Java Card Safety and Security through Static Analysis*

René Rydhof Hansen[1] and Igor Siveroni[2]

[1] Technical University of Denmark, Lyngby, Denmark
[2] Imperial College, London, UK
Email: rrh@imm.dtu.dk, siveroni@doc.ic.ac.uk

1 Introduction

Java Card is a new technology that enables Java applets to be loaded and executed in new generation smart cards and other devices with limited memory. Due to the limited possibilities for user intervention and interaction available to applications on a Java Card, or smart card in general, and the frequently sensitive or critical nature of such applications, it is important that these applications are as robust (safe) and secure as possible. We propose the use of program analysis techniques for the verification of safety and security properties of Java Card applications.

One important aspect regarding the safety of JC applets is that errors and exceptions, where possible, must be handled gracefully within the application, as it is often impossible for the user to assist or "help" the application recover after an error. Only exceptions of a certain nature should be reported back directly to the user (or interface) eg. security exceptions and irreparable exceptions. Under these circumstances, prediction of uncaught exceptions has high priority.

Java Card provides support for applet isolation and object sharing. The Java Card firewall implements applet isolation by partitioning the object system into separate object spaces, called contexts, and by limiting access only to objects that share the same context. This mechanism, by itself, would be too restrictive given the nature of Java Card applications. Java Card provides several mechanisms that allow applets to interact with each other and with the JCRE, such as the inclusion of global arrays or the use of `Shareable` interfaces. A `Shareable` interface - defined as part of the JC API - allows the user to define a set of shared interface methods which can be invoked from objects in any context. This feature opens the door to a series of situations where confidential information might be accessed by unauthorised users. The API provides a set of methods that allow certain degree of stack (context) and owner inspection. However, these checks prove to be insufficient [2] and static techniques are required to ensure secure JC applets.

We define a control and data flow analysis that includes exception handling and takes into account owner context information. This information can then be used to verify that a JC applet has a number of desirable safety and security properties.

* Partially funded by the EU project SecSafe (IST-1999-29075), cf.[11]

$$
\begin{array}{l}
o \in \mathsf{ClassInst} = \mathsf{Class} \times \mathsf{Owner} \times \mathsf{EntryPoint} \times (\mathsf{FieldID} \to (\mathsf{Val} \backslash \mathsf{RetAddr})) \\
a \in \mathsf{ArrayObject} = \mathsf{Type} \times \mathsf{Nat} \times \mathsf{Owner} \times \mathsf{Global} \times [(\mathsf{Val} \backslash \mathsf{RetAddr}] \\
\mathsf{GContext} = \mathsf{PackageAID} \cup \{\mathtt{JCRE}\} \quad \mathsf{Owner} = \mathsf{GContext} \times (\mathsf{AppletAid} \times \mathtt{JCRE}) \\
\mathsf{Inst} ::= \mathtt{nop} \mid \mathtt{push}\ t\ c \mid \mathtt{pop}\ t \mid \mathtt{swap}\ n_1\ n_2 \mid \mathtt{numop}\ t\ op \mid \mathtt{if}\ t\ cmp\ \mathtt{goto}\ addr \\
\qquad \mathtt{goto}\ addr \mid \mathtt{load}\ t\ i \mid \mathtt{store}\ t\ i \mid \mathtt{inc}\ t\ i\ c \mid \mathtt{new}\ \sigma \mid \mathtt{checkcast}\ \tau \\
\qquad \mathtt{instanceof}\ \tau \mid \mathtt{getstatic}\ f \mid \mathtt{putstatic}\ f \mid \mathtt{getfield}\ f \mid \mathtt{putfield}\ f \\
\qquad \mathtt{invokevirtual}\ m \mid \mathtt{invokedefinite}\ m \mid \mathtt{invokeinterface}\ m \mid \mathtt{return}\ t \\
\qquad \mathtt{throw} \mid \mathtt{jsr}\ addr \mid \mathtt{ret}\ i \mid \mathtt{arraylength} \mid \mathtt{arrayload}\ t \mid \mathtt{arraystore}\ t \\
t \in \{\mathtt{n,r,ra}\} \quad addr \in \mathsf{Addr} \quad c \in \mathsf{Nat} \cup \{\mathtt{null}\} \quad i \in \mathsf{Index} \quad n \in \mathsf{NWords}
\end{array}
$$

Fig. 1. Carmel Objects and Instruction Set

2 The JCVM and Carmel

Carmel [7] is an abstraction of the JCVML. It abstracts away certain complexities of the CAP file format e.g. the constant pool, and presents a significantly smaller instruction set while retaining the expressive power of the JCVML. The JCRE (modelled by $J \in \mathsf{JCRE}$) starts execution of the loaded Carmel program ($J.program \in \mathcal{P}(\mathsf{Package})$) by choosing the specified applet from the table of installed applets ($J.applets \in \mathsf{AppletAID} \to \mathsf{Ref}$) and invoking the JCVM.

