# A Prototype Tool for JavaCard Firewall Analysis

René Rydhof Hansen*

Informatics and Mathematical Modelling, Technical University of Denmark,
DK-2800 Kongens Lyngby, Denmark
E-mail: rrh@imm.dtu.dk

**Abstract.** JavaCard is a variant of the Java programming language specifically designed for use on Smart Cards. In order to support the secure execution of several different applets on a Smart Card, the JavaCard Virtual Machine implements a *firewall* that isolates applets from each other by preventing unwanted information sharing and communication between applets.
In this paper we report on a prototype tool that can statically verify that a JavaCard applet does not (try to) violate the firewall rules. Such a tool is useful for increasing confidence in the security of an applet. Furthermore, a developer can use the tool for guaranteeing in advance, ie. before the applet is deployed, that it will not throw a security exception at an inopportune moment, thus leading to more robust and user-friendly applets.

**Keywords:** JavaCard, firewall, static analysis, prototype tool.

## 1 Introduction

The JavaCard platform is a multi-applet platform, meaning that a given JavaCard may contain and execute several different applets from several different, possibly competing, vendors. In order for an applet to protect sensitive data from other, malicious, applets the JavaCard platform implements a firewall to separate applets from each other.

The firewall mediates all method invocations and field accesses between applets and determines whether or not to allow it. If a given method invocation or field access is not allowed a *security exception* is thrown. Note that security exceptions can not be caught by the program itself, but is rather communicated all the way back to the card access terminal, ie. the user.

In this paper we discuss the formal development of a prototype tool, based on well-known static analysis techniques, for verifying that a JavaCard program does not (try to) violate the JavaCard firewall. Such a tool is useful for increasing confidence in a programs behaviour (by ensuring that a program does not try to violate the security policy) and also for enhancing the robustness of a program (by guaranteeing that no security exceptions are thrown).

* This work was partially funded by the SecSafe project (EU IST-1999-29075).

Furthermore the developments in this paper are intended to show that the Flow Logic framework and methodology used for specifying and extending the analysis is well suited for adding functionality in a structured fashion in step with increasing demand for guarantees that a program has (or lacks) certain properties, thereby reducing the workload when designing analyses for such validation.

The rest of the paper is structured as follows. Section 2 discusses the language used, Section 3 gives a brief overview of the JavaCard firewall, following that Section 4 specifies a so-called *ownership analysis* that will conservatively approximate the set of possible owner contexts an object can have. The result of this analysis will form the basis for verifying programs as shown in Section 5. Section 6 then shows how the analysis and validation can be implemented by constraint generation and solving. The paper concludes with an example in Section 7 and conclusions and future work in Section 8.

## 2 Introducing Carmel

The JavaCard language contains more than 100 instructions, a significant number of which are very specialised, eg. the instruction `push0` pushes the integer constant 0 on top of the stack. Such highly specialised instructions are mainly used for optimisation purposes and are not, as such, essential features of the language.

The large number of instructions makes JavaCard rather unwieldy and work-intensive for formal treatment. Therefore we have decided to base our work upon a JavaCard derivative, called Carmel, developed as part of the SecSafe EU project, cf. [13].

Carmel is directly derived from JavaCard, mainly by removing non-essential instructions and adding more generality to the remaining instructions ending up with a language consisting of only 30 instructions while retaining all the power, flexibility and expressibility of JavaCard but in a more manageable form. Thus Carmel can be seen as a "rational reconstruction" of JavaCard. In this paper we do not consider the instructions for subroutines.

In this section we briefly, and informally, review the Carmel language. For a formal definition of the Carmel language, including an operational semantics, see [14, 15].

### 2.1 The Carmel Language

In the following, square brackets are used to denote [*optional*] arguments to a given instruction. Many instructions are explicitly annotated with the type of their argument and/or result. For the purpose of this paper, such annotations are ignored.

The instructions for basic stack manipulation in Carmel are as follows:

| | |
|---|---|
| `push` $t$ $c$ | push constant $c$ |
| `pop` $n$ | pop the top $n$ values |
| `dup` $n$ $d$ | duplicate the $n$ top values at depth $d$ |
| `swap` $m$ $n$ | swap the top $m$ values with the following $n$ values |

All arithmetic and boolean operators are combined into one instruction parameterised on the particular operation to perform. Operators pop their argument(s) from the stack and push the result:

| | |
|---|---|
| `numop` $t$ $op$ $[t']$ | use operator $op$ |

For control flow Carmel has the following instructions

| | |
|---|---|
| `goto` $L$ | unconditional jump |
| `if` $t$ $cmpOp$ $[nullCmp]$ `goto` $L$ | conditional |
| `lookupswitch` $t$ $(k_i$`=>`$L_i)_1^n$, `default=>`$L_0$ | branch on key |
| `tableswitch` $t$ $l$`=>`$(L_i)_1^n$, `default=>`$L_0$ | branch on index |

The conditional usually compares the two top elements on the stack (using $cmpOp$); optionally it compares the top element to null. The `lookupswitch` and `tableswitch` instructions are convenient for case constructs.

Methods in the Carmel language can access and modify local variables:

| | |
|---|---|
| `load` $t$ $x$ | retrieve the value of $x$ |
| `store` $t$ $x$ | store a new value in $x$ |
| `inc` $t$ $x$ $c$ | add $c$ to the value in $x$ |

The instructions for manipulating objects are as follows:

| | |
|---|---|
| `new` $\sigma$ | create a new instance of class $\sigma$ |
| `getstatic` $f$ | get the value of the static field $f$ |
| `putstatic` $f$ | store a value in the static field $f$ |
| `getfield [this]` $f$ | get the value of field $f$ |
| `putfield [this]` $f$ | store a value in field $f$ |

The instructions `getfield` and `putfield` look for a reference to the object whose field is being accessed, on top of the stack. The optional "`this`"-modifier indicates that rather than looking for such a reference on top of the stack, the current object (the one in which the `getfield` or `putfield` instruction is executed) should be used.

