing and debugging tool, but our aim is not to compete with conventional smart card manufacturers, but to propose new methods for creating development environments with beneficial functionality. The rest of this paper is organizaed as follows. In Section 2, we discuss Java Card. Then we present the architecture of our testing and debugging tool and explain how it operates. The following sections illustrate simulation process in relation to applet using message digest algorithms, debugging, and simulation results. Finally, we present conclusions.

2 Java Card

2.1 Overview

Starting in 1996, Schlumberger, a smart card manufacturer, demonstrated a Java-based smart card by adding a light-weight Java bytecode interpreter to a smart card's OS (Operating System) and downloading Java class files, which were converted to a smaller, proprietary format.

In October 1996, Sun Microsystems issued a first Java card specification. The specification limited its description to the Java card's general goals and architecture. Afterwards, Sun issued Java Card 2.0 specification, which is more essential and concrete, in 1997, and Java Card 2.1 specification in 1999. In addition, Java Card 2.2 specification in 2002.

If you want to write a Java application that should run on a smart card, you can use a smart card that is compliant to the Java card API specification. A Java card, as all common smart cards, has a Central Processing Unit (CPU), Read Only Memory (ROM) and Electrically Erasable and Programmable ROM (EEPROM). The card operating system consists of a

Java Card Virtual Machine (JCVM), a piece of software that can execute programs (applets) written in the Java language. These Java card applets are written in a similar way to the "normal" Java applets, but due to the limited memory and computing power of the smart card, only a small subset of the language features are supported.

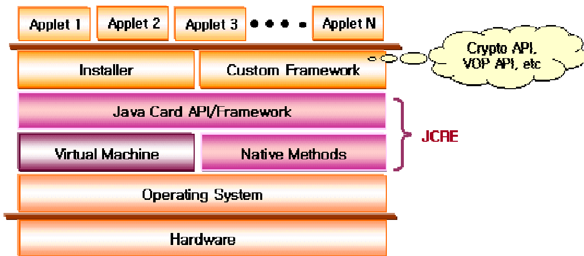

Fig.1 Java Card architecture

For the communication between smart cards and the outside world, the client–server model is used. The messages are transferred by Application Protocol Data Units (APDU), which is the communication protocol, specified in ISO 7816. The card always waits for a command APDU from a terminal, executes the requested command, and replies with an adequate response APDU. The header is always present in the command APDU. If there are no errors, a response APDU should always be returned, even if they contain no data (status words are always present). Command and response APDU's are built as shown in Table 1.

Table 1. Command and response APDU

Command APDU						
Command APDU Header				Lc	Data	Le
CLA	INS	P1	P2			
CLA: Class byte (command-ID), INS: Instruction, P1, P2: Parameter, Lc: Length of command data, Data: Command data, Le: Length of expected data						

Response APDU		
Data	SW1	SW2
Data: Response data, SW1, SW2: Status Word		

Java card applications are called applets. Applets are identified and selected by an AID (Application IDentifier), and multiple applets can reside on one card. Because applet objects exist for the life of the card, once installed an applet lives on the card forever [2][3].

2.2 Applet development process

Development of a Java Card applet begins as with any other Java program: a developer writes one or more Java classes and compiles the source code with a Java compiler. Next, the applet is run, tested and debugged in a simulation environment. The simulator simulates the Java Card runtime environment on a PC or a workstation. In the simulation environment, the applet runs on a Java virtual machine, and thus the class files of the applet are executed. Then the class files of the applet that make up a Java package are converted to a CAP (Converted APplet) file by using the Java Card converter. If an applet comprises several packages, a CAP file and an export file are created for each package. Finally, the CAP file(s) that represent the applet are loaded and tested in an emulation environment. Fig.2 demonstrates the applet development process [3].