Values and Objects. A value (Val) is either a number(Num), a reference (Ref) or a return address (RetAddr). Numeric values are integers while return addresses, introduced by the subroutine instructions, are elements of the domain Addr of program addresses. A non-null reference uniquely identifies an object in the heap H ($\mathsf{Heap} = \mathsf{Ref} \backslash \{\mathtt{null}\} \to \mathsf{Object}$). Objects (Figure 1) can be class instances or one-dimensional arrays. A class instance keeps information about the class of the object, its owner, the type of entry point and its field values. An array object keeps information about its element type, size, owner, global status and element values. In addition, the static heap K ($\mathsf{FieldID} \to \mathsf{Val} \backslash \mathsf{RetAddr}$) keeps track of the values associated to the static fields.

Firewall. The Java Card firewall partitions the object system into separated protected object spaces called group contexts (GContext). In general, it allows access only between objects that share the same context. Unauthorised access - as defined by the firewall - causes the JCVM to throw a `SecurityException`. A group context is either a package AID or the special `JCRE` context, which has irrestricted access privileges. An object is owned (Owner) by the applet that was active when it was instantiated and, together with the owning applet's AID, is assigned a group context: the AID of the package where the applet was declared. Thus, access between applets of the same package is always allowed. In addition, special JCRE-owned objects (JCRE entry points and global arrays) can be accessed from any context.

Semantics. The operational semantics of Carmel is defined as a relation between machine configurations. The running configuration $\langle J, K, H, SF{::}F\rangle$, with current frame $F = \langle own, m, pc, L, S\rangle$, describes the state of the machine before the execution of the instruction at program counter pc in method m with local stack S (Val^*) and local variable array L ($[\mathsf{Val}]$). The rest of the call stack SF (Frame^*) keeps track of the pending method calls (frames). We show a subset of the Carmel instruction set in Figure 1, which includes instructions for exception throwing and subroutines. The evaluation relation $\Rightarrow$ is defined by a series of rules that describe the execution of single instructions. For example, the rules:

$$\frac{\begin{array}{c} I = \texttt{invokevirtual}\ m' \wedge S = S_0{::}loc{::}v_1{::}\ldots{::}v_{|m'|} \wedge \mathit{checkNotNull}(loc) \\ o = H(loc) \wedge m_v = \mathit{methodLookup}(m', o) \wedge \mathit{checkInvoke}(I, own, o) \\ L' = loc{::}v_1{::}\ldots{::}v_n \ \wedge\ F' = \langle o.owner, m_v, 0, L', \epsilon\rangle \end{array}}{\langle J, K, H, SF{::}\langle own, m, pc, L, S\rangle\rangle \ \Rightarrow\ \langle J, K, H, SF{::}\langle own, m, pc, L, S_0\rangle{::}F'\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{invokevirtual}\ m' \wedge S = S_0{::}loc{::}v_1{::}\ldots{::}v_{|m'|} \\ e = \begin{cases} \texttt{NullPointerException} & \neg\mathit{checkNotNull}(loc) \\ \texttt{SecurityException} & \neg\mathit{checkInvoke}(I, own, o) \end{cases} \\ C' = J.\mathit{catchException}(\langle J, K, H, SF{::}\langle own, m, pc, L, S\rangle\rangle, e) \end{array}}{\langle J, K, H, SF{::}\langle own, m, pc, L, S\rangle\rangle \ \Rightarrow\ C'}$$

describe the successful evaluation of `invokevirtual` and the case when an exception is thrown. Instructions take their arguments from the operand stack S and local variables are accessed by specifying their position - $i \in \mathsf{Index}$ - in the local variable array L. As JCVM instructions, most Carmel instructions are typed and operate on values that have an expected runtime type (operand type t). However, since the JCVM works on verified code, no type-related runtime checks are performed during execution. Instead, only firewall violations (e.g. *checkInvoke*) - and conditions leading to the other `RuntimeExceptions` (e.g. *checkNotNull*) - are checked. The process of throwing an exception is described by a separate rule, as shown above. Once the type of exception has been determined, the *catchException* function is used to find the correct handler in the call stack. If no handler is found, the rule generates a special terminal configuration.

