A Type System for Checking Applet Isolation in Java Card

Werner Dietl[1], Peter Müller[1], and Arnd Poetzsch-Heffter[2]

[1] ETH Zürich, Switzerland
{Werner.Dietl,Peter.Mueller}@inf.ethz.ch
[2] Technische Universität Kaiserslautern, Germany
poetzsch@informatik.uni-kl.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 static checking of applet isolation. With this type system, firewall violations are detected at compile time. Only a special kind of downcast requires dynamic checks.

1 Introduction

The Java Card technology allows applications—so-called Java Card applets—to run on smart cards. 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 `SecurityException`s. These casts occur when static fields are accessed and when a reference is passed to another applet and then retrieved again. Programmers and verifiers can focus on these cast expressions when reasoning about applet isolation.
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.

In this paper, we are only interested in checking applet code and do not consider the Java Card Runtime Environment (JCRE) implementation.

Overview. In the remainder of this introduction, we describe the applet firewall and our approach. Section 2 presents the refined type system, which tracks context information statically. Type safety is proved in Section 3. Based on the refined type information, context conditions can check applet isolation statically. These static checks are explained in Section 4. We discuss the presented and related work in Sections 5 and 6.

1.1 Applet Firewall

The applet firewall essentially partitions the object store of a smart card into separate protected object spaces called *contexts* [24, Sec. 6]. 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. Each applet installed on a smart card belongs to exactly one applet context. This context is determined by the package in which the applet class is declared. It contains the applet objects and all objects created by method executions 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. Static fields can be accessed from any context. Objects referenced by static fields belong to an applet context or to the JCRE context.

Firewall Protection. We say that an object is *accessed* if it serves as receiver for a field access, array element access, or method invocation, if its reference is used to evaluate a cast or `instanceof` expression, or if the object is an exception that is thrown. In general, an object can only be accessed if it is in the currently active context (see below for object access across contexts). 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. There are *permanent entry point objects* (PEPs for short), *temporary entry point objects* (TEPs for short), and *global arrays*. Global arrays share many properties of TEPs: References to global arrays and TEPs cannot be stored in fields. An applet can invoke methods on entry point objects, but not access their fields.

(2) Interaction between applets is enabled by *shareable interface objects* (SIOs for short). An object is an SIO if its class transitively implements the `Shareable` interface. An applet can get a reference to an SIO of another applet by invoking a static method of the JCRE. Access to SIOs is severely restricted. An applet can invoke those methods of an SIO which are declared in an interface that extends `Shareable`. However, it can neither access fields of SIOs nor cast an SIO to a type other than a shareable interface [24].

(3) The JCRE has access to objects in any context.

Example. The following faulty implementation of two cooperating applets illustrates the dynamic checks of the applet firewall. Fig. 1 shows the implementation of a client applet. We assume that the client and a server applet are installed on the same card, but are contained in different packages.

```
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 {
  public void process(APDU apdu) {
    AID    svr  = ...;                    // server's AID
    Shareable s = JCSystem.getAppletShareableInterfaceObject(svr, (byte)0);
    Service ser = (Service)s;             // legal cast: s refers to a Service object
    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.

The following interaction is initiated by method `process`: By invoking the static method `JCSystem.getAppletShareableInterfaceObject`, the client requests an SIO from the server. This call returns 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.

This interaction leads to a `SecurityException`: The client and server applet reside in different contexts. The `Service` SIO and the `Status` object belong to the context of the server. When the invocation `sta.isSuccess()` is checked, none of the three cases for object access across contexts applies: (1) The `Status` object is not an entry point object. (2) Since `Status` does not implement `Shareable`, the `Status` object is not an SIO. (3) Since method `process` is executed on an `Applet` object, the currently active context is an applet context, not the JCRE context. Therefore, 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`.

1.2 Approach

To detect firewall violations at compile time, we adapt ownership type systems for alias control [11, 17, 19]. 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, (3) whether it is a TEP or a global array, or (4) whether it can belong to any context. Type rules guarantee that every execution state is well-typed, which means, in particular, that the context information is correct. We use downcasts to turn references of kind (4) 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 by refined type information. This information is used to impose additional context conditions on expressions to 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 `SecurityException`s. 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 [23].

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.

2.1 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 with context information.

Since we are interested in checking applet code, we consider statements and expressions that are executed in an applet context. From the point of view of an applet context, C, we can distinguish (a) internal references to objects in C, (b) PEP references, (c) TEP references including global arrays, and (d) references to objects in any context.

In the type system, this distinction is reflected by the *context tags* `i` for internal, `p` for PEP, `t` for TEP and global arrays, and `a` for any. The `a`-tag is used for references to non-TEP objects in contexts that are not known statically. For checking applet isolation, it would be desirable to have more precise information about the context of an object. However, the sharing mechanism through method `JCSystem.getAppletShareableInterfaceObject` does not provide any static information about the context of the returned SIO.

A tagged type is either a simple tagged type for primitive values or class instances, or a tagged array type.

Simple Tagged Types. Let *TypeId* denote the set of declared type identifiers of a given Java Card program; then the tagged type system comprises the following types for primitive values and class instances:

$$SimpleTaggedType = \{booleanT, intT, \ldots, nullT\} \cup (\{\mathtt{i}, \mathtt{p}, \mathtt{t}, \mathtt{a}\} \times TypeId)$$

Except for the null-type, which 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. In actual code examples we will use the keywords `intern`, `pep`, `tep`, and `any` instead of the symbols used for the formalization.

Tagged Types. In general, an array type has two tags: The *array tag* specifies the context that contains the array object, whereas the *element tag* specifies the context of the array elements relatively to the context of the array object. For instance, an array of type `intern any Object[]` belongs to the currently active context and stores objects belonging to any context.

Global arrays serve as temporary entry points to the JCRE context. Therefore, we use the `tep` tag to mark an array as global. For instance, the `APDU` buffer, a global array of bytes, has type `tep byte[]`. Since the element type is a primitive type here, there is no element tag.

Formally, a tagged type is either a simple tagged type or an array type. Since Java Card does not provide multi-dimensional arrays, the array elements have a simple tagged type. Permanent entry point arrays do not exist in Java Card.

$$TaggedType = SimpleTaggedType \cup (\{\mathtt{i}, \mathtt{t}, \mathtt{a}\} \times SimpleTaggedType)$$

Notation. In the following, the meta-variables S and T denote Java types; TS and TT range over *TaggedType*. Calligraphic $\mathcal{S}$ and $\mathcal{T}$ can stand for Java types or *TaggedType*s. It is often convenient to use a tuple notation for tagged reference types. (γ, T) is the simple tagged type for objects of Java type T with tag γ. (γ, TT) is a tagged array type with element type TT. $(\gamma, \mathcal{T})$ can be the type of both a class instance or an array.

