Prototype Implementation of an Integrated Interpreter and Analyser of Carmel Programs

Authors:	Igor Siveroni and Luke Jackson
Date:	October, 2003
Number:	SECSAFE-ICSTM-015
Version:	1.0
Classification:	Public
Status:	Software - Work in Progress

1 Introduction

We present a prototype implementation of an integrated interpreter and analyser of Java Card virtual machine programs. The program is made of three components: loader, interpreter and analyser. Programs are written in Carmel [7], a language that abstract away some of the low-level elements of the JCVML while retaining the main features of Java Card. The loader is in charge of the parsing, linking and verification of Carmel programs. The interpreter implements the operational semantics of Carmel [9, 10] and provides a step-by-step visualisation of the execution of Carmel programs. The analyser performs control and data flow analysis on selected Carmel programs based on the Flow Logic specification given in [2, 1] and later extended in [4]. The analysis specification is encoded into ALFP formulas which are solved by the SML/NJ implementation of the Succinct Solver [11]. The solutions generated by the solver are parsed by the program, formatted to a text file and used to provide information about basic safety and security property violations such as ill-formed transactions, unauthorised access and uncaught exceptions.

2 Loader

The Carmel program loader performs three tasks: parsing, linking and verification of Carmel programs. A Carmel program is made of a set of packages. The contents of a Carmel package (class and interface definitions) must be placed in a Carmel file, identified by the `.cml` extension. The Carmel loader uses the Java system class loader to load Carmel packages, and hence uses the CLASSPATH environment variable as a search path. An added advantage of using the Java class loader is that Carmel package files can be stored within JAR and ZIP files. As a convenience to developers, it is possible to bypass this search process and load Carmel files directly. Such files can contain multiple package definitions.

The program loader parses a Carmel program and verifies that it conforms to the Carmel syntax. An abstract syntax tree is built, which is then linked, resolving references to classes, methods, fields and instructions. References to classes not defined within the same file are also loaded.

During this process, program verification is performed. An important subset of the verifications defined in the Java Virtual Machine Specification [6] and the Java Language Specification are carried out. Examples of such tests are verifying class and class member accessibility, ensuring that methods are overridden with the same return type using equivalent or stronger access modifiers, and that there are no cycles in the abstract

```
package p1 { 0x00, 0x00, 0x00, 0x00, 0x01 };

public class Num {
   protected short val;
   protected static Num instance;

   public Num() {
      0:  load r 0
      1:  push s 5
      2:  putfield val
      3:  return
   }

   public short getVal() {
      0:  load r 0
      1:  getfield val
      2:  return s
   }

   public static void install(){
      0:  new Num
      1:  store r 1
      2:  load r 1
      3:  invokedefinite Num.<init>()
      4:  load r 1
      5:  putstatic instance
      6:  return
   }

   public static Num getInstance() {
      0:  getstatic instance
      1:  return r
   }
}
```

```
package p2 { 0x00, 0x00, 0x00, 0x00, 0x02 };

import p1.Num;

public class Test {
   Test() {
      0:  load r 0
      1:  return
   }

   public static short main() {
      0:  new Test
      1:  store r 1
      2:  load r 1
      3:  invokedefinite Test.<init>()
      4:  invokedefinite p1.Num.getInstance()
      5:  invokedefinite p1.Num.getVal()
      6:  return s
   }
}
```

Figure 1: A Carmel Program

syntax tree. A notable exception is that type checking is not performed at link time, but dynamically by the interpreter.

Figure 1 shows an example of a Carmel program. The syntax of Carmel is very similar to the syntax of JCVML (and JVML) programs. Carmel instructions differ slightly from JCVML instructions. The main difference relies on the use of extra operands, such as operand types and words numbers, that allows the encoding of several similar JCVML instruction into a single Carmel instruction. For example, there is a single `load t i` instruction, where `t` indicates the type of the value stored in local variable `i`, instead of the several `load` JCVML instructions, each specialised to work on a particular type. The example in Figure 1 shows two packages, `p1` and `p2`. Package `p2` calls methods declared in class `Num` of package `p1`. The presence of the `import p1.Num` declaration allows the correct loading and linking of the package.