3 Analysing Carmel Programs

In this section we outline the specification of our analysis. It is based on a simple 0-CFA described in detail in [6].

The Abstract Domains. Carmel programs often initialise a number of fields at the beginning of the program to contain useful constants. Many of these are computed or derived from other constants, thus defeating a simple constant propagation analysis. The abstract domain for numbers chosen in our analysis reflects this situation and tracks constants through a certain (small) number of computations: $\overline{\mathsf{Num}} = (\mathbb{Z} \times \{0, \ldots, n\})^\top$.

Object references are abstracted into their corresponding class, the address of the `new` instruction that created it, and the owner of the object: $\overline{\mathsf{ObjRef}} = \mathsf{ClassName} \times \mathsf{Addr} \times \mathsf{Owner}$ similarly for arrays: $\overline{\mathsf{ArrRef}} = \mathsf{Type} \times \mathsf{Addr} \times \mathsf{Owner}$.

The possible values of a Carmel program are numbers and references (to objects and arrays) and return addresses: $\overline{\mathsf{Val}} = \overline{\mathsf{Num}} + \overline{\mathsf{ObjRef}} + \overline{\mathsf{ArrRef}} + \overline{\mathsf{RetAddr}}$ and abstract values are sets of values $\widehat{\mathsf{Val}} = \mathcal{P}(\overline{\mathsf{Val}})$.

Context sensitive analyses usually use invocation points, e.g. in the form of call strings, as context for the analysis. Instead we have chosen to use ownership information as context, which is essential for modelling the JCRE firewall. Thus our contexts keep track of two owners: the current owner and the last owner: $\overline{\mathsf{Context}} = \mathsf{Owner} \times \mathsf{Owner}$. Context information is recorded for each method in a so-called context-cache: $\mathsf{ContextCache} = \mathsf{Method} \to \overline{\mathsf{Context}}$.

Objects are modelled as a map from (instance) fields to abstract values: $\widehat{\mathsf{ClassInst}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$. In addition the the set of possible values in an array, we also keep track of the length of the array: $\widehat{\mathsf{ArrayObj}} = \widehat{\mathsf{Val}} \times \mathbb{N}_0^{\top}$.

The operand stack is modelled as a sequence of abstract values. For completeness $\top$ is used to represent infinitely high stacks. Note that bytecode verified programs can only produce stacks of finite height: $\widehat{\mathsf{Stack}} = \overline{\mathsf{Context}} \to \mathsf{Addr} \to (\widehat{\mathsf{Val}}^*)^{\top}$.

Modelling of local variables is straightforward as a map from variables to abstract values: $\widehat{\mathsf{LocHeap}} = \overline{\mathsf{Context}} \to \mathsf{Addr} \to \mathsf{Var} \to \widehat{\mathsf{Val}}$. Note that we keep track of both an abstract stack and an abstract local heap for *every* instruction. This is inspired by Freund and Mitchell's work on type systems for a subset of JVML [3] and allows for greater precision; however, if the extra precision is not needed, a less expensive analysis can trivially be obtained by modifying the domains to merge all stacks and variables for a given method or class or even for the entire program.

The heap is split into two parts, in correspondence with the semantics: one for dynamically allocated memory and one for static fields. The dynamic heap maps object (array) references to a corresponding abstract object (array): $\widehat{\mathsf{Heap}} = (\overline{\mathsf{ObjRef}} \to \widehat{\mathsf{ClassInst}}) + (\overline{\mathsf{ArrRef}} \to \widehat{\mathsf{ArrayObj}})$ whereas the static memory simply maps (static) fields to abstract values: $\widehat{\mathsf{StaHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$.

Since exceptions are really objects and are passed along as object referenced in the semantics, we simply model exceptions using abstract object references: $\widehat{\mathsf{Exception}} = \mathcal{P}(\overline{\mathsf{ObjRef}})$. Like context information, exception information is recorded for each method in an exception cache: $\widehat{\mathsf{ExceptionCache}} = \mathsf{Method} \to \widehat{\mathsf{Exception}}$.