Subtyping on Tagged Types. The subtype relation $\preceq$ on tagged types follows Java's subtype relation $\preceq_J$ on Java types. It is the smallest reflexive and transitive relation satisfying the following axioms.

(1) $(\gamma, \mathcal{T}) \preceq (\gamma, \texttt{java.lang.Object})$
(2) $nullT \preceq (\gamma, \mathcal{T})$
(3) $(\texttt{i}, \mathcal{T}) \preceq (\texttt{a}, \mathcal{T})$
(4) $(\texttt{p}, \mathcal{T}) \preceq (\texttt{a}, \mathcal{T})$
(5) $(\gamma, S) \preceq (\gamma, T) \Leftrightarrow S \preceq_J T$
(6) $(\gamma, (\delta, S)) \preceq (\gamma, (\delta, T)) \Leftrightarrow S \preceq_J T$

Every reference type is a subtype of the tagged type for `Object`, provided that both types have the same tag (1). The null-type is a subtype of any tagged reference type (2). `intern` and `pep` types are subtypes of the corresponding `any` type (3,4). Note that there is no such axiom for `tep` types. `tep` types must not be subsumed under `any` types to prevent TEPs from being stored in fields or arrays (see Section 4.1). Two tagged types with the same tag are subtypes iff the corresponding Java types are subtypes (5). Covariant array subtyping requires runtime checks for each array update. For tagged types, these checks would involve context information and could throw `SecurityException`s. To avoid such checks, we allow only limited covariant subtyping of tagged array types. Two tagged array types can only be subtypes if they have the same element tag (6). That is, covariant subtyping is only possible in terms of Java types, but not of tags. For instance, if S is a subtype of T then `intern intern S` is a subtype of `intern intern T`, but not of `intern any S`.

Since Java Card imposes weaker restrictions on PEPs than on TEPs, we could allow `pep` types to also be subtypes of the corresponding `tep` types, and forbid downcasts from `tep` to `pep`. We omitted this subtype relation for simplicity.

Casts. Casts on tagged types work analogously to Java. A downcast can be used to specialize the tagged type of an expression, in particular, the context information. For instance, an expression of type $(\texttt{a}, T)$ can be cast to $(\texttt{i}, T)$. A runtime check ensures that the refined context information is correct. If not, a `SecurityException` is thrown.

Example. Fig. 2 shows the `Service` interface and the `Client` class from Fig. 1 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, the returned `Status` object is external to the client context and must, thus, be tagged `any`. The type rules that enforce these tags are discussed in the next subsection. The static checks that detect the firewall violation are presented in Sec. 4.

```
public interface Service extends Shareable {
  intern Status doService();
}
public class Client extends Applet {
  public void process(tep APDU apdu) {
    pep AID      svr  = ...;                    // server's applet id is a PEP
    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 firewall check fails
      ...
  }
}
```

Fig. 2. `Service` interface and `Client` class with tagged types.

2.2 Tagged Type Rules

In the tagged type system, a type judgment of the form $\vdash e :: TT$ means that expression e is well-typed and has tagged type TT. $\vdash s$ expresses that statement s is well-typed. In the formalization, we omit the declaration environment and all rules that handle the environment. Instead, we use $[f]$, TP, and TR to denote the tagged type of a field f, the (sole) parameter type, and the return type of a method, respectively. A more complete formalization of Java's type system including declaration environments is presented in [20, 22, 12].

Fig. 3 shows the most interesting rules of the tagged type system. Since the type rules for statements are trivial, we focus on expressions here. In the type rules, premises marked by $(\star)$ are only needed for static checks of applet isolation. These premises will be discussed in Section 4.1. The function *ShareItf?* yields whether the argument is an interface that extends `Shareable`.

For brevity, we do not present the rules for exceptions. Like all reference types, exceptions are tagged. For `throw` and `try` statements, as well as for the declaration of exceptions in method signatures, the normal Java rules apply based on the subtyping of tagged types. The rules for method invocations treat exceptions analogously to normal return values.

Object Creation, Cast, and `instanceof`. Newly created objects always belong to the currently active context. Therefore, the type of the `new` expression has tag `intern` (T-New and T-NewArray). For simplicity, we assume that a `new` expression directly returns a fresh object without calling a constructor.

The tagged type of a cast expression is the type TT appearing in the cast operator (T-Cast). For simplicity, we do not allow upcasts. That is, TT has to be a subtype of the expression type TS. Upcasts can be simulated by an assignment to a local variable of the desired type.

$$\text{T-New}\ \frac{}{\vdash \texttt{new T()} :: (\texttt{i}, T)} \qquad \text{T-NewArray}\ \frac{\vdash e :: intT}{\vdash \texttt{new TT[e]} :: (\texttt{i}, TT)}$$

$$\text{T-Cast}\ \frac{\vdash e :: TS \qquad TT \preceq TS \qquad (\star)\ TS = (\texttt{a}, \mathcal{S}) \wedge TT = (\gamma, \mathcal{T}) \Rightarrow (\mathcal{S} \preceq_J \texttt{Shareable}^1 \wedge \mathit{ShareItf?}(\mathcal{T}) \vee \mathcal{S} = \mathcal{T})}{\vdash \texttt{(TT) e} :: TT}$$

$$\text{T-Instanceof}\ \frac{\vdash e :: TS \qquad (\star)\ TS = (\texttt{a}, \mathcal{S}) \wedge TT = (\gamma, \mathcal{T}) \Rightarrow (\mathcal{S} \preceq_J \texttt{Shareable}^1 \wedge \mathit{ShareItf?}(\mathcal{T}) \vee \mathcal{S} = \mathcal{T})}{\vdash \texttt{e instanceof TT} :: \mathit{boolean}T}$$

$$\text{T-Invoke}\ \frac{\vdash e1 :: TS \quad \vdash e2 :: TT \quad TS * TT \preceq TP \qquad (\star)\ TS = (\texttt{a}, \mathcal{S}) \Rightarrow \mathit{ShareItf?}(\mathcal{S})}{\vdash \texttt{e1.m(e2)} :: TS * TR} \qquad \text{T-SInvoke}\ \frac{\vdash e :: TS \qquad TS \preceq TP}{\vdash \texttt{T.m(e)} :: TR}$$

$$\text{T-FRead}\ \frac{\vdash e :: TS \qquad (\star)\ TS = (\texttt{i}, \mathcal{S})}{\vdash \texttt{e.f} :: [f]} \qquad \text{T-FWrite}\ \frac{\vdash e1 :: TS \quad \vdash e2 :: TT \quad TT \preceq [f] \qquad (\star)\ TS = (\texttt{i}, S) \quad (\star)\ TT \neq (\texttt{t}, \mathcal{T})}{\vdash \texttt{e1.f=e2} :: TT}$$

$$\text{T-SRead}\ \frac{}{\vdash \texttt{T.f} :: [f]} \qquad \text{T-SWrite}\ \frac{\vdash e :: TT \quad TT \preceq [f] \quad (\star)\ TT \neq (\texttt{t}, \mathcal{T})}{\vdash \texttt{T.f=e} :: TT}$$

$$\text{T-ARead}\ \frac{\vdash e1 :: (\gamma, TE) \quad \vdash e2 :: intT \quad (\star)\ \gamma = \texttt{i} \vee \gamma = \texttt{t}}{\vdash \texttt{e1[e2]} :: (\gamma, TE) * TE}$$

$$\text{T-AWrite}\ \frac{\vdash e1 :: (\gamma, TE) \quad \vdash e2 :: intT \quad \vdash e3 :: TT \quad (\gamma, TE) * TT \preceq TE \qquad (\star)\ \gamma = \texttt{i} \vee \gamma = \texttt{t} \qquad (\star)\ TT \neq (\texttt{t}, T)}{\vdash \texttt{e1[e2]=e3} :: TT}$$

Fig. 3. Tagged type rules.

