A Type System for Checking Applet Isolation in Java Card

Peter Müller and Arnd Poetzsch-Heffter

FernUniversität Hagen, 58084 Hagen, Germany,
{Peter.Mueller, Arnd.Poetzsch-Heffter}@Fernuni-Hagen.de

Abstract. A Java Card applet is, in general, not allowed to access fields and methods of other applets on the same smart card. This *applet isolation* property is enforced by dynamic checks in the Java Card Virtual Machine. This paper describes a refined type system for Java Card that enables mostly static checking of applet isolation. With this type system, most firewall violations are detected at compile time.

1 Introduction

The Java Card technology allows applications written in a subset of Java—so-called Java Card applets—to run on smart cards [Che00]. Several applets can run on a single card and share a common object store. Since the applets on a card may come from different, possibly untrusted sources, a security policy ensures that an applet, in general, cannot inspect or manipulate data of other applets. To enforce this *applet isolation* property, the Java Card Virtual Machine establishes an *applet firewall*, that is, it performs dynamic checks whenever an object is accessed, for example, by field accesses, method invocations, or casts. If an access would violate applet isolation, a `SecurityException` is thrown.

Dynamically checking applet isolation is unsatisfactory for two reasons: (1) It leads to significant runtime overhead. (2) Accidental attempts to violate the firewall are detected at runtime, that is, after the card with the defective applet has been issued, which could lead to enormous costs. In this paper, we sketch a refined type system for the Java Card language that allows one to detect most firewall violations statically by checks on the source code level. This type system serves three important purposes:

1. It reduces the runtime overhead caused by dynamic checks significantly.
2. Most firewall violations are detected at compile time. At runtime, only certain casts can lead to `SecurityExceptions`. These casts point programmers and verifiers at the potentially critical spots of a program.
3. The refined type information provides formal documentation of the kinds of objects handled in a program such as entry point objects, global arrays, etc., and complements informal documentation, especially, of the Java Card API.

Overview. In the remainder of this introduction, we describe the applet firewall and explain our approach. Section 2 presents the refined type system. The context conditions that replace dynamic checking of applet isolation are explained in Section 3. We offer some conclusions in Section 4.

1.1 Applet Firewall

The applet firewall essentially partitions the object store of a smart card into separate protected object spaces called *contexts* [Sun00, Sec. 6]. The firewall is the boundary between one context and another. It allows object access across contexts only in certain cases. In this subsection, we describe contexts, object access across contexts, and the dynamic checks that enforce the firewall.

Contexts. There is one context for each applet installed on a smart card (we neglect group contexts for brevity). The context for an applet `A` contains all `A` objects and all objects created by methods executed on objects in that context. The operating system of the card is contained in the Java Card Runtime Environment (JCRE) context. At any execution point, there is exactly one *currently active context* (in instance methods, this context contains `this`). When an object of context C invokes a method m on an object in context D, a *context switch* occurs, that is, D becomes the new currently active context. Upon termination of m, C is restored as the currently active context

Class objects do not belong to any context. There is no context switch when a static method is invoked. Objects referenced by static fields are ordinary objects, that is, they belong to an applet or to the JCRE context.

Firewall Protection. We say that an object is *accessed* if it serves as target for a field access or method invocation, or if its reference is used to evaluate a cast or `instanceof` expression (we do not treat exceptions and arrays here). In general, an object can only be accessed if it is in the currently active context (see below for exceptions to this rule). To enforce this rule, the Java Card Virtual Machine performs dynamic checks. If an object is accessed that is not in the currently active context, a `SecurityException` is thrown.

Object Access Across Contexts. The Java Card applet firewall allows certain forms of object access across contexts: (1) Applets need access to services provided by the JCRE. These services are provided by *JCRE entry point objects*. These objects belong to the JCRE context, but can be accessed by any object. In this paper, we only consider permanent entry point objects (PEPs for short). An extension to temporary entry point objects and global arrays is straightforward. (2) To support cooperating applets, applets can interact via *shareable interface objects* (SIOs for short). An object is an SIO if its class implements the `Shareable` interface. An applet can get a reference to an SIO of another applet by invoking a static method of the JCRE. It can then invoke methods on this SIO [Sun00, Sec. 6]. (3) The JCRE has access to objects in any context.

Example. In the following, we explain the dynamic checks for an invocation `e.m(...)` of a dynamically-bound method. We assume that `T` is the compile time type of `e` and that the evaluation of `e` yields an object X. Before the

invocation is executed, it is checked whether at least one of the following cases applies[1]. If no case applies, a `SecurityException` is thrown.

D1: X is in the currently active context;
D2: X is an entry point object and `T` is a class;
D3: `T` is an interface that extends `Shareable`;
D4: The JCRE context is the currently active context.

