Owners as Trusters: Trusted Ownership Declassification With Neighbourhood

Pradeep Kumar D.S and Janardan Misra

Accenture Technology Labs
Bangalore, India, 560060
{p.duraisamy,janardan.misra}@accenture.com

Abstract. Ownership types impose information hiding by statically restricting an owned object from becoming accessible outside the ownership boundary. However, such restriction on the object's accessibility limits the expressiveness of the software to capture the properties of applications requiring dynamic sharing of information with constraints aiming to limit potential encapsulation breaches. Hence, in order to design applications involving complex sharing of objects at run-time, we require two major constructs: dynamic sharing of objects and trusted sharing environment (i.e., an ability to control the accessibility of the shared objects). In this paper, we propose an ownership types system for trusted object-level access control called *'owners-as-trusters'*, which is implemented as a language construct. In this system, information can be made visible dynamically through explicit declassification. However, accessibility of the declassified information is controlled through trusted neighbourhood. Thus with our new ownership types system, we decouple accessibility and visibility from the ownership mechanisms.

Keywords: ownership types; neighbourhood; trusted ownership; declassification

1 Introduction

Real applications often require continuous change of behaviour in order to adapt dynamically according to the requirements that cannot be anticipated at the application design time. Designing such a system may involve dynamic object sharing between systems. Such changes must be well supported by the programming languages by ensuring that the object sharing is trusted and can be controlled dynamically.

One of the major threats to application level security is Aliasing [15], which may permit unauthorized access to the object's internal representations. Protecting objects against aliasing at package level in OO programming languages (e.g., Java) may get compromised by exposing the object through assignments, method calls, or method returns. Ownership types [9] encapsulation has been proposed as an effective technique for protection against aliases at the object reference level. Ownership types maps objects onto an *ownership tree*, which

refers to the reflexive transitive closure of the ownership relation among objects and is rooted at the *world* level. Thereby the ownership types enforces *owners-as-dominators* [9,10] property by specifying information hiding for an object by making it accessible (or mutable) only within the encapsulation provided by its owner and an access from outside crossing the encapsulation boundary is restricted. This is known as fixed aliasing or non-interference property. However, the *owners-as-dominators* property may constrain implementing some of the useful software patterns like call-backs and iterators that require references to the objects' internal representation and also may hinder many functionalities that require the release of sensitive information to become accessible to all.

The other flavor of the static ownership types is *owners-as-modifiers* [13]. It enforces the *owners-as-dominators* property by restricting the accessibility (or mutability) from outside the ownership boundary, however, it relaxes the non-interference property by permitting the read-only access from outside the ownership boundary, which makes the system weak in information hiding.

In contrast to the above static ownership types, the dynamic ownership types as given by *owners-as-downgraders* [22], specifies that the objects can be exposed outside the ownership boundary through explicit declassification [4]. With declassification, the strict ownership confinement property is relaxed by explicit release of confidential data at most one level up the ownership tree. Hence, if an object has to be exposed through more than one level, it must be explicitly declassified at each level up in the ownership branch. Thus a sequence of authorizations are made effective in the process of declassification [22]. Thus after declassification the declassified object is reflected with a new dynamic owner and thereby the declassified object's accessibility also changes dynamically depending on its dynamic owner. However, the declassification mechanism and its associated ownership-based accessibility are inherently unsafe and create the possibility that the declassified object may become dynamically accessible to other outside objects also, which may not have been originally intended by the system designer.

In this paper, we propose a new ownership types called *owners-as-trusters*, which allows trusted sharing of objects after declassification. Our new ownership types system gives assurance to the programmers that information sharing can be permitted safely by explicit declassify operation, however, information access can be controlled via explicit *neighbourhood* relationship. Thus in our model with the introduction of *neighbourhood*, we empower the owners to control the accessibility of its declassified objects to a restricted subset of objects at the peer-level. It is assumed that these restricted subset of objects specified by the designer comprises the set of *"trusted"* objects that should access the declassified object. Further discussion on the declassification mechanism and accessibility problem is presented in the next section.

The paper is organized as follows: Section 2 describes background and motivation for the work with an example which will be used throughout this paper for illustration of introduced syntax and semantics. Section 3 provides an informal overview of our proposed neighbourhood system. Section 4 presents formal defi-

nition and semantics of the neighbourhood model. Section 5 evaluate our system with respect to other related works and finally Section 6 concludes the paper.

2 Background and Problem Statement

In OO programming, aliasing is considered a double-edged knife - offering advantage in creation of advanced data structures and disadvantage in object's reference leakage [**?**], which may permit unauthorized access to the data structure. John Hogg et al. recognized object aliasing as a major problem for security in [16]. Earlier work on encapsulation, namely, the Islands [15] and the Balloon types [3] present a work on full alias encapsulation, where both incoming and outgoing references to an object's internal representation are restricted. However, such a restriction also reduces the flexibility of working with advanced data structures. *ownership types* [9,10] removes this restriction by permitting outgoing references while still restricting incoming references. In property 1, we define the static owners-as-dominators property.