Method Invocation. For simplicity, we assume that methods have exactly one formal parameter.

The rule for the invocation of instance methods (T-Invoke) has to handle context switches. Consider for example the invocation `ser.doService` in Fig. 2. The declared return type of `doService` is `intern` because the result object is intern to the server context in which the method is executed. When `doService` is invoked from the client context, the returned `Status` object is external to the client context and, therefore, must be tagged `any`. This adaption of the tag is described by the $*$-operator, which combines two tagged types. It is defined as follows:

[1] We write $\mathcal{S} \preceq_J \texttt{Shareable}$ to express that $\mathcal{S}$ is a Java type, which is a subtype of `Shareable`.

$$
\begin{array}{l}
* : \mathit{TaggedType} \times \mathit{TaggedType} \to \mathit{TaggedType} \\
\quad (\gamma, \mathcal{T}) * (\mathtt{i}, \mathcal{S}) = (\mathtt{a}, \mathcal{S}), \qquad \text{if } \gamma \neq \mathtt{i} \\
\quad (\gamma, \mathcal{T}) * \mathit{TS} \quad = \mathit{TS}, \qquad \text{in all other cases}
\end{array}
$$

The $*$-operator tags the parameter or result as `any` when the invocation could lead to a context switch ($\gamma \neq \mathtt{i}$) and an internal reference is passed to or returned by the method.

Static methods are always executed in the currently active context. Therefore, in rule T-SInvoke, the tags do not need to be adapted by the $*$-operator.

Field and Array Access. The type of a field read is the declared type of the field, $[f]$ (T-FRead, T-SRead). T-FWrite and T-SWrite check that the tagged type of the right-hand side of a field update is a subtype of the declared type of the field.

Besides the rules for accessing static fields, there is also a requirement for their declaration: Since class objects do not belong to any context, static fields must not have an `intern` type.

Tagged element types specify the context of array elements relatively to the context of the array object. Therefore, the $*$-operator is used to combine the tagged array type, (γ, *TE*), with the tagged element type, *TE*, to determine the type of an array read access (T-ARead). Similarly, the $*$-combination of the array type and the type of the right-hand side expression of an array update has to be a subtype of the element type (T-AWrite).

2.3 Annotations for the Java Card API

To typecheck applet implementations, tags have to be added to the Java Card API. In particular, these tags determine which objects are entry point objects. Fig. 4 illustrates such API annotations for three methods of class `JCSystem`. According to the API specifications, method `getAID` returns an `AID` object that is a PEP. `getAppletShareableInterfaceObject` takes a `pep AID` and returns a reference to an SIO in any applet context, hence the result type `any Shareable`. Method `makeTransientByteArray` illustrates that exceptions thrown by the JCRE are TEPs. Since the method creates a new array in the context in which it is called, the result type has tag `intern`.

3 Dynamic Semantics

In this section, we formalize and prove type safety based on an operational semantics of a subset of Java Card. Although we have proved type safety for the full language, we omit primitive types, arrays, and exceptions in this formalization for simplicity.

```
public final class JCSystem {
  static pep AID getAID() {...}
  static any Shareable getAppletShareableInterfaceObject
                                (pep AID serverAID, byte parameter) {...}
  static intern byte[ ] makeTransientByteArray(short length, byte event)
            throws tep NegativeArraySizeException, tep SystemException {...}
  // other methods omitted
}
```

Fig. 4. Tags for selected methods of the Java Card API.

3.1 State Model

We build on the formalization of the state model of Java presented in [23]. In the following, we summarize those aspects that are specific to Java Card such as the treatment of contexts.

Contexts, Objects, and Values. A *Context* is either the JCRE context or an applet context, defined by a package name. A key property of the formalization is that each object "knows" the context it belongs to and whether it is a PEP or TEP. Since we do not consider primitive types here, a *Value* is either a reference to an object or *null*. Sorts *PackageId*, *ClassId*, and *ObjId* stand for package names, class names, and object identifiers (addresses), respectively. The function *ctxt* yields the context an object belongs to.

$$
\begin{array}{l}
\mathit{Context} = \mathit{jcreC} \\
\qquad\quad \mid\ \mathit{appletC}(\ \mathit{PackageId}\ ) \\
\\
\mathit{Value} = \mathit{ref}(\ \mathit{Object}\ ) \\
\qquad\quad \mid\ \mathit{null} \\
\\
\mathit{Object} = \mathit{o}(\ \mathit{ClassId}, \mathit{ObjId}, \mathit{Context}\ ) \\
\qquad\quad \mid\ \mathit{pepo}(\ \mathit{ClassId}, \mathit{ObjId}\ ) \\
\qquad\quad \mid\ \mathit{tepo}(\ \mathit{ClassId}, \mathit{ObjId}\ )
\end{array}
\qquad
\begin{array}{l}
\mathit{ctxt} : \mathit{Value} \to \mathit{Context} \cup \{\mathit{undef}\} \\
\mathit{ctxt}(\mathit{ref}(\mathit{o}(T, O, C))) = C \\
\mathit{ctxt}(\mathit{ref}(\mathit{pepo}(T, O))) = \mathit{jcreC} \\
\mathit{ctxt}(\mathit{ref}(\mathit{tepo}(T, O))) = \mathit{jcreC} \\
\mathit{ctxt}(\mathit{null}\ ) \qquad\quad = \mathit{undef}
\end{array}
$$

In addition to these definitions, we use the following functions: *typeof* yields the dynamic Java type of a value. *pepo*? and *tepo*? test whether an object is a PEP or a TEP.