Two instructions are provided for dynamically checking that an object is of a certain type:

| | |
|---|---|
| `checkcast` $\sigma$ | check if an object is a subclass of $\sigma$ |
| `instanceof` $\sigma$ | check if an object is a subclass of $\sigma$ |

The above two instructions differ only in their return value: `checkcast` throws an exception if the object is not of the specified type, whereas `instanceof` simply returns 0 in that case.

The instructions below cover the various forms of method invocation:

| | |
|---|---|
| `invokevirtual` $m$ | invoke a virtual method |
| `invokestatic` $m$ | invoke a static (class) method |
| `invokespecial` $m$ | invoke class initialisation method |
| `invokeinterface` $m$ | invoke an interface |
| `return` | return from a method invocation |

Static methods are class methods and can be invoked directly from a class as opposed to virtual methods that require a class instance (an object).

Certain static information not directly available in the Carmel syntax is accessed through special auxiliary functions, eg. to obtain the return type of a method $m$ we can use the function returnType($m$) which, following tradition, we write as $m$.returnType.

JavaCard, and thus Carmel, programs rely heavily on the use of arrays, both for storage and communication. The instructions supporting arrays follow

| | |
|---|---|
| `new (array` $t$`)` | create new array |
| `arraylength` | return length of array |
| `arrayload` $t$ | load a value from an array |
| `arraystore` $t$ | store a value in an array |

Finally exceptions can be thrown using the `throw` instruction:

`throw` throw an exception

Handlers for exceptions are resolved at a higher level and thus have no direct representation in Carmel (or JavaCard).

In addition to the instructions outlined above, a typical Carmel program will use a number of higher-level library functions, eg. for copying the contents of an array or creating cryptographic keys, and standard APIs, eg. for communicating with a terminal. In the present paper we do not model the libraries and APIs.

## 3 The JavaCard Firewall

Smart Cards are often used to store sensitive information, such as cryptographic keys and personal information, and for this reason it is important that applets are protected against malicious access to its sensitive data. The JavaCard platform implements a firewall to separate applets from each other and to make sure that no unwanted access to an applets data is allowed.

The firewall policy is based on the notion of *contexts*: applets belonging to different contexts are not allowed to access each others data, neither fields nor methods, with a few exceptions discussed below.

JavaCards package structure forms the basis for contexts: two applets that are instances of classes from the same package are assigned the same context. Additionally a "system" context is defined by the JavaCard Runtime Environment (JCRE), and applets belonging to the JCRE context may access applets in all other contexts without the firewall intervening.

During execution of an applet, objects that are created are assigned an *owner context* based on the context of the applet that created it. A method executes in the context of its owner, with the exception of static methods that are executed in the context of the invoker.

Certain specialised applets may wish to share some data with another applet in a different context, eg. a wallet applet may wish to share some data with a so-called loyalty applet in order to award "bonus-points" for purchases made with the wallet. For such situations JavaCard defines two ways in which the firewall can be circumvented in a controlled manner: JCRE entry points and sharable objects. The JCRE entry points are objects owned by the JCRE specifically designed to be accessed from all other contexts. The primary example of such an entry point being the APDU, through which all communication outside the card is handled. Sharable objects can be used to grant limited access to an objects methods (not the fields) across contexts.

It should be noted that the above mechanisms merely allow for data to cross firewall boundaries, it is still the responsibility of the applet wishing to share data that it properly authenticates the applet with which to share data. In support of this, the JavaCard system library implements a limited form of stack inspection, in the form of a method called `getPreviousContextAID` that allows an applet to find out the owner context of the method executing immediately prior to the last context switch. For the sake of clarity and brevity, we do not consider the facilities sharing and stack inspection in this paper. The analyses and techniques discussed in a later section are easily extended to handle these concepts and this is briefly indicated where relevant in the following sections.

In this paper we shall not go further into the formal details of the firewall semantics, merely refer to [14, 15]. For a thorough introduction to the JavaCard firewall and sharable objects and their use, see [2].

## 4 Analysing Carmel Programs

In this section we specify a so-called *ownership analysis*, which is a static analysis that conservatively approximates the set of *owner contexts* assigned to an object. This will form the basis for verifying that no security exceptions could possibly be raised by executing the program.

In order for the ownership analysis to be semantically sound, it needs to consider all possible program executions; rather than start from scratch and designing an ownership analysis that directly considers all program executions, we designed the ownership analysis as an extension to a previously developed control flow analysis (CFA) for Carmel. This CFA is described in detail in [5], including a formal statement and proof of semantic correctness.

The CFA, and hence the ownership analysis, is specified in the *Flow Logic* framework of Nielson and Nielson, cf. [8, 12, 7]. In the following subsections we first introduce the abstract domains for the static analysis, following that is a brief overview of the Flow Logic specification framework and finally we discuss a few specification clauses for the ownership and control flow analysis in detail.

### 4.1 Abstract Domains

The abstract domains are based on a simplified version of the concrete domains used in the semantics, cf. [14]. The simplified domains ignore semantic information that is not pertinent to the analysis. This minimises unnecessary notation and increases legibility of both analysis and theoretical results.