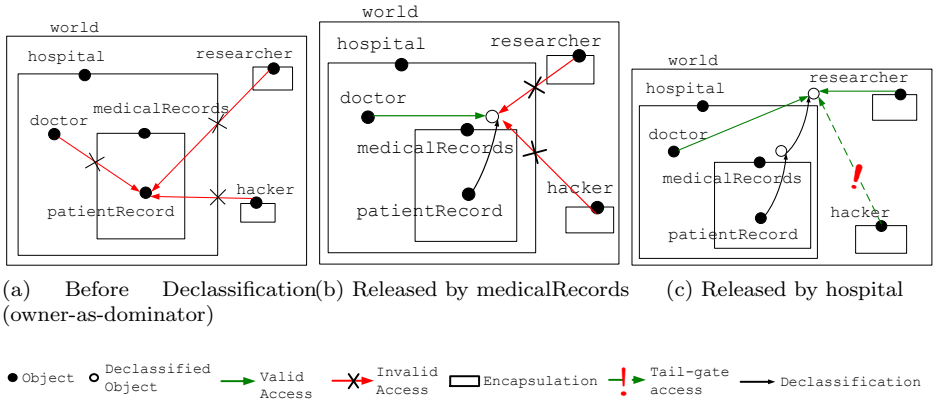
Property 1 (Owners-As-Dominators Property: Static Ownership). An object A can be directly accessed by an object B, iff $B \prec: owner(A)$.

The function *owner(A)* defines the static owner of an object A, which is fixed at A's creation time and remains same throughout its lifetime. The owner of an object gives a logical boundary to its internal objects and protects it from direct accesses from the outside objects. This property is known as owners-as-dominators, which states that every access to an object must go through its owner.

The symbol $\prec:$, known as *within* (or *dominated by*) relationship, denotes the ownership tree as a reflexive and transitive closure of the ownership relation [27]. Using static ownership, an object will have definite position in the ownership tree that specifies the capability of the object by determining its accessibility and visibility properties.

In order to overcome fixed aliasing policy of the owners-as-dominators, the owners-as-downgraders [22] allowed the developers to expose the encapsulated objects at run-time through an explicit declassification operation. This is achieved by step-wise upward propagation of object's visibility in the ownership hierarchy, which results into corresponding authorizations to access the exposed object by other peer-level objects. Through explicit declassification operation an object can change its owner dynamically (called *dynamic owner*). The dynamic owner is the context that represents the object after declassification. Unlike other ownership mechanisms, where accessibility of an object is statically determined by its static owner, with owners-as-downgraders an object's accessibility is determined based on its dynamic owner. Thus the accessibility of an object can be changed at run-time after declassification depending on its dynamic owner without violating the owners-as-dominators property.

However, the inherent problem with the above system lays with the associated ownership-based accessibility, where the accessibility on the declassified object



(a) Before Declassification (b) Released by medicalRecords (c) Released by hospital (owner-as-dominator)

Fig. 1: Medical Study Example.

[22] cannot be further refined by limiting the access only to a specific subset of objects.

Let us illustrate the declassification mechanism and accessibility problem with an example of medical study adapted from [4]. Figure 1 depicts the step-by-step declassification operation and its side-effects. In this scenario, we have a *hospital* object composed of a *doctor* object and a *medicalRecords* object. The *medical record* object owns the *patientRecord* object, which contains sensitive information including disease and prescription histories, which should be kept confidential and whose accessibility should be limited to trusted entities in a controlled fashion.

Later, in order to improve the hospital's medical practicing standard, the hospital decides to share its sensitive patient records to some of the trusted external researchers for a detailed study of diseases and practice. Now, with the classical owners-as-dominators policy, shown in Figure 1(a), the *patientRecord* object cannot be directly accessed by other objects outside its owner.

To achieve this, we can use dynamic declassification of the patients' records, as suggested in [22], by changing the dynamic owner of the *patientRecord* object, and thereby permitting access to it dynamically. Dynamically declassifying an object makes the object move up in the ownership tree and thus the accessibility (a.k.a. mutability) and visibility (a.k.a. immutability or 'read-only access') of that object also changes dynamically.

In Figure 1(b), the *patientRecord* object declassifies its dynamic owner from *medicalRecords* to the *hospital*. This will make the *doctor* object its peer and therefore as per the classical ownership accessibility policy the *doctor* object can now access the *patientRecord* object. However, to make the *researcher* object to access the *patientRecord* object, we have to declassify it to another level up. Hence, by declassifying the *patientRecord* object one level up will change its dynamic owner from the *hospital* to the *World*. Now the *researcher* object becomes a peer object for the *patientRecord* object and hence *patientRecord* object will become accessible to the *researcher* object.