Object Stores. The state of an object is given by the values of its instance variables. We assume a sort *Location* for the instance variables of objects and the static fields of classes. The functions

$$
\begin{array}{l}
\mathit{iv} : \mathit{Value} \times \mathit{FieldId} \ \ \to \mathit{Location} \cup \{\mathit{undef}\} \\
\mathit{sv} : \mathit{ClassId} \times \mathit{FieldId} \to \mathit{Location} \cup \{\mathit{undef}\}
\end{array}
$$

are used to create a location from a value (or class) and a field name (sort *FieldId*).

The state of all objects in the current execution state is formalized by an abstract data type *Store* with the following functions:

$$
\begin{array}{ll}
\_(\_) & : \mathit{Store} \times \mathit{Location} \to \mathit{Value} \\
\_\langle \_ := \_ \rangle & : \mathit{Store} \times \mathit{Location} \times \mathit{Value} \to \mathit{Store} \\
\_\langle \_, \_ \rangle & : \mathit{Store} \times \mathit{ClassId} \times \mathit{Context} \to \mathit{Store} \\
\mathit{new} & : \mathit{Store} \times \mathit{ClassId} \times \mathit{Context} \to \mathit{Object}
\end{array}
$$

$OS(L)$ yields the value of location L in store OS. $OS\langle L := V\rangle$ yields the object store that is obtained from OS by updating location L with value V. $OS\langle T, C\rangle$ yields the object store that is obtained from OS by allocating a new object of type T in context C. $\mathit{new}(OS, T, C)$ yields a reference to a new object of type T in context C. The functions for object creation, $OS\langle T, C\rangle$ and $\mathit{new}(OS, T, C)$, are connected by appropriate axioms. Since these axioms as well as other properties of the above functions are not needed in this paper, we refer the reader to [23] for their axiomatization.

Program States. Program states are formalized as mappings from identifiers to values. A designated variable $\mathcal{C}$ contains the currently active context. We use \$ as identifier for the current object store. We assume that each method has exactly one formal parameter, `p`. *VarId* is the set of identifiers for local variables.

$$
\begin{array}{rl}
\mathit{State} \equiv & (\mathit{VarId} \cup \{\, \texttt{this}, \texttt{p} \,\} \to \mathit{Value} \cup \{\mathit{undef}\}) \quad \cup \\
& (\{\, \$ \,\} \to \mathit{Store}) \quad \cup \quad (\{\, \mathcal{C} \,\} \to \mathit{Context})
\end{array}
$$

For $\sigma \in \mathit{State}$, we write $\sigma(x)$ for the application to a variable or parameter identifier x. In static methods, we set $\sigma(\texttt{this}) = \mathit{null}$. By $\sigma[x := V]$, we denote the state that is obtained from σ by updating variable x with value V. An analogous notation is used for the current object store, \$, and the currently active context, $\mathcal{C}$. *initS* denotes the state that is undefined for all variables, \$, and $\mathcal{C}$.

3.2 Operational Semantics

The operational semantics has two kinds of transitions: $\sigma :: e \to V, \sigma'$ expresses that the evaluation of expression e in state σ yields value V and final state σ'. For statements, $\sigma : s \to \sigma'$ expresses that the execution of statement s in state σ leads to state σ'. Since the rules for statements are the usual Java rules, we omit them here and refer the reader to [21].

The rules for expressions are found in Fig. 5. In the rules, we mark the premises for the dynamic firewall checks with "$(\star)$". We refer to the semantics including the dynamic firewall checks as *strong* semantics, whereas the *weak* semantics does not contain these checks. In the following, we use $\to$ and $\to^\star$ to denote transitions in the weak and strong semantics, respectively.

Following Drossopoulou and Eisenbach [12], we assume that all expressions are annotated with their static types. These annotated versions of the expressions

are produced by the type rules, although we leave that implicit in Fig. 3. In the semantics rules, the static Java type of an expression e is denoted by $[e]$.

The most complex rule handles invocations of instance methods (S-Invoke). $\mathit{impl}(T, m)$ yields the implementation of method m in type T. This implementation can be inherited from a superclass. First, the receiver and actual parameter expressions are evaluated. Next, the implementation of the dynamically-bound method m is executed in a state that maps the formal parameters to the actual parameters and \$ to the store after evaluating the actual parameter. The new currently active context is the context of the receiver object. That is, a context switch may occur. The return value of the method is stored in the special variable, `res`. The strong semantics requires in addition that the receiver object is in the currently active context, an entry point object, or that the static type of the receiver is a shareable interface[2]. These conditions correspond to the firewall checks described in Section 1.1.

3.3 Type Safety

Type safety w.r.t. tagged types means that the tag of the static type of a program element e correctly reflects the context the object denoted by e belongs to. For instance, the object held by a local variable of type `intern T` in an execution state σ has to belong to the currently active context of σ.

Most Specific Tagged Types. An object knows its class, whether it is a PEP, a TEP, or an ordinary object, and its context. Tagged types approximate this information statically. The best approximation for an object X relative to a context C is determined by $\mathit{ttype}(X, C)$. In particular, for a non-entry point object X, $\mathit{ttype}(X, C)$ yields an `intern` type if X is in context C and an `any` type if X belongs to a different context. For example, if X is an instance of class T in context C, then the *most specific tagged type* relative to context C is $(\mathtt{i}, T)$ because X is intern to C. $(\mathtt{a}, T)$ would also be a valid tagged type for X, but is not the most specific one. Function *ttype* is defined as follows:

$$
\begin{aligned}
&\mathit{ttype} : \mathit{Value} \times \mathit{Context} \to \mathit{TaggedType}\\
&\quad \mathit{ttype}(\mathit{ref}(o(T, O, C)), C) = (\mathtt{i}, T)\\
&\quad \mathit{ttype}(\mathit{ref}(o(T, O, C)), D) = (\mathtt{a}, T) \text{ for } C \neq D\\
&\quad \mathit{ttype}(\mathit{ref}(\mathit{pepo}(T, O)), D) = (\mathtt{p}, T)\\
&\quad \mathit{ttype}(\mathit{ref}(\mathit{tepo}(T, O)), D) = (\mathtt{t}, T)\\
&\quad \mathit{ttype}(\mathit{null}, D) = \mathit{nullT}
\end{aligned}
$$

[2] The Java Card documentation [24] formulates these rules for bytecode instructions. Java bytecode provides different instructions for methods declared in classes and interfaces. This distinction is reflected in our uniform invocation rule by referring to the static type of the receiver.