3 The Interpreter

3.1 The JCVM

The Carmel interpreter is a Java implementation of the operational semantics of Carmel, as defined in [9, 10]. The interpreter models the following elements of the virtual machine state:

- Call stack, comprised of stack frames. Each stack frame is made of the frame's owner, the executing method, program counter (instruction address, file name and line number), operand stack and local variable array.
- Heap: A mapping from locations to Class instances (owner and value of each instance field) and Array values (owner, length and value at each index). It also contains information about the size in bytes of each allocated object and the total memory allocated.
- Static fields: Each loaded class contains a map from each static field to its value.

Once the program and all of its referenced packages haven been successfully parsed, linked and verified, execution can begin. The entry point of a program is a either a static method that takes no parameters, a default constructor, or an instance method that takes no parameters. Executing an instance method requires selecting a class instance on the heap on which to invoke the method. There is no constraint on method return type.

Execution takes place in a separate thread which can be paused before the execution of any instruction. It is possible to step through a program, stepping into, over and out of method invocations. An extra step is added to be able to visualise the moment when the interpreter finds the appropriate handler for a thrown exception.

In Figure 2 we show a snapshot of the execution of the Carmel program presented in Figure 1. The figure presents the state of the JCVM before the execution of the `invokedefinite` instruction located at program counter (pc) 3 of method `main()` in class `p2.Test`, as indicated by the information displayed by the call stack. The GUI also shows the contents of the operand stack and local variable array. The top of the stack contains the reference value, denoted by address 34, that corresponds to the location of the `p2.Test` class instance created by the first instruction of the method and subsequently stored in local variable 1. The Heap element displays the addresses of all objects allocated by the JCRE (runtime exceptions) and during the execution of the program. For example, address 31 corresponds to the object of class `p1.Num` instantiated by the `p1.install` static method. We can also see the contents (fields) of class instances and arrays e.g the value of the `val` field of the `p1.Num` object mentioned above is 5.

The interpreter performs basic firewall checks. In particular, it prevents object accesses across contexts i.e access of objects owned by different packages is not allowed. In the example above, the `invokedefinite`