Objects are abstracted into their class, thus object references are modeled as classes (similar to the *class object graphs* of [16]) whereas arrays are abstracted into their elementtype:

$$\mathsf{ObjRef} = \mathsf{Class} \qquad \mathsf{ArRef} = \mathsf{Type}$$

In order to enhance readability we write (Ref $\sigma$), rather than merely $\sigma$, for object references and (Ref (`array` $\tau$)) rather than $\tau$ for array references.

References are either object references or array references:

$$\mathsf{Ref} = \mathsf{ObjRef} + \mathsf{ArRef}$$

Values are taken to be either numerical values (since we are only interested in control flow and ownership, numerical values are abstracted to a single constant) or reference values and abstract values to be sets of such values:

$$\mathsf{Val} = \mathsf{Num} + \mathsf{Ref} \qquad \widehat{\mathsf{Val}} = \mathcal{P}(\mathsf{Val}) \qquad \mathsf{Num} = \{\mathsf{INT}\}$$

In [4] the control flow analysis is extended with a data flow component.

For the ownership analysis we model owners simply as a set of concrete owners and also associate an abstract owner, ie. a set of possible owners, to each method in the program, called an "owner cache":

$$\widehat{\mathsf{Owner}} = \mathcal{P}(\mathsf{Owner}) \qquad \widehat{\mathsf{OwnerCache}} = \mathsf{Method} \to \widehat{\mathsf{Owner}}$$

In order to support stack inspection the above should be extended to record not only the set of possible owners but also the set of possible context switches.

Abstract objects comprise a mapping from the field ID's of the object to the set of abstract values possibly contained in that field and also a set of possible owners for that object:

$$\widehat{\mathsf{Object}} = (\text{fieldValue} : \mathsf{FieldID} \to \widehat{\mathsf{Val}}) \times (\text{owner} : \widehat{\mathsf{Owner}})$$

Information about a given objects status as a JCRE entry point or its shareability can trivially be added here and then checked when verifying a program, cf. Section 5.

Arrays are modeled in the simplest possible way, namely as an abstract value. This means that the structure (and length) of the array is abstracted away:

$$\widehat{\mathsf{Array}} = \widehat{\mathsf{Val}}$$

Adresses consist of a method and a program counter, making adresses unique in a program. In order to correctly handle return values from method invocations a special "placeholder" address is defined for every method. This placeholder address is encoded using a special $\mathsf{END}$-token instead of the regular program counter

$$\mathsf{Addr} = \mathsf{Method} \times (\mathbb{N} \uplus \{\mathsf{END}\})$$

The first instruction in a method is assumed to be at program counter 1 and we write $(m, \mathsf{END}_m)$ for the placeholder address belonging to the method $m$.

We let $|m|$ denote the arity of method $m$, meaning the number of arguments the method expects on the operand stack.

The local heap is modeled as a (curried) map from adresses to (local) variables to abstract values. Thus in our model there is a local heap associated with *every instruction* in a method. This is similar to Freund and Mitchells approach, cf. [3].

$$\widehat{\mathsf{LocHeap}} = \mathsf{Addr} \to \mathsf{Var} \to \widehat{\mathsf{Val}}$$

For $\hat{L} \in \widehat{\mathsf{LocHeap}}$ we shall write $\hat{L}(m_1, pc_1) \sqsubseteq \hat{L}(m_2, pc_2)$ to mean

$$\forall x \in \mathrm{dom}(\hat{L}(m_1, pc_1)) : \hat{L}(m_1, pc_1)(x) \sqsubseteq \hat{L}(m_2, pc_2)(x)$$

and $\hat{L}(m_1, pc_1) \sqsubseteq_{\{x\}} \hat{L}(m_2, pc_2)$ to mean

$$\forall y \in \mathrm{dom}(\hat{L}(m_1, pc_1)) \setminus \{x\} : \hat{L}(m_1, pc_1)(y) \sqsubseteq \hat{L}(m_2, pc_2)(y)$$

Note that local variables are denoted by natural numbers and zero, ie. $\mathsf{Var} = \mathbb{N}_0$.

We now turn to the operand stack. Since the model has to be able to cope with potentially infinite operand stacks, we use the following domain as the basis for the stack model:

$$\widehat{\mathsf{Val}}^{\infty} = \widehat{\mathsf{Val}}^{\omega} \cup \widehat{\mathsf{Val}}^{*}$$

However, in anticipation of later developments and applications of the analysis, rather than using the above domain directly, we use it to induce a more convenient domain via a Galois connection (cf. [8]):

$$\widehat{\mathsf{Val}}^{\infty} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathsf{Val}}^{*})^{\top}$$

where the abstraction function, $\alpha$, simply acts as the identity on finite stacks and maps infinite stacks to top.

With the basic domain for abstract stacks in place, we now associate an abstract operand stack with every instruction in a method in order to track operations on the stack in that method:

$$\widehat{\mathsf{Stack}} = \mathsf{Addr} \to (\widehat{\mathsf{Val}}^{*})^{\top}$$

Elements of $(\widehat{\mathsf{Val}}^{*})^{\top}$ are written much in the same style as SML lists, thus $(A_1 :: A_2 :: \cdots :: X) \in (\widehat{\mathsf{Val}}^{*})^{\top}$ represents a stack with $A_1 \in \widehat{\mathsf{Val}}$ as its top element and $X \in \widehat{\mathsf{Val}}^{\omega}$ as the "bottom" of the stack. The empty stack is denoted by $\epsilon$.