$$\text{S-New}\ \frac{}{\sigma :: \texttt{new T()} \to new(\sigma(\$), T, \sigma(\mathcal{C})), \sigma[\$:= \sigma(\$) < T, \sigma(\mathcal{C}) >]}$$

$$\text{S-Cast}\ \frac{\begin{array}{c}\sigma :: e \to V, \sigma' \qquad ttype(V, \sigma(\mathcal{C})) \preceq TT \\ (\star)\ ctxt(V) = \sigma(\mathcal{C}) \vee pepo?(V) \vee tepo?(V) \vee \\ (typeof(V) \preceq_J \texttt{Shareable} \wedge ShareItf?(TT))\end{array}}{\sigma :: (TT)e \to V, \sigma'}$$

S-Invoke

$$\frac{\begin{array}{c}\sigma :: e1 \to V1, \sigma' \qquad \sigma' :: e2 \to V2, \sigma'' \qquad V1 \neq null, \\ initS[\texttt{this} := V1, \texttt{p} := V2, \$:= \sigma''(\$), \mathcal{C} := ctxt(V1)] : impl(typeof(V1), m) \to \sigma''' \\ (\star)\ ctxt(V1) = \sigma(\mathcal{C}) \vee pepo?(V1) \vee tepo?(V1) \vee ShareItf?([e1])\end{array}}{\sigma :: e1.m(e2) \to \sigma'''(\texttt{res}), \sigma''[\$:= \sigma'''(\$)]}$$

$$\text{S-SInvoke}\ \frac{\sigma :: e \to V, \sigma' \qquad initS[\texttt{p} := V, \$:= \sigma'(\$), \mathcal{C} := \sigma(\mathcal{C})] : impl(T, m) \to \sigma'''}{\sigma :: T.m(e) \to \sigma'''(\texttt{res}), \sigma'[\$:= \sigma'''(\$)]}$$

$$\text{S-FRead}\ \frac{\sigma :: e \to V, \sigma' \qquad V \neq null \qquad (\star)\ ctxt(V) = \sigma(\mathcal{C})}{\sigma :: e.f \to \sigma'(\$)(iv(V, f)), \sigma'}$$

$$\text{S-FWrite}\ \frac{\begin{array}{c}\sigma :: e1 \to V1, \sigma' \qquad \sigma' :: e2 \to V2, \sigma'' \qquad V1 \neq null \\ (\star)\ ctxt(V1) = \sigma(\mathcal{C}) \qquad (\star)\ \neg tepo?(V2)\end{array}}{\sigma :: e1.f = e2 \to V2, \sigma''[\$:= \sigma''(\$) < iv(V1, f) := V2 >]}$$

$$\text{S-SRead}\ \frac{}{\sigma :: T.f \to \sigma(\$)(sv(T, f)), \sigma}$$

$$\text{S-SWrite}\ \frac{\sigma :: e \to V, \sigma' \qquad (\star)\ \neg tepo?(V)}{\sigma :: T.f = e \to V, \sigma'[\$:= \sigma'(\$) < sv(T, f) := V >]}$$

Fig. 5. Selected rules of the operational semantics.

Well-Typed States. Based on the function *ttype*, we can define well-typed states: The most specific tagged type for a variable and a context has to be a subtype of the declared type of the variable (written as $[v]$ for a variable v).

Definition 1 (Well-Typed States). *A state is well-typed if (1) the local variables and formal parameters are correctly typed relative to the currently active context; (2) all instance variables $X.f$ are correctly typed relative to the context of X; (3) all static fields $T.f$ are correctly typed relative to any context:*

$$\begin{array}{l} wt : State \to Bool \\ wt(\sigma) \Leftrightarrow (\forall v \in VarId \cup \{\texttt{this}, \texttt{p}\} : ttype(\sigma(v), \sigma(\mathcal{C})) \preceq [v]) \wedge \\ \qquad (\forall L \in Location : L = iv(X, f) \Rightarrow ttype(\sigma(\$)(L), ctxt(X)) \preceq [f]) \wedge \\ \qquad (\forall L \in Location : L = sv(T, f) \Rightarrow \forall C \in Context : ttype(\sigma(\$)(L), C) \preceq [f]) \end{array}$$

For objects that are neither PEPs nor TEPs, *ttype* uses the context argument C to determine whether the object is internal to C (tag `i`) or not (tag `a`). If a local variable or formal parameter is typed `intern`, the referenced object has to be in the currently active context, $\sigma(\mathcal{C})$. If an instance field f is typed `intern`, the object referenced by $X.f$ has to be in the same context as X. Since static fields can be read and written from any context, they cannot be typed `intern`. Requiring that the object X referenced by static field $T.f$ is correctly typed relative to *any* context enforces that X is a permanent entry point object or $[f]$ has tag `any`.

Java Card is type safe w.r.t. the tagged type system. That is, the type rules—*without* the static firewall checks—ensure that tags correctly reflect dynamic context information. Type safety does not rely on the dynamic firewall checks. That is, it can be proved based on the weak operational semantics.

Theorem 1 (Type Safety). *If the evaluation of a well-typed expression e starts in a well-typed state, σ, and terminates then the final state, σ', is well-typed and has the same currently active context as σ. The resulting value is correctly typed:*

$$\vdash e :: TT \wedge \sigma :: e \rightarrow V, \sigma' \wedge wt(\sigma) \Rightarrow wt(\sigma') \wedge ttype(V, \sigma'(\mathcal{C})) \preceq TT \wedge \sigma(\mathcal{C}) = \sigma'(\mathcal{C})$$

If the execution of a well-typed statement s starts in a well-typed state, σ,and terminates then the final state, σ', is well-typed and has the same currently active context as σ:

$$\vdash s \wedge \sigma : s \rightarrow \sigma' \wedge wt(\sigma) \Rightarrow wt(\sigma') \wedge \sigma(\mathcal{C}) = \sigma'(\mathcal{C})$$

The proof of this theorem uses the following auxiliary lemma. This lemma is used to relate (i) the argument of a method call to the context in which the method is executed and (ii) the result of a call to the context of the caller.

Lemma 1 (Combination Lemma). *Let TS be the tagged type of object X relative to a context C. (i) If TT is the tagged type of value Y relative to C, then the tagged type of Y relative to the context of X is a subtype of $TS * TT$. (ii) If TT is the tagged type of value Y relative to the context of X, then the tagged type of Y relative to C is a subtype of $TS * TT$.*

$$(i)\ \ X \neq null \wedge ttype(X, C) = TS \wedge ttype(Y, C) = TT \Rightarrow ttype(Y, ctxt(X)) \preceq TS * TT$$

$$(ii)\ X \neq null \wedge ttype(X, C) = TS \wedge ttype(Y, ctxt(X)) = TT \Rightarrow ttype(Y, C) \preceq TS * TT$$

Proof: The proof of Lemma 1 runs by case distinction on the tags of TS and TT. It is straightforward and, therefore, omitted.