Analysis Specification. The analysis itself is specified in the Flow Logic framework of Nielson and Nielson, cf. [9, 8]. The Flow Logic framework is a "specification approach" to static analysis, rather than an "implementation approach". Rather than detailing how a particular analysis is to be carried out, the Flow Logic framework is used to specify what it means for an analysis result (or rather a *proposed* analysis result) to be acceptable (correct) with respect to a program.

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

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

where $\hat{K} \in \widehat{\mathsf{StaHeap}}$, $\hat{H} \in \widehat{\mathsf{Heap}}$, $\hat{E} \in \widehat{\mathsf{ExceptionCache}}$, $\hat{O} \in \widehat{\mathsf{OwnerCache}}$ $\hat{L} \in \widehat{\mathsf{LocHeap}}$, $\hat{S} \in \widehat{\mathsf{Stack}}$, addr $\in$ Addr and instr is the instruction at addr. Intuitively the above states that the left-hand side is an *acceptable* analysis for the instruction instr at address addr. For every method, m, in the program we define a special address, (m, END_m), indicating the "exit" point of that method[3], which is used for handling return values.

For presentation purposes, we introduce a special *exception predicate* that is used to determine whether or not a given exception (modelled as an object reference) can be handled locally. If so, the stack and local variables are set up for the handler. If not, the exception is recorded in the global cache for unhandled exceptions thrown in that method. Such exceptions may be caught by later handlers. The predicate is formally defined in Appendix A.

Below we discuss the analysis specification for the invokevirtual instruction in detail, cf. Fig. 2. The specification for this instruction is by far the most involved; most of the other instructions need only a few lines of specification. Line

$$
\begin{array}{ll}
(\hat{K}, \hat{H}, \hat{E}, \hat{O}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual}\ m & \\
\quad \text{iff} \quad \forall \pi \in \hat{O}(m_0) : & (1)\\
\qquad A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}_\pi(m_0, pc_0) : & (2)\\
\qquad \forall (\text{Ref } \langle \sigma', (m', pc'), o' \rangle) \in B : & (3)\\
\qquad\quad m_v = \mathit{methodLookup}(m, \sigma') & (4)\\
\qquad\quad \pi' = \delta(o', \pi) & (5)\\
\qquad\quad \{(\text{Ref } \langle \sigma', (m', pc'), o' \rangle)\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}_{\pi'}(m_v, 1)[0..|m_v|] & (6)\\
\qquad\quad m.\text{returnType} \neq \texttt{void} \Rightarrow & (7)\\
\qquad\qquad T :: \_ \triangleleft \hat{S}_{\pi'}(m_v, \mathsf{END}_{m_v}) : (T :: X \sqsubseteq \hat{S}_\pi(m_0, pc_0 + 1)) & (8)\\
\qquad\quad m.\text{returnType} = \texttt{void} \Rightarrow (X \sqsubseteq \hat{S}_\pi(m_0, pc_0 + 1)) & (9)\\
\qquad\quad \forall (\text{Ref } \langle \sigma_X, (m_X, pc_X), o_X \rangle) \in \hat{E}_{\pi'}(m_v) : & (10)\\
\qquad\qquad \mathsf{P}^{\mathsf{EXC}}_{(\hat{L}, \hat{S}, \hat{E})}((\text{Ref } \langle \sigma_X, (m_X, pc_X), o_X \rangle), (m_0, pc_0), \pi) & (11)\\
\qquad\quad \neg \mathit{CheckInvoke}(\pi, \sigma', o') \Rightarrow & (12)\\
\qquad\qquad \mathsf{P}^{\mathsf{EXC}}_{(\hat{L}, \hat{S}, \hat{E})}((\text{Ref } \langle \sigma_{\texttt{SecurityException}}, (m_0, pc_0), \pi \downarrow_1 \rangle), (m_0, pc_0), \pi) & (13)\\
\qquad \hat{L}_\pi(m_0, pc_0) \sqsubseteq \hat{L}_\pi(m_0, pc_0 + 1) & (14)\\
\qquad \{\texttt{null}\} \subseteq B \Rightarrow & (15)\\
\qquad\quad \mathsf{P}^{\mathsf{EXC}}_{(\hat{L}, \hat{S}, \hat{E})}((\text{Ref } \langle \sigma_{\texttt{NullPointerException}}, (m_0, pc_0), \pi \downarrow_1 \rangle), (m_0, pc_0), \pi) & (16)
\end{array}
$$