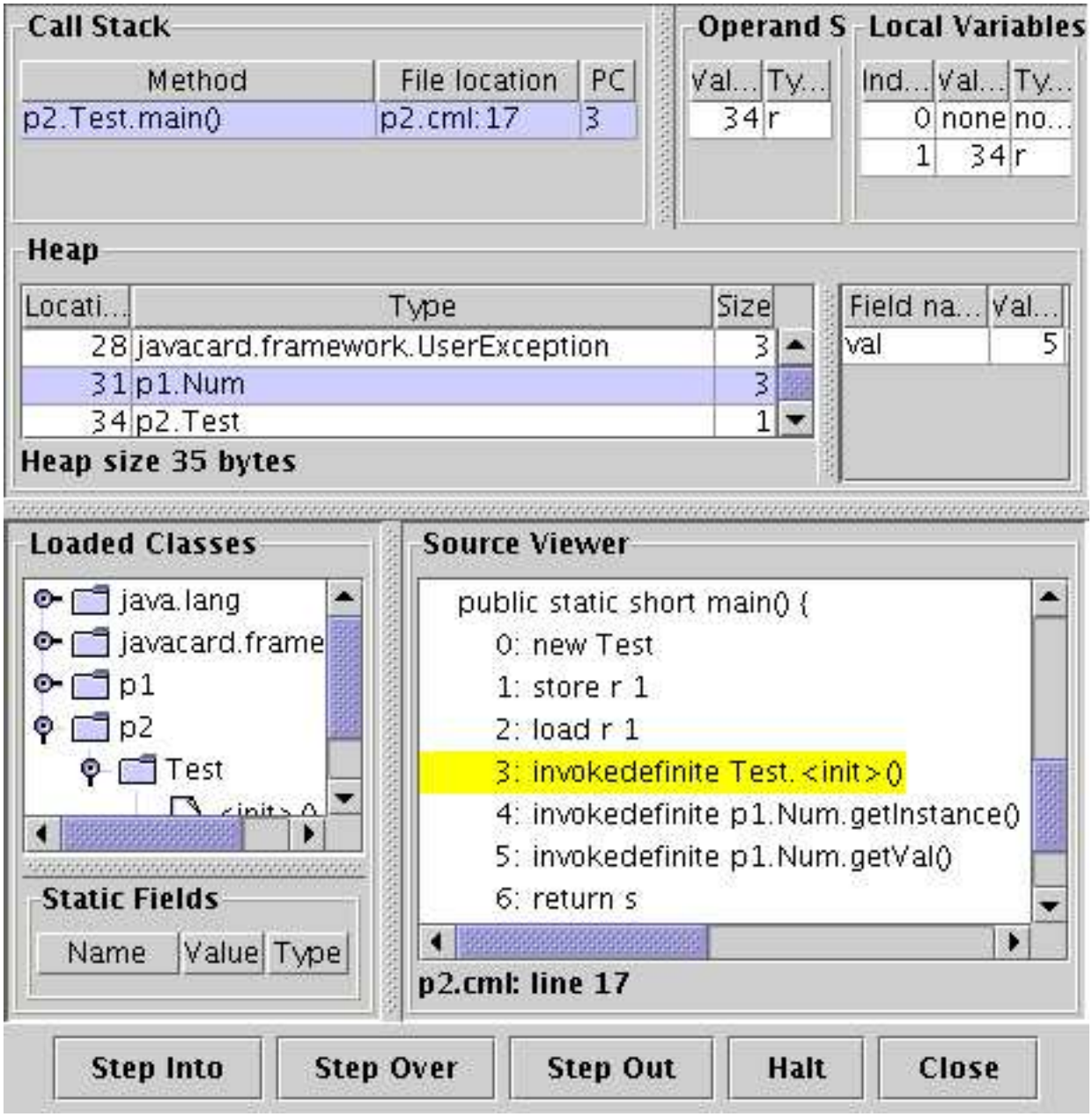

Figure 2: The Interpreter GUI

instruction located at pc 5 of method `p2.Test.main()` tries to invoke method `p1.Num.getVal()` on an object owned by package `p1` (address 31). This operation causes the interpreter to throw a `SecurityException`, reported by a window similar to:

where address 6 corresponds to the `SecurityException` class instance owned by the system (JCRE).

3.2 The JCRE and API

The interpreter implements certain aspects of the JCRE. JCRE features are either directly implemented by the program, such as transactions and applet table manipulation, or partially implemented by Carmel code in the form of loaded API classes. Similarly, certain API methods are provided as Carmel code or declared as native and, therefore, implemented by the program.

4 The Analyzer

The analyser is made of three components: Constraint generator, constraint solver and the results display.

4.1 Constraint Generator

The constraint generator traverses the selected Carmel package and generates the respective constraints - per class, method and instruction - according to the Flow Logic specification proposed in [4]. The constraints are written in ALFP and their textual representation is copied into a separate text file. An example of this process can be found in [3], where Hansen describes the translation of a Flow Logic specification of a 0-CFA of Carmel into ALFP constraints.

We show the constraints generated for the `new` instruction located at pc 0 of method `p2.Test.main()`. The ALFP clauses shown above reason about instruction reachability and the data flow between the elements of the current frame: operand stack and local variable array. We have:

```
/* **method: p2.Test.main() is m_108 */
/* 0: new p2.Test */
&
(A cc. A o1. A o2. ((Reach(m_108,pc_0,cc) & cc = CC(o1,o2)) =>
                      (S(m_108,pc_1,cc,zero,Ref(cl_30,m_108,pc_0,o1)))))
&
(A cc. A y. A i. A a. ((Reach(m_108,pc_0,cc) & y = suc(i) &
                      S(m_108,pc_0,cc,i,a)) => (S(m_108,pc_1,cc,y,a))))
&
(A cc. A x. A a. ((Reach(m_108,pc_0,cc) & L(m_108,pc_0,cc,x,a)) =>
                      (L(m_108,pc_1,cc,x,a))))
&
(A cc. ((Reach(m_108,pc_0,cc)) => (Reach(m_108,pc_1,cc))))
```

The analysis performs two important abstractions: on objects and call contexts (call stack). Objects are abstracted to their class, owner and creation point, that is, all objects of the same class, owned by the same package and created by the same `new` instruction are denoted by the same element (abstract reference). Call contexts (`cc`) are represented by two elements (owners): the current context and the context that was active before the last context switch. The interaction between call contexts and ownership can be seen in the first clause above. The clause says that if the instruction at address (`m_108,0`) is reachable and the current owner of the frame is `o_1` then the top of the stack of the next instruction (`m_108,1`) must contain an abstract reference to an object of class `cl_30` (methods, fields, classes and packages are represented by their index in the AST), created at address (`m_108,0`) and owned by `o_1`.