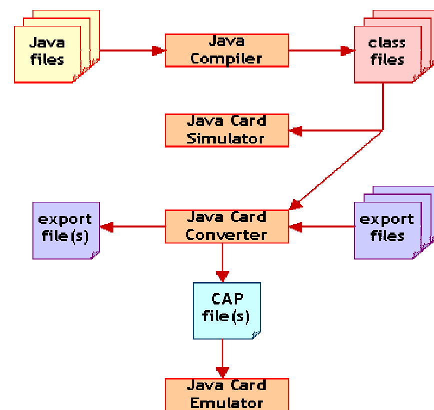

Fig.2 Applet development process

3 Testing and debugging tool

Our Java Card development environment is made up of two parts: a structure editor and a simulator for testing applets. And simulator again is composed of five parts. Fig.3 illustrates system architecture of our simulator.

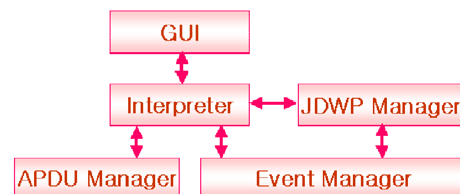

Fig.3 System architecture of simulator

3.1 Overview

As you see in Fig.3, system architecture of a simulator consists of GUI (Graphic User Interface), interpreter (On-card JCVM), JDWP (Java Debug Wire Protocol) manager, APDU (Application Protocol Data Unit) manager, and event manager.

JDWP are used for communication between a debugger and the JCVM (Java Card Virtual Machine), which it debugs. And JDWP define additional data types, constants, and commands that can be used to debug applets developed for and executed in the JCRE. JDWP manager changes debugging command into JDWP packet and processes an applicable command. Also, JDWP extracts various kinds of debugging information from CAP file [4][5][6].

Interpreter executes instructions of bytecode, which are generated by Java Card applet and APIs. And event manager processes events that are generated according to user's command. Finally, APDU manager processes command and response APDU.

3.2 Features of integrated simulation tool

Integrated simulation tool supports following features:

- Source level debugging for system classes
- Automatic generation of client/server applet stub/ skeleton
- Generation of cross-reference tree
- Message tracing between terminal application and card applet
- Script execution
- Variable watch and value change trace
- Monitoring and reporting resource used for each applet
- Support various cryptographic algorithms
- Applet compiling, converting, and mask generation

Tool bar is composed of various functions: new java file, open java file, save java file, start new project, open project, save project files, add file to project, remove file from project, abruptly terminate the target VM, suspend, resume, set breakpoint, set trace, step by line into methods, step by line over methods, step out of current frame, and start APDU toolkit. And utility item of a simulator consists of convert classes, generate CAP, dump CAP, script generation, view bytecode usages, view profile, and connect APDU tool. In addition, project item is composed of complie files, convert files, maskgen files, make all, project properties, and mask options. Fig.4 describes tool bar of our simulator.

Fig.4 Tool bar of simulator

To develop and test applet, applet developer can use tool bar, project and utility item. In the next section, we will show simulation process and results of test applet using message digest cryptographic algorithms (MD5, RIPEMD160, SHA-1) [7][8].

4 Simulations

Our simulator supports two kinds of cryptographic algorithms up to now: such as MD5, RIPEMD160, SHA-1 as hash function, and SEED as secret key (which is proposed for standardization in KOREA). It indicates that we implement MD5, RIPEMD160, SHA-1, and SEED cryptographic APIs.