We introduce the following ordering on abstract stacks, $A_1 :: \cdots :: A_n$ and $B_1 :: \cdots :: B_m$, from $(\widehat{\mathsf{Val}}^*)^\top$:

$$(A_1 :: \cdots :: A_n) \sqsubseteq (B_1 :: \cdots :: B_m) \iff \\ m \geq n \wedge \forall i \in \{1, \ldots, n\} : A_i \subseteq B_i$$

In the interest of succinctness we shall abuse the above notation slightly by writing $(A_0 :: \cdots :: A_n) \sqsubseteq \hat{L}(m_0, pc_0)[0..n]$ as a shorthand for

$$\forall i \in \{0, \ldots, n\} : A_i \subseteq \hat{L}(m_0, pc_0)(i)$$

The abstract global heap comprises two components: an object component, keeping track of instance fields of individual objects, and a static component, that tracks the values of static fields for each class:

$$\widehat{\mathsf{StaHeap}} = \mathsf{FieldID} \rightarrow \widehat{\mathsf{Val}} \qquad \widehat{\mathsf{Heap}} = (\mathsf{ObjRef} \rightarrow \widehat{\mathsf{Object}}) + (\mathsf{ArRef} \rightarrow \widehat{\mathsf{Array}})$$

### 4.2 The Flow Logic Framework

The Flow Logic framework can be seen as a "specification approach" to static analysis, rather than an "implementation approach". In the framework, instead of detailing how a particular static analysis is to be carried out, it is specified what it means for an analysis result (or rather a *proposed* analysis result) to be acceptable (correct) with respect to a program. Flow Logic specifications are usually classified as either *verbose* or *succinct* according to the style of specification: succinct resembling the style of type-systems in only reporting "top-level" information and verbose more like traditional data flow and constraint based analyses in recording all internal flows. The specification in this paper is a verbose specification. We shall not go into further detail with the framework here, merely refer to [8, 12, 7] for further information.

The judgements of the Flow Logic specification for the analysis of Carmel will be on the form

$$(\hat{K}, \hat{H}, \hat{O}, \hat{L}, \hat{S}) \models \texttt{addr} : \texttt{instr}$$

where $\hat{S} \in \widehat{\mathsf{Stack}}$, $\hat{L} \in \widehat{\mathsf{LocHeap}}$, $\hat{H} \in \widehat{\mathsf{Heap}}$, $\hat{K} \in \widehat{\mathsf{StaHeap}}$, $\hat{O} \in \widehat{\mathsf{OwnerCache}}$, $\texttt{addr} \in \mathsf{Addr}$ and `instr` is the instruction at `addr`. Intuitively the above states that $(\hat{K}, \hat{H}, \hat{O}, \hat{L}, \hat{S})$ is an *acceptable* analysis for the instruction `instr` at address `addr`. A detailed discussion of the clauses and judgements for Carmel are given in the following sections.

### 4.3 Example Clauses

**The `putfield` Instruction** First the specification for the `putfield` instruction:

$$(\hat{K}, \hat{H}, \hat{O}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield } f$$

The `putfield` instruction transfers the value of the top element of the stack to the field named as argument to the instruction in the object referenced in the second element of the stack. Thus the stack must contain at least two elements:

$$A :: B :: X \triangleleft \hat{S}(m_0, pc_0) :$$

The specific object to be accessed is resolved at runtime, and a reference to that object is stored in the second (from the top) element of the stack. The value of the top element is then stored in the field of the object so referenced:

$$\forall (\text{Ref } \sigma') \in B : A \sqsubseteq \hat{H}(\text{Ref } \sigma').\text{fieldValue}(f)$$

Here we use the abstract global heap to hold information about the fields of abstract objects. As noted in Section 4.1 objects are abstracted into their class. Thus field information for all objects of the same class is merged and stored in the abstract global heap.

The bottom of the stack is then transferred to the next instruction:

$$X \sqsubseteq \hat{S}(m_0, pc_0 + 1)$$

and since no local variables were modified, the abstract local heap is transferred unchanged to the next instruction

$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$

We then arrive at the following clause for `putfield` instructions:

$$\begin{array}{ll}
(\hat{K}, \hat{H}, \hat{O}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield } f & \\
\quad \text{iff} \quad A :: B :: X \triangleleft \hat{S}(m_0, pc_0) : & \\
\qquad\quad \forall (\text{Ref } \sigma') \in B : & \\
\qquad\qquad A \sqsubseteq \hat{H}(\text{Ref } \sigma').\text{fieldValue}(f) & \\
\qquad\quad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) & \\
\qquad\quad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1) &
\end{array}$$

Note that while the `putfield` instruction is mediated by the firewall, there is no switch in owner context and therefore no constraints specific to the ownership analysis in the above specification (contrary to the case for `invokevirtual`).

**The `invokevirtual` Instruction** Finally we discuss the specification for the `invokevirtual` instruction:

$$(\hat{K}, \hat{H}, \hat{O}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual } m$$

In order to call an instance method, the `invokevirtual` instruction is used. Arguments to the method is found at the top of the stack, and as was the case for the `putfield` instruction, a reference to the specific object containing the invoked method is found on the stack, immediately following the arguments to the method:

$$A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}(m_0, pc_0) :$$

Next a method lookup is needed in order to find the actual method that is executed:

$$m_v = \mathit{methodLookup}(m, \sigma')$$

The arguments are transferred to the called method as *local variables* of the called method. Furthermore a reference to object containing the called method is passed as the first local variable (in effect a `this` pointer):

$$\{(\text{Ref } \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|]$$

Furthermore when a method is invoked in an object, the method will execute in the same owner context as the object from which it was invoked. This is modeled in the ownership analysis in the following way:

$$\hat{H}(\text{Ref } \sigma').\text{owner} \sqsubseteq \hat{O}(m_v)$$

Had we chosen to model not only the set of possible owners but also the set of possible context switches (to support stack inspection, as mentioned in Section 4.1) a further constraint, updating the set of context switches, would be needed here.