We illustrate these checks by a faulty implementation of two cooperating applets. Fig. 1 shows the implementation of a client applet. We assume that the client and a server applet are installed on the same card. The following interaction is initiated by method `Client.process`: The client requests an SIO from the server by invoking `JCSystem.getAppletShareableInterfaceObject`, which yields an SIO that is cast to the shareable interface `Service`. The client then invokes `doService` on the SIO. This invocation yields a new `Status` object that is used to check whether the service was rendered successfully.

In our implementation, this interaction leads to a `SecurityException`: The client and server applets reside in different contexts. The `Service` SIO and the `Status` object belong to the context of the server. Since `Status` does not implement `Shareable`, the `Status` object is not an SIO. When the invocation `sta.isSuccess()` is checked as explained above, cases **D1**–**D4** do not apply. Thus, the access is denied and the exception is thrown. To correct this error, one would have to use an interface that extends `Shareable` instead of class `Status`; then case **D3** would apply.

1.2 Approach

To detect firewall violations at compile time, we adapt type systems for alias control such as ownership types and universes [CPN98,MPH01,Mül01]. Whereas these type systems focus on restricting references between different contexts, we permit references between arbitrary contexts, but restrict the operations that can be performed on a reference across context boundaries.

Our type system augments every reference type of Java with context information that indicates (1) whether the referenced object is in the currently active context, (2) whether it is a PEP, or (3) whether it can belong to any context. Type rules guarantee that every execution state is well-typed, which means especially that the context information is correct.

We use downcasts to turn references of kind (3) into references of more specific types. For such casts, dynamic checks guarantee that the more specific type is legal. Otherwise, a `SecurityException` is thrown.

To check an applet with our type system, its implementation as well as the interfaces of applets it interacts with and of the Java Card API must be enriched

[1] The checks correspond to the rules explained in the above paragraphs. We have adopted them from [Sun00, Sec. 6.2.8] although **D2**'s requirement that `T` be a class seems overly restrictive. Please refer to [Sun00] for a more detailed explanation of the checks.

```
public class Status {
  private boolean success;
  public Status(boolean b)   { success = b; }
  public boolean isSuccess() { return success; }
}

public interface Service extends Shareable {
  Status doService();
}

public class Client extends Applet {
  private Client() { register(); }
  public static void install(byte[] a, short o, byte l) { new Client(); }

  public void process(APDU apdu) {
    AID     svr = ...;              // server's AID
    Shareable s = JCSystem.getAppletShareableInterfaceObject(svr,(byte)0);
    Service ser = (Service)s;       // cast is legal
    Status  sta = ser.doService();  // invocation is legal
    if (sta.isSuccess())            // leads to SecurityException
      ...
  }
}
```

Fig. 1. Implementation of a client applet. All classes are implemented in the same package. `package` and `import` clauses are omitted for brevity. We assume that a server applet is implemented in a different package.

by refined type information. This information is used to impose additional context conditions for field accesses, method invocations, casts, and `instanceof` expressions that guarantee that the firewall is respected.

In the execution of a program that is type correct according to our type system, only the evaluation of downcast expressions requires dynamic firewall checks and might lead to `SecurityExceptions`. Thus, casts point programmers at the critical spots in a program, which simplifies code reviews and testing. Moreover, they allow standard reasoning techniques to be applied to show that no `SecurityException` occurs.

In theory, our type system can replace almost all dynamic firewall checks. However, if only some applets on a card are checked by our type system, the dynamic checks have to stay in place to prevent applets from untrusted sources from violating the firewall. Still, our type system is useful to detect possibly fatal program errors at compile time, which simplifies reasoning and reduces costs.

In the following sections, we present the refined type system and some of the additional context conditions. For brevity, we focus on a subset of Java Card and omit exceptions and arrays. Moreover, we do not treat temporary entry points and global arrays. An extension of our work to these features is straightforward.

2 The Type System

A type system expresses properties of the values and variables of a programming language that enable static checking of well-definedness of operations and their application conditions, in this case, Java Card's firewall constraints.

Tagged Types. In order to know whether an operation is legal in Java Card, we need information about the context in which the operation is executed. The basic idea of our approach is to augment reference types by context information.

In this paper, we are only interested in checking applet code and do not consider the JCRE implementation. Thus, statements and expressions are either executed in an applet context (in case of instance methods) or outside all contexts (in case of static methods). From the point of view of an applet context C, we can distinguish (a) internal references to objects in C, (b) PEP references, and (c) external references to objects in applet contexts different from C or to non-PEP objects in the JCRE.