Fig. 2. Flow Logic clause for invokevirtual

(1) makes sure that the instruction is analysed in all the possible instructions it

[3] This is similar to exit-nodes of control flow graphs.

may be executed. We introduce the notation used in line (2), "$A_1 :: \cdots :: A_{|m|} :: B :: X \triangleleft \hat{S}(m_0, pc_0)$:", as a succinct way of expressing that the abstract stack should be on the form $A_1 :: \cdots :: A_{|m|} :: B :: X$, ie. it should have at least $|m|+1$ elements. Furthermore, it acts as a binder for the variables $A_1, \ldots, A_{|m|}$, B, and X so that they may subsequently be referred to. With this in mind line (2) simply ensures that the right number of parameters (and an object reference) are present on the stack before invoking a method. For each of the object references found on the stack (line (3)), the parameters should be forwarded to the corresponding methods, found by a lookup function in line (4). Line (5) computes the context of the invoked method. Since we are interested modelling programs that make use of the (limited) stack inspection features of the JCRE, we define δ to only record the current context and the last context different from the current:

$$\delta(o, (o_1, o_2)) = \begin{cases} (o_1, o_2) \text{ if } o = o_1 \\ (o, o_1) \quad \text{if } o \neq o_1 \end{cases}$$

Instead of recording ownership contexts, the contexts could easily be changed, or augmented, to record invocation points like a more traditional n-CFA. Line (6) then copies the data on the stack to the invoked method, as local variables. Lines (7) and (8) handle the return value of the invoked method. This is done by copying what is on top of the stack at the end (represented by the special address END_{m_v}) of the invoked method. Line (9) is used for methods that do not return a value. Any exception thrown and not caught by the invoked method are passed back to the invoking method, this is handled in lines (10) and (11) using the exception predicate discussed previously. Lines (12) and (13) check if the current instruction, taking contexts into consideration, can possibly (try to) cross the firewall boundary and access a non-shared object, modelled by the predicate *CheckInvoke*, and if so give rise to a security exception. In line (14), since no local variables (of the invoking method) were changed, they are simply copied forward to the next instruction. Finally lines (15) and (16) check if the `null` reference can possibly occur on the stack in the position where there should be a reference to the object on which a method is invoked. If this is the case, then the current instruction may give rise to a null pointer exception, modelled and handled by the exception predicate.

Theoretical Properties. Following the approach in [10] it is straightforward to define *representation functions* for all of the abstract domains and extend these to semantic configurations, β_{Conf}. For lack of space we shall not go into further details here. The representation functions are then used to establish a subject reduction property for the analysis:

Theorem 1. *If* $\mathcal{A} \models P$ *then* $(\beta_{\text{Conf}}(C) \sqsubseteq \mathcal{A} \wedge \frac{I=\texttt{instr}}{C \Rightarrow C'}) \Longrightarrow \beta_{\text{Conf}}(C') \sqsubseteq \mathcal{A}$

Proof (sketch). By case analysis using a technical lemma guaranteeing that the call stack is well-formed, i.e. that it only contains frames from methods that have been invoked.

A detailed proof of a simpler 0-CFA version of the analysis can be found in [6]. Parts of that proof have been independently implemented and verified using the Coq Proof Assistant.

4 Verifying Safety and Security Properties

In this section we sketch how the analysis described in previous section can be used to verify a number of properties about JCVML programs.

Interestingly, many of the safety and security properties that are useful in a Carmel program can be guaranteed by showing that the program will never throw certain exceptions. This suggests that simply inspecting the exception part of our analysis can be used for validating a wide range of properties, including (but not limited to) the following: *(1)* no divison by zero, *(2)* no dereferencing of null pointers, *(3)* no violations of array bounds, *(4)* no firewall violations.

In addition, our analysis is useful for validating a number of properties that are not directly connected to an exception:

No leaks. In order to overcome the deficiencies of the JCRE firewall, cf. [2], applets use a limited version of *stack inspection* (based on ownership contexts) to ensure that they are accessed only by properly authorised applets. Since our analysis faithfully models the ownership information of applets, it can be used to verify that a shared object is not accessed by a non-authorised, malicious applet thereby ensuring that confidential data is not leaked or destroyed. This is similar to the analysis of [2].