4.2 Constraint Solver

We use the SML/NJ implementation of the Succinct Solver. Communication between the main program and the solver is made via text files. The solver takes as input a text file containing ALFP formulas and outputs another text file with the solution, presented as a list of relations and its elements.

The text file generated by the solver is made of a list the relations used by the constraints and its members. For example, the elements of the relation `S` used to model the operand stack are listed using the following format:

```
Relation S/5:
(m_109, pc_1, CC (Own (p_3), Own (JCRE)), zero, Ref (cl_30, m_108, pc_0, Own (p_3))),
(m_108, pc_5, CC (Own (p_3), Own (JCRE)), zero, Ref (cl_29, m_105, pc_0, Own (p_2))),
....
(m_105, pc_1, CC (Own (p_2), Own (JCRE)), zero, Ref (cl_29, m_105, pc_0, Own (p_2))),
```

The last element of the list above indicates that the top element (`zero`) of the operand stack at instruction (`m_105, pc_1`) during the execution of context (`p_2,JCRE`) contains a reference to an instance of class `cl_29` created at instruction (`m_105, pc_0`) and owned by `p_2`.

4.3 Results Display

The file generated by the solver is parsed. The results, represented by a list of relation objects, are marshaled to the respective program elements. For example, the contents of the (S) (operand stack) and `L` (local variable) relations are processed and copied to the respective instruction object of the AST. The main goal of the program is to present the analysis results in the most clear and meaningful way possible. We plan to achieve this by preparing a special GUI that relates program points and program structures with control and data flow information, and informs the user of possible security and safety threats.

The current version of the program prepares a formatted text file where the program is listed with the relevant analysis results. For example, the results associated to the first two instructions of the `p1.Num.install()` method are presented in the following way:

```
**Method: p1.Num.install
->0: new p1.Num
  S[]  L[(t:0)]
->1: store r 1
  S[(0:Ref<p1.Num@install,0>)]  L[(t:0)]
```

The results above indicate that the top of the stack at pc 1 after the execution of the `new` instruction contains a reference to an instance of the `p1.Num` class created at address (`install,0`). The `->` symbol indicates that the respective instruction is reachable. Tha complete analysis result generated for package `p2` is:

```
Package: p2

*Class: p2.Test
**Constructor: p2.Test.<init>
->0: load r 0
  S[]  L[(0:Ref<p2.Test@main,0>)]
->1: return
  S[(0:Ref<p2.Test@main,0>)]  L[(0:Ref<p2.Test@main,0>)]

**Method: p2.Test.main
%%Exceptions:
     Ref<java.lang.SecurityException@JCRE,0> @ (main,5)
->0: new p2.Test
  S[]  L[(t:0)]
->1: store r 1
  S[(0:Ref<p2.Test@main,0>)]  L[(t:0)]
->2: load r 1
  S[]  L[(1:Ref<p2.Test@main,0>)(t:0)]
->3: invokedefinite p2.Test.<init>
  S[(0:Ref<p2.Test@main,0>)]  L[(t:0)(1:Ref<p2.Test@main,0>)]
->4: invokedefinite p1.Num.getInstance
  S[]  L[(1:Ref<p2.Test@main,0>)(t:0)]
->5: invokedefinite p1.Num.getVal
  S[(0:Ref<p1.Num@install,0>)]  L[(t:0)(1:Ref<p2.Test@main,0>)]
  6: return s

***** End Class *****
******** End Package:p2*******
```

The `SecurityException` thrown by the `p2.Test.main()` method at pc 5 is reported by the `Exceptions` section at the beginning of the method. Unreachable instructions, such as the `return` instruction at pc 6, are not marked, and information is neither generated nor displayed.

The analysis keeps track of instance (Heap) and static field values as well. The results generated for package `p1` (Figure 3) show that static field `instance` of class `p1.Num` may contain a reference to a `p1.Num` object, and that instance field `val` of class `p1.Num` may be 5. The same kind of information can be generated for array objects.

5 Technologies Used

The program, with the sole exception of the Succinct Solver, is implemented in Java, and requires Java 1.3 or later. The GUI is implemented in Swing, a rich set of graphical components designed around the model-view-controller design pattern, which makes them well suited to displaying the internal state of the virtual machine.

The parsers used to parse Carmel programs and Succinct Solver results are generated from grammar definition files using the JavaCC [5] tool, jointly developed by Sun and WebGain. The Succinct Solver is implemented in SML/NJ [8].