When a method invocation returns, there are two possibilities: either it does not return a value, ie. it has return type `void`, or it does return a value. In the first case, $m.\text{returnType} = \texttt{void}$, we simply copy the rest of the stack on to the next instruction:

$$\begin{array}{l} m.\text{returnType} = \texttt{void} \Rightarrow \\ \quad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \end{array}$$

In the latter case, $m.\text{returnType} \neq \texttt{void}$, the return value is the top element of the stack of the invoked method. In order to handle multiple returns from the invoked method correctly a special address is used, indicated by the $\mathsf{END}$-token discussed in Section 4.1; it is the responsibility of the clause for the `return` instruction to ensure, that all the possible stacks at all possible `return` instructions are transferred to the stack at the special address.

In order for the invoking method to access the return value, it must be transferred from the top of the stack of the *invoked* method to the top of the stack of the *invoking* method (less the arguments and the object reference):

$$\begin{array}{l} m.\text{returnType} \neq \texttt{void} \Rightarrow \\ \quad T :: Y \triangleleft \hat{S}(m_v, \mathsf{END}_{m_v}) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \end{array}$$

Finally, none of the local variables (of the invoking method) have been altered and are therefore passed on to the next instruction:

$$\hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)$$

Joining the above equations we obtain the following clause for `invokevirtual` instructions:

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{O}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual}\ m \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: B :: X \lhd \hat{S}(m_0, pc_0) : \\
\qquad \forall (\text{Ref } \sigma') \in B : \\
\qquad\quad m_v = \mathit{methodLookup}(m, \sigma') \\
\qquad\quad \{(\text{Ref } \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|] \\
\qquad\quad \hat{H}(\text{Ref } \sigma').\text{owner} \sqsubseteq \hat{O}(m_v) \\
\qquad\quad m.\text{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad T :: Y \lhd \hat{S}(m_v, \mathsf{END}_{m_v}) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad\quad m.\text{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)
\end{array}
$$

## 5 Verifying Carmel Programs

In this section we show how an acceptable analysis result, as specified in Section 4, can be used to guarantee that Carmel programs do not (try to) breach the firewall.

The `putfield` instruction is checked by the firewall and may potentially throw a security exception. The instruction is allowed to proceed only if either the method executing the `putfield` instruction is in the `JCRE` system-context, or if it has the same owner context as the object whose field is being accessed. This can be formalised as a formula to be checked against the analysis of a program: for every `putfield` instruction the following predicate must hold:

$$
\begin{array}{l}
\hat{O}(m_0) = \{\texttt{JCRE}\} \vee \\
(|\hat{O}(m_0)| = |\hat{H}(\text{Ref } \sigma').\text{owner}| = 1) \wedge (\hat{O}(m_0) = \hat{H}(\text{Ref } \sigma').\text{owner})
\end{array}
$$

where $m_0$ and (Ref $\sigma'$) are quantified as in the analysis (cf. Section 4.3). If it does hold for every `putfield` instruction then none of those instructions will violate the firewall rules. The same rationale applies to the `invokevirtual` instruction and gives rise to exactly the same predicate that should be checked.

Note that due to the conservative nature of the ownership analysis, guarantees can only be made when we are sure of the owner, ie. when $|\hat{O}(m_0)| = |\hat{H}(\text{Ref } \sigma').\text{owner}| = 1$.

Had we chosen to support sharable objects, entry points and stack inspection the observation predicates would of course be more involved, but still easily definable.

## 6 Implementation

Following the tradition of the Flow Logic framework, analyses are implemented by first converting the high-level Flow Logic specification into a corresponding *constraint generator* over a suitable constraint language.

In this section we briefly outline how the flow logic specification given in Section 4 systematically can be transformed into a specification for generating constraints in the Alternation-free Least Fixed-Point logic (ALFP). Constraints over this logic can be solved efficiently using the techniques described in [10, 11].

### 6.1 Alternation-free Least Fixed-Point logic

Formulae in ALFP consists of clauses of the following form:

$$\mathsf{pre} ::= R(x_1, \ldots, x_k) \mid \mathsf{pre}_1 \wedge \mathsf{pre}_2 \mid \mathsf{pre}_1 \vee \mathsf{pre}_2 \mid \exists x : \mathsf{pre}$$

$$\begin{aligned}\mathsf{clause} ::= {} & R(x_1, \ldots, x_k) \mid \mathbf{1} \mid \mathsf{clause}_1 \wedge \mathsf{clause}_2 \\ \mid {} & \mathsf{pre} \Rightarrow \mathsf{clause} \mid \forall x : \mathsf{clause}\end{aligned}$$

where $R$ is a $k$-ary relation symbol for $k \geq 1$ and $x_1, \ldots$ denote variables while $\mathbf{1}$ is the always true clause.

We shall not go in to any detail here, but it is straightforward to define the satisfaction relation, $(\rho, \sigma) \models_{\mathrm{ALFP}} t$, for ALFP over a universe of atomic values and interpretations $\rho$ and $\sigma$ of relation symbols and free variables respectively. For a given interpretation, $\sigma$, of constants we call an interpretation, $\rho$, of relation symbols a *solution* to a clause $\mathsf{clause}$ if indeed $(\rho, \sigma) \models_{\mathrm{ALFP}} \mathsf{clause}$.