Proof of Type Safety. The proof of Theorem 1 runs by rule induction on the rules of the weak operational semantics. For brevity, we show only the most interesting case, calls of instance methods. Consider the invocation $e1.m(e2)$. We have to prove that if the evaluation of the call starts in a well-typed state then

(1) the state in which the implementation of m is executed is well-typed. This is necessary to establish the induction hypothesis for the method implementation; (2) the induction hypothesis holds for the final state of the evaluation of $e1.m(e2)$. In the following, TS, TT, TP, and TR are used like in the type rule T-Invoke (Fig. 3).

Part 1: $TT_{\texttt{this}}$ denotes the tagged type of the implicit parameter of m's implementation. $TT_{\texttt{this}} = (\texttt{i}, S)$, where S is the class in which m is implemented. That is, $\mathit{typeof}(V1) \preceq_J S$.

$\mathit{wt}(\sigma)$
$\quad\Rightarrow\quad$ [induction hypothesis for $\sigma :: e1 \to V1, \sigma'$ and $\sigma' :: e2 \to V2, \sigma''$]
$\mathit{wt}(\sigma'') \wedge \mathit{ttype}(V1, \sigma(\mathcal{C})) \preceq TS \wedge \mathit{ttype}(V2, \sigma(\mathcal{C})) \preceq TT$
$\quad\Rightarrow\quad$ [Lemma 1 (i), $V1 \neq \mathit{null}$]
$\mathit{wt}(\sigma'') \wedge \mathit{ttype}(V2, \mathit{ctxt}(V1)) \preceq TS * TT$
$\quad\Rightarrow\quad$ [$TT_{\texttt{this}} = (\texttt{i}, S)$, $\mathit{typeof}(V1) \preceq_J S$; $TS * TT \preceq TP$]
$\mathit{wt}(\sigma'') \wedge \mathit{ttype}(V1, \mathit{ctxt}(V1)) \preceq TT_{\texttt{this}} \wedge \mathit{ttype}(V2, \mathit{ctxt}(V1)) \preceq TP$
$\quad\Rightarrow$
$\mathit{wt}(\mathit{initS}[\texttt{this} := V1, \texttt{p} := V2, \$:= \sigma''(\$), \mathcal{C} := \mathit{ctxt}(V1)])$

Part 2:
$\mathit{wt}(\sigma)$
$\quad\Rightarrow\quad$ [induction hypothesis for $\sigma :: e1 \to V1, \sigma'$ and $\sigma' :: e2 \to V2, \sigma''$]
$\mathit{wt}(\sigma'') \wedge \mathit{ttype}(V1, \sigma'(\mathcal{C})) \preceq TS \wedge \sigma(\mathcal{C}) = \sigma''(\mathcal{C}) \wedge \sigma'(\mathcal{C}) = \sigma''(\mathcal{C})$
$\quad\Rightarrow\quad$ [induction hypothesis for method implementation]
$\mathit{wt}(\sigma'') \wedge \mathit{wt}(\sigma''') \wedge \mathit{ttype}(V1, \sigma'(\mathcal{C})) \preceq TS \wedge \sigma(\mathcal{C}) = \sigma''(\mathcal{C}) \wedge \sigma'(\mathcal{C}) = \sigma''(\mathcal{C})$
$\quad\Rightarrow\quad$ [$\texttt{res}$ is a local variable of m's implementation with type TR]
$\mathit{wt}(\sigma'') \wedge \mathit{wt}(\sigma''') \wedge \mathit{ttype}(V1, \sigma''(\mathcal{C})) \preceq TS \wedge$
$\mathit{ttype}(\sigma'''(\texttt{res}), \mathit{ctxt}(V1)) \preceq TR \wedge \sigma(\mathcal{C}) = \sigma''(\mathcal{C})$
$\quad\Rightarrow\quad$ [Lemma 1 (ii), $V1 \neq \mathit{null}$]
$\mathit{wt}(\sigma''[\$:= \sigma'''(\$)]) \wedge \mathit{ttype}(\sigma'''(\texttt{res}), \sigma''[\$:= \sigma'''(\$)](\mathcal{C})) \preceq TS * TR \wedge$
$\sigma(\mathcal{C}) = \sigma''[\$:= \sigma'''(\$)](\mathcal{C})$

□

Type Progress. Besides type safety, progress is an interesting property of a type system: Progress means that a well-typed program can actually be executed, that is, applying the rules of the operational semantics does not lead to stuck configurations. We do not prove progress formally in this paper. However, one can easily show that the tagged type system guarantees progress if the original Java Card type system does: (1) If a program $\mathbf{P}_{tJC}$ is well-typed in the tagged type system, then the Java Card program $\mathbf{P}_{JC}$ obtained from $\mathbf{P}_{tJC}$ by omitting all tags is well-typed in the Java Card type system, because the tagged type system only imposes additional checks. (2) Besides minor differences for object creation and cast, the strong operational semantics for $\mathbf{P}_{JC}$ and $\mathbf{P}_{tJC}$ are identical. That is, since $\mathbf{P}_{JC}$ can be executed (progress of the Java Card type system), $\mathbf{P}_{tJC}$ can be executed as well. (3) The weak operational semantics is obtained from the strong operational semantics by omitting several requirements. Therefore, $\mathbf{P}_{tJC}$ can also be executed in the weak operational semantics.

4 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 explain these checks and prove that they enforce applet isolation.

4.1 Static Checks

Applet isolation is enforced by additional checks in the tagged type rules (Fig. 3), which are marked by ($\star$). We will explain these premises below.

Object Creation, Cast, and `instanceof`. Object creation is always allowed (T-New, T-NewArray). In Java Card, even finding out type information about objects in other applet contexts is considered a security violation. Therefore, casts are allowed if the object is in the currently active context, if it is an entry point object, or if the object's class implements `Shareable` and the object is cast into a shareable interface. That is, the cast is legal if the object is intern, TEP, or PEP. For tag `any`, we check that the object's class implements `Shareable` and that it is cast into a shareable interface (T-Cast). Moreover, we allow casts from `any T` to `intern T` or `pep T` to refine the tagged type information. The rule for `instanceof` expressions (T-Instanceof) is analogous.