In order to simulate test applet, at first, we wrote test applet to utilize message digest cryptographic algorithm. Contents of test applet are described below:

```
package com.sun.javacard.samples.Md;

import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

public class Md extends Applet
{
final static byte MD_CLA = (byte)0xC0;
final static byte RECEIVE =   (byte) 0x10;
final static byte SEND16 = (byte) 0x20;
final static byte SEND20 = (byte) 0x30;
final static byte MD5 = (byte) 0x40;
final static byte RIPEMD160 = (byte) 0x50;
final static byte SHA1 = (byte) 0x70;

static byte[] orMessage = new byte[3];
static byte[] mdMessage_16 = new byte[16];
static byte[] mdMessage = new byte[20];

static byte mesLeng;
MessageDigest md5;

private Md (byte[] bArray,short bOffset,byte bLength){
 pin = new OwnerPIN(PIN_TRY_LIMIT,   MAX_PIN_SIZE);
pin.update(bArray, bOffset, bLength);
register();
}

public static void install(byte[] bArray, short bOffset, byte bLength){
  new Md(bArray, bOffset, bLength);
}

public boolean select(){
    if ( pin.getTriesRemaining() == 0 )
```

```
      return false;
  return true;
}

public void deselect(){
  pin.reset();
}

public void process(APDU apdu) {
byte[] buffer = apdu.getBuffer();
if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
   (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)) )
 return;
if (buffer[ISO7816.OFFSET_CLA] != MD_CLA)
  ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

switch (buffer[ISO7816.OFFSET_INS]) {
case MD5:  md5(apdu);  return;
case RIPEMD160:  ripemd160(apdu);  return;
case SHA1:  sha1(apdu);  return;
case RECEIVE:  receive(apdu);  return;
case SEND16:  send16(apdu);  return;
case SEND20:  send20(apdu);  return;
default:      ISOException.throwIt
              (ISO7816.SW_INS_NOT_SUPPORTED);
}
}

private void receive(APDU apdu){
      byte[] buffer = apdu.getBuffer();
      short re = apdu.setIncomingAndReceive();
      mesLeng = buffer[ISO7816.OFFSET_LC];
      for( byte i=0; i<orMessage.length; i++){
          orMessage[i] = buffer[(byte)(ISO7816.OFFSET_CDATA+i)];
      }
}

 private void md5(APDU apdu) {
 md5 = MessageDigest.getInstance(MessageDigest.ALG_MD5,true);
 md5.doFinal(orMessage,(short)0,(short)orMessage.length,
              mdMessage_16,(short)0);
}

private void ripemd160(APDU apdu) {
 MessageDigest ripemd160 =
MessageDigest.getInstance(MessageDigest.ALG_RIPEMD160,true);
ripemd160.doFinal(orMessage, (short)0,(short)orMessage.length,
                     mdMessage,(short)0);
}

private void sha1(APDU apdu) {
 MessageDigest sha1 =
MessageDigest.getInstance(MessageDigest.ALG_SHA,true);
sha1.doFinal(orMessage, (short)0,(short)orMessage.length,
              mdMessage,(short)0);
}

private void send16(APDU apdu){
    byte[] buffer = apdu.getBuffer();
   short le = apdu.setOutgoing();
   apdu.setOutgoingLength((byte)16);
   for (byte i=0; i < 16; i++){
          buffer[i] = mdMessage_16[i];
   }
  apdu.sendBytes((short)0, (short)16);
}
private void send20(APDU apdu){
  byte[] buffer = apdu.getBuffer();
  short le = apdu.setOutgoing();
 apdu.setOutgoingLength((byte)20);
 for (byte i=0; i < 20; i++){
          buffer[i] = mdMessage[i];
   }
  apdu.sendBytes((short)0, (short)20);
 }
```

And then, applet was compiled, converted. After mask generation is completed, we can use various functions to test and debug developed applet. Fig.5 illustrates simulation process of message digest applet and various debugging functions.

As you see in Fig.5, simulation tool supports several beneficial functions such as view profile, view bytecode usages, message trace, and APDU monitor.

5 Results

Fig.6 shows simulation results of message digest applet. In this figure, we can see that script execution starts after receiving ATR (Answer To Reset) and applet operates very well.

Script file used for simulation is organized as follows:

- ♦ Select Installer applet
- ♦ Begin Installer
- ♦ Create test applet
- ♦ End Installer
- ♦ Select test applet
- ♦ Send input message
- ♦ Run MD5 algorithm
- ♦ Receive hash value of MD5 algorithm
- ♦ Run RIPEMD160 algorithm
- ♦ Receive hash value of RIPEMD160 algorithm
- ♦ Run SHA-1 algorithm
- ♦ Receive hash value of SHA-1 algorithm

Input and output data of each message digest algorithms is summarized in Table 2.