However it turns out that such declassification could lead to untrusted access to the declassified information at run-time. As shown in Figure 1(c), where the *hacker* object is located at the same level of the *researcher* object by default gain the access to the *patientRecord* object, which may not be intended by the system designer. This is an example of the *tail-gate access* problem, described next.

Definition 1 (Tail-gate Access Problem). *The possibility of the declassified object becoming accessible to other (unintended) objects dynamically owing to the fixed ownership access policy may create adverse side-effects that may not be intended by the system designer.*

We aim *to find solution to the tail-gate access problem* by *restricting accesses to the declassified object among peer objects and their contexts*. This is achieved by introducing neighbourhood relationship at the peer level.

We define a new ownership types system called *owners-as-trusters*. When comparing to the classic ownership types systems where objects' visibility and accessibility are coupled, our type system decouples the accessibility and visibility from the ownership mechanisms in two key ways. First, by dynamically changing the visibility of the object using explicit declassification. Secondly, by controlling the accessibility of the declassified object by providing trusted access environment specifying the neighbourhood contexts (ref. Section 3) to whom the dynamic sharing of the object's ownership is permitted. Note that, the owners-as-trusters model ensures that the declassified information will still be preserved within the dynamic owners boundary, however additionally it ensures that only those trusted neighbourhood objects can access the declassified information.

3 Object Sharing with Trusted Neighbourhood

In this section, we will introduce the concept of trusted ownership and its properties, illustrate the approach with an example, and reason that using the notion of trusted neighbourhood we can achieve trusted declassification.

3.1 Trusted Ownership

In this section we dicuss the trusted ownership types mechanism in detail. For this purpose, we need to introduce notions of *dynamic owner*, *trusted owner*, and *neighbourhood relationship* to specify the trust among objects explicitly. In our system declassification can be initiated via explicit **declassify** operation and the trusted neighbourhood relations can be established via **neighbours** operation. The neighbourhood relation is directional, i.e. if D **neighbours** C, it means D has given access permission to C on its declassified object but the reverse need not hold. Thus with the neighbourhood relationship the developer can explicitly specify a subset of objects that were granted permission to access the declassified object and can restrict others from accessing it even though it is visible to those objects'.

```
class Hospital<O> {
  MedicalRecords<this> mR;
  Doctor<this> dR;
  [?][this]PatientRecord pR;
  [this][O]PatientRecord getRecord(ID)
  {  //declassify to one-level above
    pR = mR.getPRecord(ID); //OK
    //dR.review(pR);  // error
    mR neighbours dR;
    dR.review(pR);     // OK
    return declassify pR;
  }}
class MedicalRecords<h> {
  PatientRecord<this> pR1;
  [this][h]PatientRecord getPRecord(ID)
  {
    pR1 = new PatientRecord<this>();
    return declassify pR1;
  }}
```

```
//client code
Hospital<world> hospital =
       new Hospital<world>();
[?][world]PatientRecord<world> pR2 =
       new PatientRecord<world>();

pR2 = hospital.getRecord(PID);

Researcher<world> rX =
       new Researcher<world>();
// rX.diseaseResearch(pR); //error

hospital neighbours rX;
rX.diseaseResearch(pR);

Hacker hX = new Hacker<world>();
hX.trackRecord(pR);
```

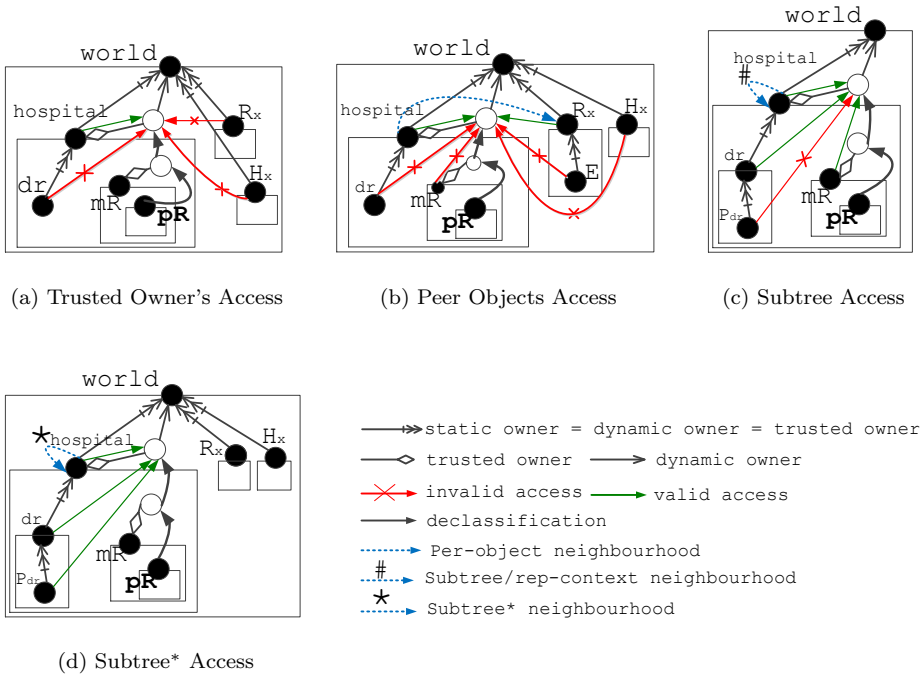
Fig. 2: Scenario Code.