Method Invocation. Instance methods can be invoked on objects (including arrays) in the currently active context, on PEPs and TEPs, and if the static type of the receiver is a shareable interface. Rule T-Invoke requires that if the receiver can be in any context (tag `any`), then its static Java type must be a shareable interface. Static methods can be invoked from any context and need no checks (T-SInvoke).

Field and Array Access. As mentioned in Section 1.1, Java Card forbids field access on objects (including the `length` field of arrays) not in the currently active context. Therefore, the type of the receiver must have tag `intern` (T-FRead, T-FWrite). Since it is not allowed to store TEPs in fields, the right-hand side of a field update must not have tag `tep`. Static fields can be accessed from any context. Therefore, only the check for TEP objects is required (T-SRead, T-SWrite).

Access to an array element is only allowed if the array is either in the currently active context or a global array. Therefore, rules T-ARead and T-AWrite require the tag of the receiver expression to be `intern` or `tep`. Like for field updates, the tagged type of the right-hand side of an array update must not have tag `tep`.

Example. In the example in Fig. 2, the firewall violation would be detected statically. Since `sta` has tag `any`, the invocation `sta.isSuccess()` does not pass the static checks of rule T-Invoke: `Status` is a class and does not implement `Shareable`.

4.2 Applet Isolation Lemma

The static checks described above guarantee applet isolation: 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 `SecurityException`s (except for the dynamic checks for casts).

Theorem 2 (Applet Isolation). *Let e be an expression that can be typed in the tagged type system and that passes the static firewall checks. If e's evaluation in a well-typed state σ terminates normally then the evaluation of the corresponding Java Card expression without tags, $\hat{e}$, in σ terminates normally in the same final state and yields the same value:*

$$\vdash^\star e :: TT \wedge \sigma :: e \to V, \sigma' \wedge wt(\sigma) \Rightarrow \sigma :: \hat{e} \to^\star V, \sigma'$$

where $\vdash^\star e :: TT$ denotes that e is well-typed and passes the static firewall checks. $\to$ and $\to^\star$ denote transitions in the weak and strong semantics, respectively.

Note that omitting tags from expressions makes those casts dispensable that do not change the Java type of an expression. For instance, for a variable v of tagged type $(\mathtt{a}, T)$, omitting the tags from the cast $(\mathtt{i}\ T)v$ yields v. For such cast expressions, the implication of the theorem is trivially true.

Proof of Applet Isolation. The proof of Theorem 2 runs by rule induction on the weak operational semantics. It uses type safety of the tagged type system and the static firewall checks. Again, we show the proof for the most interesting case: the invocation of instance methods.

Since we have the transition $\sigma :: e \to V, \sigma'$, we know that all premises of rule S-Invoke in the weak semantics hold. Applying the induction hypothesis to these premises yields the corresponding premises in the strong semantics. It remains to show that the additional premise in the strong semantics holds: $\mathit{ctxt}(V1) = \sigma(\mathcal{C}) \vee \mathit{pepo?}(V1) \vee \mathit{tepo?}(V1) \vee \mathit{ShareItf?}([e1])$.

From the premise $\sigma :: e1 \to V1, \sigma'$ and type safety (Theorem 1), we get $\mathit{ttype}(V1, \sigma(\mathcal{C})) \preceq TS$. We may assume that TS is a reference type $(\gamma, [e1])$. We continue by case distinction on the tag γ:

1. *Case* `i`: Subtyping on tagged types gives that $\mathit{ttype}(V1, \sigma(\mathcal{C}))$ has tag `i`. The definition of *ttype* yields $V1 = \mathit{ref}(o(S, O, \sigma(\mathcal{C})))$ for some S, O. Therefore, $\mathit{ctxt}(V1) = \sigma(\mathcal{C})$.
2. *Case* `p`: Analogously to Case `i`, we get $V1 = \mathit{ref}(\mathit{pepo}(S, O))$ for some S, O. Therefore, $\mathit{pepo?}(V1)$ holds.
3. *Case* `t`: This case is analogous to Case `p`.
4. *Case* `a`: The static firewall check of rule (T-Invoke) gives directly the result $\mathit{ShareItf?}([e1])$

□

Theorem 2 shows that well-typedness and the static firewall checks guarantee that execution of an expression does not violate the firewall at runtime. Therefore, the checks can be used to enforce applet isolation statically.

5 Discussion

In this section, we discuss the expressiveness of our type system, the overhead it imposes on programmers, and its possible applications.

5.1 Expressiveness

The proposed type system does not significantly limit the expressiveness of Java Card: Almost all ordinary Java Card programs can be handled, possibly by introducing additional downcasts. This flexibility is due to the fact that `any` types are supertypes of the corresponding `intern` and `pep` types. Therefore, variables that may hold references to objects in various contexts can be typed `any`, and casts can be used when such variables are read. In such situations, the expressiveness of the type system comes at the price of runtime checks. However, extra downcasts are only needed for two purposes: (1) when an internal object is stored in a static field and then read again (recall that static fields must not have `intern` types); (2) when an internal object is passed to a different context and then retrieved again (e.g., from a container in a different context).

The only pattern that cannot be typed in our type system is when a variable may hold a reference to a TEP or a non-TEP object. In such cases, the variable can neither be typed `tep` nor `any`. However, this situation is extremely uncommon since TEPs must not be stored in fields or arrays.

As presented in this paper, the tagged type system does not support contravariant subtyping, which prevents certain implementations that are admissible in Java Card. Assume that a class C inherits a method `void m(intern T p)` from its superclass, D, and implements a shareable interface, I, that declares `void m(any T p)`. Without tags, C would be a legal Java Card implementation, but C is forbidden by the tagged type system since it does not implement I's `void m(any T p)`. However, this is not a serious restriction: If D's implementation of `m` can handle parameter objects in other contexts, the parameter `p` should be declared `any`. Otherwise, C has to override the method anyway, and contravariant subtyping would allow C to widen the signature of `m` to `void m(any T p)`. Extending the tagged type system to contravariant subtyping w.r.t. tags is straightforward but omitted in this paper for simplicity.

5.2 Defaulting

The static safety of our type system comes at the price of some extra work for programmers, who have to add tags to their programs. However, for the majority of types, the tags can be determined easily. Except for static fields and program elements involved in the interaction with the JCRE or other applets, all tags are usually `intern`. Therefore, we can use `intern` as default tag for most types. More precisely, we default each untagged occurrence of a Java type T to the tagged type (δ, T), where δ is:

- `pep` if $T =$ `AID`;

– `tep` if $T =$ `APDU` or T is an exception class in the Java Card API;
– `any` if T is an interface extending `Shareable` or if the occurrence of T is in the declaration of a static field;
– `intern` otherwise.