Escape analysis. Since abstract object references carry information on where they were created, a simple escape analysis can be performed. This allows for "non escaping" objects to be stack allocated, cf. [1, 12].

Pointer confinement. Related to escape analysis is the notion of *pointer confinement*, cf. [4]. Our analysis can certainly be used for verifying that given classes are confined within their defining package in much the same way as it can be used for escape analysis.

Finally, the analysis can be used for more traditional purposes, e.g. the data flow analysis is sufficient to support a constant folding program transformation.

Note that all of the above properties can be checked from the analysis result as it is. Simple additions and modifications allow for an even wider range of properties, e.g. in [5] the analysis is extended with a component to verify that transactions are well-formed.

5 Conclusions

In this paper we have outlined a formal semantics for Carmel, a rational reconstruction of Java Card Virtual Machine Language, that forms the necessary basis for any formal reasoning about Carmel programs. Furthermore we have shown how a simple control flow analysis for Carmel can easily be extended to include exceptions, data flow, and ownership information; and how the information obtained by such an analysis can be used to verify a wide range of relevant safety and security properties, thus supporting our main position that static analysis can indeed be a very useful tool for verifying the safety and security of Carmel programs.

A further advantage of the static analysis approach, one which we do not have the space here to develop, is that it is often straightforward to implement the analysis and thereby bridging the gap between theory and practise. We have implemented a prototype of the simple 0-CFA and ownership analysis. Early experiments with a demo-applet (consisting of approximately 6500 lines of Carmel code spread over 50+ classes and implementing an electronic purse) suggests that even a naïve implementation is efficient enough to handle realistic applets.

References

1. Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'99*, pages 20–34, 1999.
2. Marc Élouard and Thomas Jensen. Secure object flow analysis for Java Card. In *Proc. of Smart Card Research and Advanced Application Conference, Cardis'02*, 2002.
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, 1999.
4. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'01*, pages 241–255, 2001.
5. René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-003-1.0, 2002.
6. René Rydhof Hansen. Flow Logic for Carmel. SECSAFE-IMM-001-1.5. Available from SecSafe web page, 2002.
7. Renaud Marlet. Syntax of the JCVM Language To Be Studied in the SecSafe Project. SECSAFE-TL-005-1.7. Available from SecSafe web page, May 2001.
8. Flemming Nielson. Flow Logic. Web page: `http://www.imm.dtu.dk/~nielson/FlowLogic.html`.
9. Hanne Riis Nielson and Flemming Nielson. Flow Logic: A Multi-paradigmatic Approach to Static Analysis. In *The Essence of Computation – Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer Verlag, 2002.
10. Flemming Nileson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
11. Igor Siveroni. SecSafe. Web page: `http://www.doc.ic.ac.uk/~siveroni/secsafe/`.
12. John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'99*, pages 187–206, 1999.

A The Exception Predicate

The exception predicate is formally defined as follows:

$$
\begin{array}{l}
\mathsf{P}^{\mathsf{EXC}}_{(\hat{L},\hat{S},\hat{E})}((\mathrm{Ref}\ \langle \sigma_X,(m_X,pc_X),o_X\rangle),(m_0,pc_0),\pi_0) \equiv \\
\quad \mathsf{findHandler}(m_0,pc_0,\sigma_X) = \bot \ \Rightarrow \\
\qquad (\mathrm{Ref}\ \langle \sigma_X,(m_X,pc_X),o_X\rangle) \in \hat{E}_{\pi_0}(m_0) \\
\quad \mathsf{findHandler}(m_0,pc_0,\sigma_X) = L_X \ \Rightarrow \\
\qquad \{(\mathrm{Ref}\ \langle \sigma_X,(m_X,pc_X),o_X\rangle)\} \sqsubseteq \hat{S}_{\pi_0}(m_0,L_X) \wedge \\
\qquad \hat{L}_{\pi_0}(m_0,pc_0) \sqsubseteq \hat{L}_{\pi_0}(m_0,L_X)
\end{array}
$$

It checks if there is an exception handler defined for the particular exception type thrown in a particular region. If there is, then the operand stack and local variables are forwarded to the handler. Otherwise the exception is entered into the exception cache.