The program is currently designed to run on Unix-based platforms. There is no platform specific code in the parts of the program implemented in Java, and hence it could be run on any platform that supports Java. However, the presence of the Solver requires a platform that also supports SML/NJ. Furthermore, platform-specific code has to be included in order to implement the (very basic) interface between the program and

```
Package: p1

*Class: p1.Num
**Static Field: instance -> {Ref<p1.Num@install,0>,Ref<p1.Num@install,0>}
**Constructor: p1.Num.<init>
->0: load r 0
  S[]  L[(0:Ref<p1.Num@install,0>)]
->1: push s 5
  S[(0:Ref<p1.Num@install,0>)]  L[(0:Ref<p1.Num@install,0>)]
->2: putfield p1.Num.val
  S[(0:5)(1:Ref<p1.Num@install,0>)]  L[(0:Ref<p1.Num@install,0>)]
->3: return
  S[]  L[(0:Ref<p1.Num@install,0>)]

**Method: p1.Num.getInstance
->0: getstatic p1.Num.instance
  S[]  L[(t:0)]
->1: return r
  S[(0:Ref<p1.Num@install,0>)(0:Ref<p1.Num@install,0>)]  L[(t:0)]

**Method: p1.Num.install
->0: new p1.Num
  S[]  L[(t:0)]
->1: store r 1
  S[(0:Ref<p1.Num@install,0>)]  L[(t:0)]
->2: load r 1
  S[]  L[(1:Ref<p1.Num@install,0>)(t:0)]
->3: invokedefinite p1.Num.<init>
  S[(0:Ref<p1.Num@install,0>)]  L[(t:0)(1:Ref<p1.Num@install,0>)]
->4: load r 1
  S[]  L[(1:Ref<p1.Num@install,0>)(t:0)]
->5: putstatic p1.Num.instance
  S[(0:Ref<p1.Num@install,0>)]  L[(t:0)(1:Ref<p1.Num@install,0>)]
->6: return
  S[]  L[(1:Ref<p1.Num@install,0>)(t:0)]

**Method: p1.Num.getVal
  0: load r 0
  1: getfield p1.Num.val
  2: return s

***** End Class *****
******** End Package:p1*******

************ Heap **********
 (Ref<p1.Num@install,0>,val)->5
```

Figure 3: Analysis result for package p1

the solver.

The initial version of the application - the interpreter - was originally implemented to be executed from the Web. The ability to launch the program from the web is provided by Java Web Start technology. This allows Java programs to be installed with one click from a web page, and ensures that the user is always running the latest version, checking for updates before each time it is run.

6 Conclusions and Future Work

The current version of the analysis implementation provides a framework for the development of more precise and specialised analyses for the verification of security properties. The current implementation provides information about control and data flow which takes into account object ownership and exception handling. The latter is very important for the verification of safety properties since most of them are reported as runtime exceptions, such as the detection of firewall violations and ill-formed transactions.

We plan to work on better ways of displaying the analysis results. This will mainly be done by extending the current GUI to display analysis as well as execution information. We also plan to improve the precision of the analysis by making better use of data flow information at branching points.

References

[1] René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-003-1.0, 2002.

[2] René Rydhof Hansen. Flow Logic for Carmel. SECSAFE-IMM-001-1.5, 2002.

[3] René Rydhof Hansen. Implementing the Flow Logic for Carmel. SECSAFE-IMM-004-1.0, 2002.

[4] René Rydhof Hansen and Igor Siveroni. Java card safety and security through static analysis. SECSAFE-IMM-011-1.0, 2003.

[5] Java compiler compiler. http://javacc.dev.java.net.

[6] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Co., Reading, MA, second edition, 1999.

[7] Renaud Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005, v1.7, Trusted Logic, May 2001. SecSafe public.

[8] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[9] Igor Siveroni. Operational semantics of the Java Card virtual machine. *Journal of Logic and Algebraic Programming, special issue on Smart Cards*, (To appear), 2003. SECSAFE-ICSTM-011.

[10] Igor Siveroni and Chris Hankin. A proposal of the JCVMLe operational semantics. Technical Report SECSAFE-ICSTM-001, Imperial College London, 2002. http://www.doc.ic.ac.uk/~siveroni/secsafe.

[11] Hongyan Sun, Hanne Riis Nielson, and Flemming Nielson. Data Structures in the Succinct Solver (V1.0). SECSAFE-IMM-005-1.0, 2002.