To begin with, *static owner* represents the context within which an object is located at the time of its creation and hence fixes the position of the object in the ownership tree and helps in determining the capability of the object. The *dynamic owner* represents the context within which the object is located after declassification and hence determines the object's visibility for other objects in the ownership tree. The *trusted owner* represents the context that declassified it recently and hence determines the object's accessibility for other objects in the ownership tree through neighbourhood. Thus, in contrast to the previous declassification model of [22], in our proposed approach, after declassification the accessibility is determined based on the object's trusted owner rather than the object's position in the ownership tree (i.e. based on dynamic owner). At the time of object creation, the trusted owner, the dynamic owner and the static owner will be equivalent.

Next, let us define some of the basic properties specifying access between peer objects in the ownership tree.

Property 2 (Owners-as-Trusters). For two objects *hospital* and pR (as shown in Fig. 3(a)), if $owner(pR) \prec: hospital \prec: dOwner(pR)$, then *hospital* must be the **trusted owner** of pR (a.k.a. $tOwner(pR)$).

Above property defines the trusted owner of a declassified object (shown in Fig. 3(a)). For example, the trusted owner of the object pR is the hospital object, which recently declassified the representation object of pR one level up in the ownership tree. The function $tOwner(pR)$ returns the trusted owner of the object pR (i.e. hospital object in this case).



(a) Trusted Owner's Access (b) Peer Objects Access (c) Subtree Access

(d) Subtree* Access

Fig. 3: Trusted Ownership Declassification

3.2 A Medical Study Example

In this section, we will illustrate the declassification (with trusted neighbourhood) using the medical study example depicted in Fig. 1. The sample code given in Fig. 2 highlights how the neighbourhood is used to limit the access among the peer objects during declassification. In Fig. 3, we illustrate the scenario code of the medical study example (in Fig. 2) by graphically depicting how the accessibility and visibility are decoupled.

In our language model, the traditional type (i.e., language defined or user defined 'class' type) of an object is prefixed with trusted owner and the dynamic owner by the programmer. The trusted owner is represented either by [**?**] or by [**this**]. If the programmer does not state the trusted owner, it is assumed implicitly as **this**. Using [**?**], the programmers can abstract the name of the concrete context that is owned by **this**. The [**?**] context helps in assignment of the declassified object to another object present in the immediate higher level context (one level up in the ownership tree), where the programmer cannot name the trusted owner context explicitly at the object creation time. Thus by abstracting as [**?**], the name can be filled by any other object's name present within the owner's representation. Note that the trusted and dynamic owners for a newly created object will always be the same as its static owner. In the scenario code

$$
\begin{array}{lll}
CD & ::= class\ C\langle O_{name}\rangle \big(extends\ C\langle O\rangle\big)_{opt}\ body; & \\
O & ::= this \mid world \mid O_{name} & \\
body & ::= \{\ \bar{\mathrm{t}}\ \bar{\mathrm{f}}\ ; \overline{\mathrm{M}}\ \} & O_{name} \in \text{ owner names} \\
t & ::= \big([t_{owner}][d_{owner}]\big)_{opt}\ C\langle O\rangle & C \in \text{ class names} \\
t_{owner} & ::= this \mid ? & m \in \text{ method names} \\
d_{owner} & ::= O & x, y \in \text{ variable names} \\
M & ::= t\, m\, (\overline{t\, x})\{e\} & e \in \text{ expressions} \\
e & ::= x; \mid e.f; \mid e.m(\bar{\mathrm{e}}); \mid e.f = e; & \overline{\mathrm{M}} \in \text{ sequence of methods} \\
 & \quad \mid new\ C\langle O\rangle(\bar{\mathrm{e}}); \mid return\ e; \mid declassify\ e; & \bar{\mathrm{t}} \in \text{ sequence of types} \\
 & \quad \mid e\ neighbours\ e\ \big(\langle \# \mid * \rangle\big)_{opt}; & \bar{\mathrm{f}} \in \text{ sequence of fields}
\end{array}
$$

Table 1: Syntax of Trusted Java

example, for the pR2 object (underlined in Fig. 2)), the trusted owner has been abstracted while the dynamic owner and the static owners are same, i.e. world, at the time of its creation. However while assigning the pR2 object with the pR object, the abstracted trusted owner of pR2 will point to the hospital object and the dynamic owner will point to the world object. In the next subsection, we shall define the accessibility between the declassified object's and their dynamic peer objects.