These defaults reduce the overhead significantly. For instance, all tags of the Java Card API methods in Fig. 4 could be omitted. Although method `process` in Fig. 2 communicates with the JCRE and another applet, only the tags for the cast `(any Service)s` and the declaration `any Status sta` have to be specified manually. The fact that defaulting does not work in the latter case already indicates the error in the program. Usually, a well-formed applet does not hold references of type `any T` if `T` is *not* an interface that extends `Shareable`.

The cases in which defaulting is not sufficient are (1) the extra downcasts needed for reading static fields and (2) arrays of type `byte[]`, which are heavily used as internal objects and as global arrays for the `APDU` buffer.

An alternative to default tags would be type inference. Since inference has been applied to the complex context information in ownership type systems [2, 9], we assume that inference would be applicable here as well. We used defaulting since it works for modular programs.

5.3 Applications

In Section 4, we have shown that the type system can be used to check Java Card's applet isolation statically. The static context information can also be used to enforce stricter policies. For instance, an applet can easily be prevented from interacting with other applets by checking that no program element has tag `any` in the applet's code. Note that this policy cannot be enforced by just forbidding calls to `JCSystem.getAppletShareableInterfaceObject` since applets can also exchange references through static fields.

Our main motivation was to simplify the verification of source programs by checking applet isolation syntactically before verifying the program. Therefore, the type system is applied to source programs. However, the type system can easily be adapted to bytecode. An adapted bytecode verifier [16] could check applet isolation at load time. In that case, a modified virtual machine would only have to check applet isolation for downcasts from `any` types to `intern` or `pep` types. This would lead to a significantly faster program execution without weakening the security of the Java Card platform.

6 Related Work

The presented type system benefited from the work on ownership type systems [1, 2, 9, 11]. Like in these type systems, objects are grouped into contexts, and types approximate context information statically. However, ownership type systems provide hierarchic context structures, whereas the contexts in Java Card are flat. Like readonly references in the Universe type system [19, 17], the work

presented here permits references between different contexts, but restricts the operations that can be performed on such references. Both Universes and the type system presented here use downcasts to specialize context information. Due to these commonalities, we expect that both type systems can be easily integrated into one type system that facilitates the verification of Java Card programs.

Most ownership type systems use owner parameters to keep track of the context an object belongs to. A similar mechanism could be useful in our work to provide more fine-grained context information than `any` tags do, and to make downcasts with dynamic context checks dispensable. However, references to SIOs are obtained through calls to `JCSystem.getAppletShareableInterfaceObject`. The most specific tagged result type for this method is `any Shareable` since it is not known statically from which context a reference is requested.

Similarly to Confined Types [8], Java Card provides one context per package. However, with Confined Types only the code in package P can modify objects in the context for P, whereas Java Card only uses the package structure to determine which applets share one context. The code that modifies the objects of an applet can reside in arbitrary packages.

Several static analyses for information flow between Java Card applets have been published. Bieber et al. [6, 7] present an approach that allows smart card issuers to verify statically by model checking that an applet satisfies a pre-defined security policy. This analysis is complementary to applet isolation. It is able to detect illicit information flow between several applets, whereas the applet firewall controls the interaction between two applets.

Caromel et al. [10] propose a dataflow analysis to infer context information statically. This information is then used to point programmers to potential firewall violations. Éluard and Jensen [13] combine a dataflow analysis with quantified conditional constraints to check more fine-grained sharing policies such as sharing between designated applets rather than all applets on a card. In contrast to dataflow analyses, the tagged type system allows programmers to record design decisions about applet sharing in the code, which serves as additional documentation and enables modular checking. Checking applet isolation based on dataflow analyses is too expensive to be performed on-the-fly by a virtual machine; our type system could be easily checked by a bytecode verifier.

The Java Card platform and, in particular, the applet firewall, have been formalized in different frameworks [5, 14]. These formalizations have been used to formally verify applet isolation and confidentiality properties [3, 4, 15]. With our type system, applet isolation can be mostly checked syntactically.

7 Conclusions

We presented a refined type system for Java Card that allows one to check applet isolation mostly statically. In theory, our type system can replace almost all dynamic firewall checks. However, unless all applets on a card are checked by our type system and a refined bytecode verifier, the dynamic checks have to stay

in place to prevent applets from untrusted sources from violating the firewall. Still, the type system is useful to detect possibly fatal errors at compile time.

Our approach to checking applet isolation is complementary to formal verification of applet properties. Using this type system reduces the verification effort significantly since applet isolation does not have to be proved for each method call, field access, instanceof, etc. as it is the case in plain Java Card. On the other hand, verification techniques can be applied to prove that downcasts do not lead to `SecurityException`s. As future work, we plan to implement the type system in our verification tool Jive.

Acknowledgments. Piotr Nienaltowski provided us with helpful comments on a draft of this paper. We also thank the anonymous reviewers of this paper as well as the reviewers of an earlier version, which was presented at the ECOOP 2001 workshop on Formal Techniques for Java Programs [18], for their valuable comments.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2004.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2002.
3. J. Andronick, B. Chetali, and O. Ly. Using Coq to verify Java Card applet isolation properties. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logic*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 2003.
4. G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: A toolset for reasoning about JavaCard. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.
5. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the JavaCard platform. In D. Sands, editor, *Programming Languages and Systems (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
6. P. Bieber, J. Cazin, A. El-Marouani, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. The PACAP prototype: a tool for detecting Java Card illegal flows. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2041 of *Lecture Notes in Computer Science*, pages 25–37. Springer-Verlag, 2001.
7. P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. *Journal of Computer Security*, 10(4):369–398, 2002.
8. B. Bokowski and J. Vitek. Confined types. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, 1999.

9. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Doctor of philosophy, Electrical Engineering and Computer Science, MIT, February 2004.
10. D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java Card object sharing. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2001.
11. 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.
12. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer-Verlag, 1999.
13. M. Éluard and T. Jensen. Secure Object Flow Analysis for Java Card. In *Proceedings of 5th Smart Card Research and Advanced Application Conference (Cardis'02)*, pages 97–110. USENIX, 2002.
14. M. Éluard, T. Jensen, and E. Denney. An operational semantics of the Java Card firewall. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 95–110. Springer-Verlag, 2001.
15. M. Huisman, D. Gurov, Chr. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: A case study. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering (FASE)*, number 2984 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 2004.
16. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.
17. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
18. P. Müller and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Formal Techniques for Java Programs*, 2001.
19. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
20. T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, New York, 1998.
21. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
22. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1998.
23. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.
24. Sun Microsystems, Inc. *The Runtime Environment Specification for the Java Card Platform, Version 2.2.1*, October 2003.