Using the techniques of [10] it is possible to efficiently find solutions to given clauses. An implementation, called Succinct Solver, using these and other advanced techniques has been made by Nielson and Seidl and is described in [11].

### 6.2 Representing the Abstract Domains

In order to use ALFP as the basis for an implementation of the analysis, we must first find a way to represent the abstract domains of the analysis in ALFP.

Here we only show how to represent the abstract stack and ownership information and refer to [6] for a more detailed discussion on how to generate ALFP constraints from Flow Logic specifications and also how to optimise the generated constraints for speed and memory.

*Stacks.* In order to model the abstract stack we use a quarternary relation, $\mathsf{S}$, relating adresses and stack positions to values, thus the clause

$$\mathsf{S}(m_0, pc_0, [3], \mathsf{INT})$$

is intended to mean that the abstract stack at address $(m_0, pc_0)$ contains an integer value at stack position three.

Since we must be able to manipulate and calculate stack positions directly within the clauses, stack positions must be represented explicitly:

$$\begin{aligned}[0] \quad &= \mathtt{zero} \\ [n+1] &= \mathtt{suc}([n])\end{aligned}$$

since only stack positions, and not eg. local variable indices, are manipulated or calculated directly within the clauses, only these need to be represented using the above.

We can now model an abstract stack, $\hat{S}(m_0, pc_0) = A_1 :: \cdots :: A_n$, at address $(m_0, pc_0)$ where $A_i = \{a_i^1, \ldots, a_i^{j_i}\}$ as follows:

$$\begin{array}{c} \mathsf{S}(m_0, pc_0, [0], a_1^1) \wedge \ldots \wedge \mathsf{S}(m_0, pc_0, [0], a_1^{j_1}) \wedge \\ \vdots \\ \mathsf{S}(m_0, pc_0, [n-1], a_n^1) \wedge \ldots \wedge \mathsf{S}(m_0, pc_0, [n-1], a_n^{j_n}) \end{array}$$

Thus, the top of the stack is at position zero, [0], with the the rest of the stack in the following positions.

*Ownership.* The owner cache is simply modeled as a binary relation, $\mathsf{O}(m, o)$, relating a method $m$ to an owner $o$ thus representing $o \in \hat{O}(m)$. Similarly the owner field of an abstract object is modeled by another binary relation, $\mathsf{H\_OWNER}(r, o)$, relating an object reference (in effect an object) to an owner, $o$ which represents $o \in \hat{H}(r)$.owner.

The other components of the Flow Logic specification are modeled in a similar manner.

### 6.3 Generating Constraints

As for the flow logic specification of the analysis, we only show a few representative cases for the constraint generation.

The constraint generation is specified as a relation, $\rightsquigarrow$, between an instruction (at a given address) and a clause in ALFP.

**The `putfield` Instruction** Storing values in instance fields is accomplished by the `putfield`-instruction. Based on the analysis of the instruction (cf. Section 4.3) we see that the value on top of the stack is copied into the field pointed to by the reference found in the second position of the stack. Converting this to constraints we get

$$\forall r : \forall a : \mathsf{S}(m_0, pc_0, [1], r) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{H}(r, f, a)$$

Now the remainder of the stack, the original stack less the top two elements, should be copied onwards to the next instruction:

$$\forall y : \forall a : \forall i : y = [i+2] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, i, a)$$

None of the local variables were modified and should simply be copied onwards:

$$\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$$

Combining the above constraints we can formulate the clause for `putfield`:

$$\begin{array}{l}(m_0, pc_0) : \texttt{putfield}\ f \rightsquigarrow \\ \quad \forall r : \forall a : \mathsf{S}(m_0, pc_0, [1], r) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{H}(r, f, a) \\ \quad \forall y : \forall a : \forall i : y = [i + 2] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, i, a) \\ \quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)\end{array}$$

Finally, having generated the above constraints for the control flow and ownership analyses, we now generate an *observation predicate* which is a formalisation of the analysis checks described in Section 5 as an ALFP formula. Thus we obtain an implementation where the verification of the analysis result is actually carried out while computing the analysis result and thereby leveraging the efficiency of the Succinct Solver technology; alternatively we could have computed the result first and then carried out the verification at the possible cost of adding further overhead.

The observation predicate below will check if the instruction in question can possibly violate the firewall rules and if so it records the addrss of the potential violation in an auxiliary relation called $\mathsf{ALERT}$:

$$\begin{array}{l}\forall r : \mathsf{S}(m_0, pc_0, [1], r) \Rightarrow \\ \quad (\exists x \exists y : \mathsf{O}(m_0, x) \wedge x \neq \texttt{JCRE} \wedge x \neq y \wedge \mathsf{H\_OWNER}(r, y)) \Rightarrow \\ \qquad \mathsf{ALERT}(m_0, pc_0)\end{array}$$

Once the solver has found a solution all that remains is to extract the list of addresses of potential violations and present them to the user. Of course more information than merely the address can be recorded, eg. exactly which owner contexts gave rise to a potential violation; what additional information will prove to be most useful can only be discovered by experimentation.