Dynamic Peer objects accessibility The following accessibility properties were based on the Fig. 3.

1. By default the object pR can be accessed by its trusted owner (also a peer object by position). As shown in Fig. 3(a), the pR can be accessed by its trusted owner hospital and all other objects can not access pR.
2. The object pR can be directly accessed by its peer object Rx, iff Rx $\prec$: $dOwner(pR)$ and $tOwner(pR)$ **neighbours** Rx. As shown in the Fig. 3(b).
3. The object pR can be directly accessed by the representation context of its peers, iff the trusted owner of pR has rep-context level access permission (a.k.a. *subtree neighbourhood*) with the corresponding peer objects, as shown in Fig. 3(c).
4. The object pR can be directly accessed by all the branch objects, iff the trusted owner of pR has recursive level access permission (a.k.a. *subtree* neighbourhood*) with the corresponding peer objects, as shown in Fig. 3(d).

4 A Type System for Neighbourhood

The syntax of the proposed language, which would be referred as *Trusted Java* (**TJ**) is given in Table 1. The meta-variables A, B, C, D, and E range over the class

names; $\overline{f}$ and $\overline{g}$ range over a sequence of field names; x, y, $a_{ob}, b_{ob}, c_{ob}, d_{ob}$, *and* e_{ob} ranges over variables; m (M) ranges over method names (method declarations); $\overline{e}$ range over terms; CD ranges over class declarations; O and P range over ownership contexts.

The declaration *class* C$\langle O\rangle$ extends D$\langle P\rangle$ $\{\overline{\mathrm{t}}\ \overline{\mathrm{f}}\,;\ \overline{\mathrm{M}}\}$ introduces a class named C with O as ownership context and with superclass D having ownership context P. The type t is prefixed with the trusted and the dynamic owner, the $[t_{owner}]$ and $[d_{owner}]$ respectively (shown in table 1). The term e contains statements for declassification and for establishing neighbourhood among objects.

The neighbourhood relationship can be classified into three modes. Firstly, ***the per-object mode***, written as x **neighbours** y, where the object x declares object y as its neighbour. This states that y can access objects for which x is the trusted owner (i.e. declassified objects of x). However y's context (a.k.a. y's subtree nodes) cannot access x's declassified objects. Secondly, ***the per-object subtree mode***, written as x **neighbours** $y\langle\#\rangle$, where the object x declares object y and its context as its neighbour. This states that y and its context objects can access the objects for which x is the trusted owner. However objects within y's context's subtree nodes cannot access x's declassified objects. Thirdly, ***the per-object subtree* mode***, written as **x neighbours y**$\langle*\rangle$, where object x declares object y and its subtree as its neighbour. This states that y and recursively its subtree node objects can access the objects for which x is the trusted owner. We defer the detailed description on neighbourhood relationship and its functionality until the subsection:(4.3).

4.1 Auxiliary Definitions

Table 2 presents auxiliary definitions for the context look up and the neighbourhood. The function *peer*(.), given in the rule (C-Peer), returns the set of peer objects for the argument object . in the ownership tree. The function $rep(C\langle O\rangle)$ in rule (C-Rep) returns a set of representation of the object c_{ob}. The function $subtree(c_{ob})$ is a reflexive closure, given in the rule (C-Subtree), returns the context of the object c_{ob} including *this* object. The function $subtree^*(c_{ob})$, given in the rule (C-Subtree*) is a reflexive and transitive closure, and returns the recursive context of the object c_{ob}.

There are three basic rules to define the neighbourhood properties. The rule (N-PerObj), define the neighbourhood relationship between peer objects. In other words, this rule specifies that the developer has explicitly permitted the neighbourhood d_{ob} object to access the declassified objects for which c_{ob} object is the trusted owner. The rule (N-Subtree), defines the neighbourhood relationship between peer objects and their representation. The rule (N-Subtree*), define the neighbourhood relationship recursively between peer objects and their representation subtree. In order to determine static valid access among the peer objects, the rule is given by (T-StaticPeerAccess), where the function $canAccess(b_{ob}, d_{ob})$ states that the objects b_{ob} and d_{ob} are peer objects, and the object b_{ob} can access d_{ob} object since static, dynamic and trusted owners of the d_{ob} are equal. This means that d_{ob} has not undergone any declassification and

(C-Peer)

$$\frac{fields(\,C\langle O\rangle\,) \;=\; \{A_1\; g_1;\; \overline{B_2}\;\overline{g_2}\};}{peer(g_1) \;=\; \{\,\overline{g_2}\,\}}$$

(C-Rep)

$$\frac{fields(\,C\langle O\rangle\,) = \{\overline{B}\;\overline{x}\,\};}{rep(C\langle O\rangle) \;=\; \{\overline{x}\}}$$