In the type system, we reflect this distinction by the *context tags* `i` for internal, `p` for PEP, and `a` for any. The `a`-tag expresses the fact that it is not known whether a reference is internal, PEP, or external. A special tag for external is dispensable, because all operations that are allowed on external references are allowed on "any" reference. Since static class members do not belong to any context, the `i`-tag must not be used for types of static fields or in static methods. Let TypeId denote the set of declared type identifiers of a given Java Card program; then the tagged type system comprises the following types:

$$\text{TaggedType} = \{\text{booleanT}, \text{intT}, ..., \text{nullT}\} \cup (\{\texttt{i}, \texttt{p}, \texttt{a}\} \times \text{TypeId})$$

Except for the null-type that is used to type the `null` literal, all reference types in the tagged type system are denoted as a pair of a tag and a Java type. The subtype relation $\preceq$ on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where G is a tag, $S, T \in \text{TypeId}$, and $\preceq_J$ denotes the subtype relation on TypeId:

$$(G,S) \preceq (G,T) \Leftrightarrow S \preceq_J T \qquad (G,T) \preceq (\texttt{a},T) \qquad \text{nullT} \preceq (G,T)$$

Tagged Type Rules. We illustrate our approach by presenting the most interesting rules for the tagged type system. Since the type rules for statements are trivial, we focus on expressions. A type judgment of the form $E \vdash e :: TT$ means that expression e has tagged type TT in the declaration environment E of the method enclosing e.

The type judgment for instance creation expressions without parameters is $E \vdash \text{new T}() :: (\texttt{i}, \text{T})$. The `i`-tag indicates that the created object belongs to the currently active context.

The tagged type of a cast expression is the type (H,T) appearing in the cast operator (see below). For simplicity, we only consider downcasts, that is, (H,T) has to be a subtype of the expression type. Note that we allow a reference tagged

"any" to be cast into an internal or PEP reference. Recall that dynamic checks guarantee that the more specific tag is appropriate (see Subsec. 1.2)

$$\frac{E \vdash e :: (G,S) \quad , \quad (H,T) \preceq (G,S)}{E \vdash (\,(H,T)\,)\, e :: (H,T)}$$

The most interesting and complex rule is the one for invocation expressions. For simplicity, we assume that methods have exactly one parameter of tagged type (F_P, T_P) and a (tagged) return type (F_R, T_R):

$$\frac{E \vdash e1 :: (H,T) \quad , \quad E \vdash e2 :: (G,S) \quad , \quad (H,T) * (G,S) \preceq (F_P, T_P)}{E \vdash e1.m(e2) :: (H,T) * (F_R, T_R)}$$

The operator $* : \text{TaggedType} \times \text{TaggedType} \to \text{TaggedType}$ is defined as follows: $(H,T) * (G,S) = (\mathtt{a}, S)$, if $H \neq \mathtt{i}$ and $G = \mathtt{i}$; $(H,T) * (G,S) = (G,S)$ in all other cases. The $*$-operator tags the parameter or result as "any" when the invocation could lead to a context switch ($H \neq \mathtt{i}$) and an internal reference is passed to or returned by the method ($G = \mathtt{i}$). This is necessary since an internal reference is external to the new currently active context, as illustrated by the following example.

Fig. 2 shows the `Service` interface and the `Client` class with tagged type information. The return type of `Service.doService` is internal since the method creates a new `Status` object in the context in which it is executed (the context of the server applet). When `doService` is invoked from the client context (see method `Client.process`, Fig. 2), the returned `Status` object is external to the client context and must, thus, be tagged "any". This adaption of the tag is described by the $*$-operator.

Type Safety. We proved that the tagged type system is type safe in the following sense: If the evaluation of an expression e starts in a type correct state, it yields a type correct state upon termination and the resulting value belongs to the tagged type of e. Informally, this property states that all references are correctly tagged, which is a prerequisite for statically checking applet isolation (see Sec. 3). The type safety proof is based on an operational semantics the states of which, in particular, contain a variable that keeps track of the currently active context.

The type safety proof runs by rule induction. An interesting aspect is the treatment of context information and their relation to tagged types. Each class instance "knows" the context it belongs to and whether it is a PEP. Based on this information, we can assign a tagged type to an object X in a context C ($\mathit{ctyp}(X)$ denotes the TypeId of X's class):

$$\mathit{ttyp} : \text{Object} \times \text{Context} \to \text{TaggedType}$$

$$\mathit{ttyp}(X,C) = \begin{cases} (\mathtt{p}, \mathit{ctyp}(X)), \text{ if } X \text{ is a PEP} \\ (\mathtt{i}, \mathit{ctyp}(X)), \text{ if the context of } X \text{ is } C \text{ and } X \text{ is not a PEP} \\ (\mathtt{a}, \mathit{ctyp}(X)), \text{ otherwise} \end{cases}$$

3 Checking Applet Isolation

Tagged types provide a conservative approximation of runtime context information. This information can be used to impose *static checks* that guarantee that an applet respects the applet firewall at runtime. In the following, we present these checks for method invocations and argue why they enforce applet isolation.