**The `invokevirtual` Instruction** First the reference to the object where the invoked method resides is copied (as a self reference) to local variable 0 of the invoked method:

$$\forall r \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_0}, r)$$

Next the parameters are transferred from the stack of the current method to the local variables of the invoked method:

$$\begin{array}{l}\forall r \forall mv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \\ \quad \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_1}, a) \\ \qquad \qquad \vdots \\ \forall r \forall mv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \\ \quad \mathsf{S}(m_0, pc_0, [|m| - 1], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_|m|}, a)\end{array}$$

And then the ownership information is copied forward:

$$\begin{array}{l}\forall r \forall mv \forall o : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \\ \quad \mathsf{H\_OWNER}(r, o) \Rightarrow \mathsf{O}(mv, o)\end{array}$$

In case the method returns a value, that value should be put on top of the stack for the next instruction, and the rest of the current stack, less the arguments to the invoked method, is also copied forward. Thus if $m.\text{returnType} \neq$ `void` then the following constraints are generated:

$$\begin{array}{l} \forall y : \forall z : \forall a : \forall i : \\ \quad y = [i + |m| + 1] \wedge z = [i + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, z, a) \\ \forall r \forall mv \forall endmv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\text{id}, r, mv) \wedge \\ \quad \mathsf{END}(mv, endmv) \wedge \mathsf{S}(mv, endmv, [0], a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, [0], a) \end{array}$$

If on the other hand the invoked method does not return a value, then only the current stack, less the arguments to the invoked method, is copied forward. Thus if $m.\text{returnType} =$ `void` then the following constraints are generated:

$$\forall y : \forall a : \forall i : y = [i + |m| + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, [i], a)$$

Finally, since the local variables of the invoking method are not modified, they are simply copied along as well:

$$\forall x \forall a : \mathsf{L}(m_0, pc_0, x, a) \;\Rightarrow\; \mathsf{L}(m_0, pc_0 + 1, x, a)$$

As for `putfield` we construct an observation predicate for `invokevirtual` along the same lines:

$$\begin{array}{l} \forall r : \mathsf{S}(m_0, pc_0, [|m|], r) \;\Rightarrow \\ \quad (\exists x \exists y : \mathsf{O}(m_0, x) \wedge x \neq \texttt{JCRE} \wedge x \neq y \wedge \mathsf{H\_OWNER}(r, y)) \;\Rightarrow \\ \quad\quad \mathsf{ALERT}(m_0, pc_0) \end{array}$$

### 6.4 Solving the Constraints

Once the constraints are generated for all the instructions, all that remains is to solve them. As mentioned earlier an efficient solver, called Succint Solver, has been implemented by Seidl and Nielson. We shall not go into more detail here, merely refer to [10, 11] for more information.

### 6.5 The Prototype

A prototype tool for parsing Carmel programs, generating constraints as discussed in this section and interfacing with the Succinct Solver has been implemented. At the moment the tool simply presents the analysis result in its entirety, however, work on a better presentation based on the program source and using syntax colouring and hyperlinks is under way.

## 7 An Example

In Figure 1 a small Carmel example program is shown. The program defines two classes: `Account` and `Bad`. The `Account` class is intended as a (very) simplified version of an electronic purse with only two methods: `credit` that checks

```
class Account {
  void credit (int) {
    /* Do some checking... */
    load 0
    load 1
    invokevirtual Account.add
    return
  }
  void add (int) {
    load 1
    getfield this Account.balance
    numop add
    putfield this Account.balance
    return
  }
}
```

```
class Bad {
  void steal (void) {
    getstatic Account.leak
    dup 1 0
    getfield Account.balance
    push 5000
    numop add
    invokevirtual Account.add
    return
  }
}
```

**Fig. 1.** Example Carmel Program

and validates crediting an account and `add` that does the actual update of the balance.

The `Bad` class is intended as a (very) simplified attacker that credits an account with 5000 units, bypassing the sanity checks. We assume that an object reference to the account is leaked through a static field in the `Account` class, called `leak`. The firewall cannot detect and stop access through static fields since they have no owners. However, when the `Bad` class tries to access the account and get the balance, this is in violation with the firewall policy. Likewise the invocation of the `add` method is also in violation with the firewall policy. Figure 2 shows the constraints generated for the `invokevirtual Account.add` instruction in the `steal` method of class `Bad` and Figure 3 the generated observation predicate for that instruction.

```
(A r. S(cl_Bad,steal,pc_6,suc(zero),r) =>
  L(r,add,pc_1,var_0,r)) &
(A r. A a. S(cl_Bad,steal,pc_6,suc(zero),r) &
  S(cl_Bad,steal,pc_6,zero,a) => L(r,add,pc_1,var_1,a)) &
(A x. A i. A a. x = suc(suc(i)) & S(cl_Bad,steal,pc_6,x,a) =>
  S(cl_Bad,steal,pc_7,i,a)) &
(A x. A a. L(cl_Bad,steal,pc_6,x,a) => L(cl_Bad,steal,pc_7,x,a))&
(A r. A o. S(cl_Bad,steal,pc_6,suc(zero),r) & OWNER(r,o) =>
  O(r,add,o))
```

**Fig. 2.** Constraints generated for `invokevirtual Account.add` in method `steal`

```
(A r. ((S(cl_Bad,steal,pc_6,suc(zero),r)) =>
  ((E x. (E y.
    ((O(cl_Bad,steal,x) & (x != cl_JCRE) &
     (x != y) & (OWNER(r,y)))))) => (ALERT(cl_Bad,steal,pc_6)))))
```

**Fig. 3.** Observation predicate generated for `invokevirtual Account.add` in method `steal`

Since we are not modelling the JCRE we must explicitly set up the initialisation otherwise done by the JCRE:

```
OWNER(cl_Account,OWN_bank) &
OWNER(cl_Bad,OWN_hacker) &
O(cl_Account,credit,OWN_bank) &
O(cl_Account,add,OWN_bank) &
O(cl_Bad,steal,OWN_hacker)
```

The above contraints correspond to installing an application owned by `bank` and having only one class, namely `Account`, with two methods: `credit` and `add`, and then installing another application owned by `hacker` consisting of the class `Bad` with only one method `steal`. These initialisation constraints could easily be computed automatically, but they are only necessary when the JCRE is not modelled explicitly. The constraints given here assume that all methods in all classes can be invoked from "the outside" and so is a safe approximation of the actual call-patterns.