(C-Subtree)

$$\frac{\begin{array}{l}c_{ob} = new\; C\langle O\rangle();\\ rep(c_{ob});\end{array}}{subtree(c_{ob}) \;=\; \{c_{ob},\; rep(c_{ob})\}}$$

(C-Subtree*)

$$\frac{\begin{array}{l}c_{ob} = new\; C\langle O\rangle();\\ rep(c_{ob});\end{array}}{subtree^*(c_{ob}) \;=\; \{c_{ob},\; subtree^*(rep(c_{ob})\}}$$

(N-PerObj)

$$\frac{\begin{array}{l}peer(c_{ob}) \;=\; \{\,d_{ob},\; e_{ob},\; \overline{x}\,\};\\ c_{ob}\; neighbours\; d_{ob};\end{array}}{neighbourhood(c_{ob}) \;=\; \{d_{ob}\}}$$

(N-Subtree)

$$\frac{\begin{array}{l}peer(c_{ob}) \;=\; \{\,d_{ob},\; e_{ob}\,\overline{x}\};\\ c_{ob}\; neighbours\; d_{ob}\langle\#\rangle;\; subtree(d_{ob});\end{array}}{neighbourhood(c_{ob}) = subtree(d_{ob})}$$

(N-Subtree*)

$$\frac{\begin{array}{l}peer(c_{ob}) \;=\; \{\,d_{ob},\; e_{ob}\,\overline{x}\,\};\\ c_{ob}\; neighbours\; d_{ob}\langle *\rangle;\; subtree^*(d_{ob});\end{array}}{neighbourhood(c_{ob}) = subtree^*(d_{ob})}$$

(T-StaticPeerAccess)

$$\frac{\begin{array}{l}peer(d_{ob}) = \{\overline{e}\};\\ \big[sOwner(d_{ob}) = dOwner(d_{ob}) = tOwner(d_{ob})\big]\; OR\\ \qquad\big[tOwner(d_{ob}) = \{\overline{e}\}\big]\end{array}}{canAccess(\overline{e}, d_{ob}) \;\; is \;\; TRUE}$$

(T-StaticIn2OutAccess)

$$\frac{\begin{array}{l}rep(world) = \{c_{ob},\; \overline{e_1}\};\\ rep(c_{ob}) = \{d_{ob}, b_{ob}, \overline{e_2}\};\; rep(b_{ob}) = \{\overline{e_3}\};\\ \big[sOwner(d_{ob}) = dOwner(d_{ob}) = tOwner(d_{ob})\big]\; OR\\ \qquad\big[neighbourhood(tOwner(d_{ob})) = \{\overline{e_3}\}\big]\end{array}}{canAccess(\overline{e_3}, d_{ob}) \;\; is \;\; TRUE}$$

Table 2: Auxiliary Definitions (For Context Lookup and Neighbourhood)

hence the capability is determined purely based on the fixed ownership policy, which allow the b_{ob} to access d_{ob}. The rule (T-StaticIn2OutAccess) defines the valid access from inside the ownership boundary to the objects outside. The rule (T-perObN) states the establishment of well-formed neighbourhood relationship between two objects, which adds the relationship details to the environment N.

4.2 Typing

The typing rules are given in the Table 3. An environment Γ and Δ is a finite mapping from variables to types, written as $\overline{\mathrm{x}}\colon \overline{\mathrm{C}}$ and contains ownership ordering respectively; a neighbourhood environment N is a finite binary mapping on the

(T-Owner)

$$\frac{\Delta; N;\; \Gamma \;\vdash\; rep(c_{ob}) \;=\; \{\overline{e}\}}{\Delta; N;\; \Gamma \;\vdash\; \overline{e} \;\prec\colon\; c_{ob}}$$

(T-perObN)

$$\frac{\Gamma; \Delta; \vdash\; neighbourhood(c_{ob}) \;=\; \{d_{ob}\} \;\epsilon\; N}{\Gamma; \Delta; \vdash\; (c_{ob} \overset{\bullet}{\mapsto} d_{ob}) \;\epsilon\; N}$$

(T-subtreeObN)

$$\frac{\Gamma; \Delta; \vdash neighbourhood(c_{ob}) = \{subtree(d_{ob})\} \;\epsilon\; N}{\Gamma; \Delta; \vdash\; (c_{ob} \overset{\#}{\mapsto} d_{ob}) \;\epsilon\; N}$$

(T-subtree*ObN)

$$\frac{\Gamma; \Delta; \vdash neighbourhood(c_{ob}) = \{subtree^*(d_{ob})\} \;\epsilon\; N}{\Gamma; \Delta; \vdash\; (c_{ob} \overset{*}{\mapsto} d_{ob}) \;\epsilon\; N}$$

Table 3: Trusted Java (typing)

set of variables used to infer the ***neighbourhood relationship*** between peer objects at various levels of the ownership tree. The neighbourhood environment N can be empty. Functions $sOwner(c_{ob})$, $dOwner(c_{ob})$, and $tOwner(c_{ob})$ return the static, dynamic, and trusted owner of the object c_{ob} respectively.