Static Checks. We perform the following static checks for each invocation statement of the form `e.m(...)`, where (H, T) is the static tagged type of `e`. *class(T)* yields whether T denotes a class.

S1: $H = \texttt{p} \Rightarrow$ *class(T)*
S2: $H = \texttt{a} \Rightarrow \neg$*class(T)* $\wedge\ T \preceq_J$ `Shareable`

To show that these static checks prevent `SecurityExceptions`, we explain for each possible tag of `e`'s type that one of the cases **D1**–**D3** of the dynamic checks presented in Sec. 1 applies. (**D4** is not relevant here since we only consider applet code, not the implementation of the JCRE.) We assume that `e` evaluates to an object X:

$H =$ `i`: Well-typedness guarantees that X is in the currently active context; that is, case **D1** applies.
$H =$ `p`: Well-typedness guarantees that X is a PEP, and static check **S1** guarantees that T is a class; thus, case **D2** applies.
$H =$ `a`: Static check **S2** guarantees that T is an interface that extends `Shareable`; thus, case **D3** applies.

In summary, well-typedness and the above static checks guarantee that execution of a method invocation does not violate the firewall at runtime. Therefore, the checks can be used to enforce applet isolation statically. The static checks for other expressions are analogous.

We proved the following *firewall lemma*: Each Java Card program with tagged types that passes the static checks behaves like the corresponding Java Card program with dynamic checks. That is, every Java Card program that can be correctly tagged does not throw `SecurityExceptions` (except for the checks for casts). The proof of the firewall lemma is based on two operational semantics: one semantics with dynamic checks, the other without them. It runs by rule induction and exploits type safety of the tagged type system and the static firewall checks.

Example. Fig. 2 shows `Client`'s `process` method with tagged type information. Since `getAppletShareableInterfaceObject` is a static method that does, in general, not return a PEP, its return type is `any Shareable`. The `any`-tag carries over to `sta` (the status is extern). Therefore, the invocation `sta.isSuccess()` does not pass static check **S2** since `Status` is a class that does not implement `Shareable`. That is, the firewall violation caused by this invocation would be detected at compile time.

```
public interface Service extends Shareable {
  intern Status doService();
}

public class Client extends Applet {
  ...
  public void process(any APDU apdu) {
    intern AID    svr = ...;     // server's applet id is intern
    any Shareable   s =
                JCSystem.getAppletShareableInterfaceObject(svr,(byte)0);
    any Service   ser = (any Service)s;     // ser is in general extern
    any Status    sta = ser.doService();    // sta is also extern
    if (sta.isSuccess())                    // static check fails
      ... }
  ...
}
```

Fig. 2. `Service` interface and `Client` class with tagged type information. In the concrete syntax, we use the keywords `intern`, `pep`, and `any` as tags.

4 Conclusions

We presented a refined type system for Java Card that allows one to check applet isolation statically. The type system is based on the more general approach of universe types that was developed for the modular verification of Java programs. Our future goal is to formally verify interesting properties of Java Card applets such as the absence of `SecurityExceptions`. To show this property, we would check the applet by the tagged type system and prove benevolence of the casts. Since we are interested in source code verification, we designed our checking technique for the source level, whereas other approaches focus on the byte code level [BCG+00,CHS01]. It could be adapted to the byte code level as well.

References

[BCG+00] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS*, 2000.

[Che00] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[CHS01] D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java Card sharing. Available from `www-sop.inria.fr/oasis/personnel/Ludovic.Henrio/JavaCardSharingAnalysis.ps.gz`, 2001.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[Sun00] Sun Microsystems, Inc. *The Java Card 2.1.1 Runtime Environment (JCRE) Specification*, May 2000.

Biographies

Peter Müller is a member of the VerifiCard project that is concerned with the formal verification of Java Card applets. He recently received a Doctorate in Computer Science from FernUniversität Hagen with a thesis on *Modular Specification and Verification of Object-Oriented Programs.*

Arnd Poetzsch-Heffter is associate professor at FernUniversität Hagen. He received a Doctor in Computer Science from the Technical University of Munich in 1991 with a thesis about programming language specification. During his postdoc year at Cornell University, he began research on the integration of program specification and verification techniques for OO-programs. In his Habilitation thesis, he developed the formal foundations for such an integration. Currently, he is especially engaged in the development of specification and verification techniques and tools for OO-programs.