Finally, in Figure 4 the result of solving the constraints is shown, namely the `ALERT` relation. The analysis finds two possible breaches of the firewall rules,

```
Relation ALERT/3:
        (cl_Bad, steal, pc_6), (cl_Bad, steal, pc_3),
```

**Fig. 4.** Solution for `invokevirtual Account.add` in method `steal`

both in method `steal` of class `Bad`, in program line three and six respectively, corresponding to the `getfield` and `invokevirtual` instruction.

Even though the analysis decsribed in this paper is rather imprecise, we belive it is sufficient for JavaCard programs that follow the current practices. In particular, most JavaCards do not (yet) have a garbage collector and thus all memory allocation is done in the initialisation phase and only rarely are classes instantiated more than once. This justifies our abstraction of objects into classes and also the rather simple notion of ownership. In the future this may well change and then the analysis might not be precise enough, however, by adding more information to the abstract object references, eg. adding ownership information, we believe that sufficient precision can be achieved.

## 8 Conclusion and Future Work

In this paper we have shown how a prototype tool for validating JavaCard programs has been constructed by extending an existing control flow analysis with ownership information. From this extended analysis a constraint generator for the Succint Solver was systematically derived and it was discussed how verification checks are formalised, also as constraints, and verified by the Succinct Solver. We believe that such a tool will be very useful, in particular for developers, for ensuring the robustness of their programs and also for increasing confidence in the security behaviour of their programs.

Work on extending and consolidating the ideas in this paper is already underway mainly with respect to adding support for sharable objects (including stack inspection) and entry points. As noted various places in the paper, there are no conceptual difficulties in supporting sharable objects, entry points and stack inspection and requires only a few modifications.

Another direction, where work is also underway, is to make the tool more user friendly, in particular regarding the presentation of the analysis result. Here a number of possibilities are being considered and tested.

Finally we would like to investigate how well the approach scales to "real-world" JavaCard programs. While the current prototype is a rather naive implementation of the analysis we believe that by employing different optimisation strategies, discussed in [1, 6, 10, 11], we can achieve the desired scalability. Anecdotal evidence based on experiments with a similar constraint generator (only for control flow analysis), seems to indicate that time complexity will be less of an issue than space consumption, indeed constraint generation and solution for a program of a few thousand lines only takes around 30 seconds. In [6] a number of ideas for reducing space consumption is described. Furthermore, even real-world JavaCard programs are of (relatively) small size which of course makes them even more amenable to using formal methods for validation. Even so, in anticipation of future needs work should be done on investigating various ways of modularising and partitioning analyses so that analysis of a system might be done in a stepwise fashion. For applets that do not communicate via shared objects such partitioning should be fairly straightforward. For applets that do communicate, analyses for determining communication interfaces is needed; earlier work on "hardest attackers" for mobile ambients, cf. [9], may be a suitable basis for such work.

## References


1. Mikael Buchholtz, Hanne Riis Nielson, and Flemming Nielson. Experiments with Succinct Solvers. SECSAFE-IMM-002-1.0. Also published as DTU Technical Report IMM-TR-2002-4, February 2002.
2. Zhiqun Chen. *Java Card Technology for Smart Cards.* The Java Series. Addison Wesley, 2000.
3. Stephen N. Freund and John C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *ACM Conference on Object-Oriented Programming,*

*Systems, Languages, and Applications, OOPSLA'99*, pages 147–166, Denver, CO, USA, November 1999. ACM Press.

4. René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-003-1.0 (available from the author upon request), 2002.
5. René Rydhof Hansen. Flow Logic for Carmel. SECSAFE-IMM-001-1.5, 2002.
6. René Rydhof Hansen. Implementing the Flow Logic for Carmel. SECSAFE-IMM-004-DRAFT. Forthcoming, 2002.
7. Flemming Nielson. Flow Logic. Web page: `http://www.imm.dtu.dk/~nielson/FlowLogic.html`.
8. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
9. Flemming Nielson, Hanne Riis Nielson, and René Rydhof Hansen. Validating Firewalls using Flow Logics. *Theoretical Computer Science*, 283(2):381–418, 2002.
10. Flemming Nielson and Helmut Seidl. Control-Flow Analysis in Cubic Time. In *Proc. ESOP'01*, April 2001. Also appears as SECSAFE-DAIMI-006-1.0.
11. Flemming Nielson and Helmut Seidl. Succinct solvers. Technical Report 01-12, University of Trier, Germany, 2001.
12. Hanne Riis Nielson and Flemming Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. Lecture Notes in Computer Science. Springer Verlag, 2002. To appear.
13. SecSafe Home Page. `http://www.doc.ic.ac.uk/~siveroni/secsafe/`.
14. Igor Siveroni and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2, October 2001.
15. Igor Siveroni, Thomas Jensen, and Marc Eluard. A Formal Specification of the Java Card Firewall. In Hanne Riis Nielson, editor, *Proc. of Nordic Workshop on Secure IT-Systems, NordSec'01*, pages 108–122, Lyngby, Denmark, November 2001. Proceedings published as DTU Technical Report IMM-TR-2001-14.
16. Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. CC'92*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer Verlag, 1992.