The rule (T-Owner) gives the ***within*** relationship, where $d \prec: c_{ob}$ states that *the object d is within the ownership boundary of the object* c_{ob}. The rule (T-subtreeObN) specifies establishment of well-formed neighbourhood relationship between an object and its rep context by adding them to the neighbourhood type environment. The rule (T-subtree*ObN) specifies establishment of well-formed neighbourhood relationship between an object and its subtree context recursively until leaf-node and then adding the relationship to the neighbourhood type environment N.

4.3 Declassification Mechanisms

In this section we define the declassification mechanism and the trusted neighbourhood properties.

(Declassification)

$$\frac{\begin{array}{c} fields(D\langle P\rangle) = D_1\ e_{ob};\ \overline{\mathrm{M}} \quad fields(C\langle O\rangle) = D\ d_{ob}; \quad [?][this]D_1\ m()\{\ return\ e_1;\ \}\ \epsilon\ \overline{\mathrm{M}} \\ N;\ \Gamma\ \vdash\ c_{ob}\colon C\langle O\rangle; \quad N;\ \Gamma\ \vdash\ d_{ob}\colon D\langle this\colon C\rangle; \quad N;\ \Gamma\ \vdash\ e_{ob}\colon C\langle this\colon D\rangle; \\ N;\ \Gamma\ \vdash\ c_{ob};\ \prec:\ world; \quad N;\ \Gamma\ \vdash\ d_{ob}\ \prec:\ c_{ob} \quad N;\ \Gamma\ \vdash\ e_{ob}\ \prec:\ d_{ob}; \end{array}}{\begin{array}{c} N;\ \Gamma\ \vdash\ declassify\ \ e_{ob}\ \epsilon\ e_1 \Rightarrow\ tOwner(e_{ob}) = \{d_{ob}\}\ ,\ dOwner(e_{ob}) = \{c_{ob}\} \\ \Rightarrow\ e_{ob}\ \prec:\ c_{ob} \end{array}}$$

The rule (Declassification) specifies that the declassification of an object results in a change in the declassified object's trusted and dynamic owners. For example, the dynamic owner of the declassified object e_{ob} changes from d_{ob} to c_{ob} object and the trusted owner changes to d_{ob} object.

Declassification with Per-Object Neighbourhood Next rule defines per-object neighbourhood. Here, object d_{ob} establishes a trusted per-object neighbourhood relationship with the b_{ob} object, which enables access permission for the object b_{ob} over e_{ob} object. However, nodes on the subtree of the object d_{ob} (i.e. $e1_{ob}$) and on the subtree of the object b_{ob} (i.e. $b1_{ob}$) and the object a_{ob} cannot access the declassified e_{ob} object because of the absence of neighbourhood relationship.

(Per-Object Neighbourhood)

$$\frac{\begin{array}{c} N = \varnothing;\ \Gamma\ \vdash\ c_{ob}\colon C\langle O\rangle \quad N = \varnothing;\ \Gamma\ \vdash\ c_{ob}\ \prec:\ world \quad rep(c_{ob}) = \{\ d_{ob},\ b_{ob},\ a_{ob}\ \} \\ rep(d_{ob}) = \{\ e1_{ob},\ e_{ob}\ \} \quad rep(b_{ob}) = \{\ b1_{ob}\ \} \quad peer(d_{ob}) = \{\ b_{ob},\ a_{ob}\ \} \quad neighbourhood(d_{ob}) = \{\ b_{ob}\ \} \\ N = \{b_{ob}\};\ \Gamma\ \vdash\ declassify\ \ e_{ob} \end{array}}{N = (d_{ob} \overset{\bullet}{\mapsto} b_{ob});\ \Gamma\ \vdash\ canAccess(b_{ob},\ e_{ob})\ \ is\ \ TRUE}$$

Declassification with Context Level Neighbourhood The rep-context neighbourhood rule defines the rep context neighbourhood relationship. For example, the e_{ob} object after declassification has object d_{ob} as its trusted owner and object

c_{ob} as its dynamic owner, where the trusted owner d_{ob} establishes subtree neighbourhood relationship with b_{ob}. This enables b_{ob} and its rep context to access the declassified object e_{ob}.