Table 2. Input and ouput data of hash functions

	Input	Output
MD5	abc	900150983cd24fb0d6963f7d28e17f72
RIPEMD 160	abc	8eb208f7e05d987a9b044a8e98c6b087f15a0bfc
SHA-1	abc	a9993e364706816aba3e25717850c26c9cd0d89d

6 Conclusions and future work

As mentioned before, in order to achieve a certain degree of confidence that a given program follows its specification, a testing phase must be included in the

program development process, and also a debugging phase to help locating the program's bugs. Therefore, we introduce our simulation tool developed for Java Card in this paper and briefly show its simulation process.

Our integrated testing and debugging tool now supports two kinds of cryptographic algorithms and provides various kinds of debug functions such as script execution, APDU message monitoring, source level debugging for system classes, and monitoring and reporting resource used for each applet and so on.

In the near future, we will add various kinds of cryptographic algorithms, such as RSA, ECC as public key algorithms, to our simulator and also Open Platform 2.1 specification will be implemented later.

References:

[1] Isabelle, A. Et al., An integrated development environment for Java Card, *Computer Networks*, 2001, pp. 391-405.

[2] Michael Caentsch, Java Card-From Hype to Reality, *IEEE Concurrency*, 1999, pp. 36-43.

[3] Chen, Zhiqun, *Java Card Technology for Smart Cards*, Addison-wesley, 2000

[4] Sun Microsystems Inc., *Java Card™ 2.1.2 Development Kit User's Guide,* 2001

[5] Sun Microsystems Inc., *Java™ Debug Wire Protocol Java Card™ Extensions,* 2001

[6] Sun Microsystems Inc., *Java Card™ 2.1.1 Runtime Environment Specification,* 2000

[7] Sun Microsystems Inc., *Java Card™ 2.1.1 Application Programming Interface Specification,* 2000

[8] Menezed, A., van Oorschot, P., Vanstone, S., *Handbook of Applied Cryptography*, CRC Press, 1997

Fig.5. Various debugging tool for applet simulation

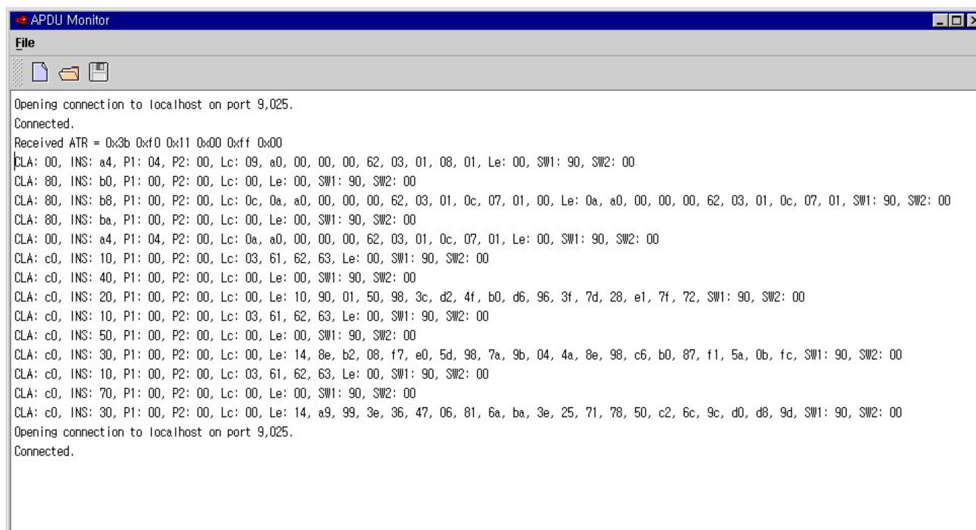

Fig.6. Simulation results of message digest applet