(Rep-Context Neighbourhood)

$$\frac{\begin{array}{c} N = \varnothing;\ \Gamma \vdash c_{ob} : C\langle O\rangle \quad N = \varnothing;\ \Gamma \vdash c_{ob} \prec: world \\ rep(c_{ob}) = \{\, d_{ob},\, b_{ob},\, a_{ob} \,\} \quad rep(d_{ob}) = \{\, e_{ob} \,\} \quad rep(b_{ob}) = \{\, b1_{ob} \,\} \\ rep(b1_{ob}) = \{\, b2_{ob} \,\} \quad peer(d_{ob}) = \{\, b_{ob},\, a_{ob} \,\} \quad neighbourhood(d_{ob}) = \{\, subtree(b_{ob}) \,\} \\ N = (d_{ob} \overset{\#}{\mapsto} b_{ob});\ \Gamma \vdash declassify\ \ e_{ob} \end{array}}{N = (d_{ob} \overset{\#}{\mapsto} b_{ob});\ \Gamma \vdash canAccess(b_{ob},\, e_{ob})\ \ is\ \ TRUE\,,}$$

The rule for defining the establishment of recursive subtree (i.e. subtree*) neighbourhood, is given next. Here, d_{ob} establishes a trusted subtree* neighbourhood relationship with the b_{ob}, which enables access permission for b_{ob} and recursively all its subtree on e_{ob}.

(Subtree* Neighbourhood)

$$\frac{\begin{array}{c} N = \varnothing;\ \Gamma \vdash c_{ob} : C\langle O\rangle \quad N = \varnothing;\ \Gamma \vdash c_{ob} \prec: world \\ rep(c_{ob}) = \{\, d_{ob},\, b_{ob},\, a_{ob} \,\} \quad rep(d_{ob}) = \{\, e_{ob} \,\} \quad peer(d_{ob}) = \{\, b_{ob},\, a_{ob} \,\} \\ neighbourhood(d_{ob}) = \{\, subtree^*(b_{ob}) \,\} \\ N = (d_{ob} \overset{*}{\mapsto} b_{ob});\ \Gamma \vdash declassify\ \ e_{ob} \end{array}}{\Delta; N = (d_{ob} \overset{*}{\mapsto} b_{ob});\ \Gamma \vdash canAccess(b_{ob},\, e_{ob})\ \ is\ \ TRUE\,,}$$

5 Discussion and Related Work

Ownership types encapsulation [25,9,10,6] is a powerful technique for protection against aliases at the object reference level. The two flavors of ownership types are owners-as-dominators and owners-as-modifiers which enforce static ownership mechanisms [12]. Enforcing restrictions on object references simplifies reasoning about the program behaviour including modular verification of programs [17,23,13]. The other flavor of encapsulation is by providing restricted access on certain objects based on their type annotations [1,29]. However, when dealing with real world applications, all these techniques are inflexible and static, where the owner of an object remains fixed throughout the lifetime of an object. Thus it limits the expressiveness of the ownership types.

To overcome this restriction and to decide the accessibility of an object dynamically, owners-as-downgraders was proposed in [22] permitting safe dynamic declassification of objects. In this model, accessibility of an object is determined by its dynamic owner rather than the static owner. Thus after declassification, the declassified object will become accessible to every peer object present in the dynamic owner's context. This again may not be suitable for every scenario, for example, ones like the medical study example [4], where the application requires

the declassified object to be accessible only to certain trusted objects and not others.

In this paper, we have proposed a solution for the above problem, by defining the trusted ownership mechanism, where application developers can specify sharing of objects at run-time only to certain trusted neighbourhood contexts. Unlike the declassification model presented in [22], our type system enforces trusted access to the declassified object by decoupling accessibility and visibility from the ownership mechanism and enables application developers to explicitly declassify objects for dynamic information sharing and specify multiple trusted neighbourhoods to whom accessibility of the declassified objects is allowed.

Similarly, Ownership Domains [2] is a flexible system which decouples encapsulation policy from ownership mechanism, however, it does not support dynamic exposure of objects where the visibility of the object changes dynamically. Compared to our system, we have decoupled visibility and encapsulation policy from ownership mechanism as stated above.

The other model for sharing of objects is through shared ownership [20,19]. MOJO [20] and Mojojojo [19] systems support multiple ownerships and sharing between two objects is determined by intersection or union of representations of these objects. The owners-as-ombudsmen [26] allows restricted sharing where objects at the peer level can access shared object by collaboratively defining an aggregate containing this shared object. However, exposing previously protected object to one level above by downgrading or declassification of object's owner may require additional constraints for dynamic security.

Other related works pertain to ownership transfer including systems where an object can change its owner dynamically at run time. The support for ownership transfer based on uniqueness [8] increases the complexity of the program understanding. The External Uniqueness [11], used to remove the problems faced by unique reference by transferring the externally unique reference of the aggregate object. The other approach is using object invariants [17,18], and the work in [5] deals with ownership transfer using separation logic, where the permission is transferred between concurrent threads. The ownership transfer gives flexibility to design dynamic system, however it becomes hard to enforce the property statically or may require some form of forced uniqueness constraint.

The work on Universe Types with Transfer (UTT) [24] supports ownership transfer while still permitting external temporary aliases on the transferable cluster of objects. The ConstrainedJava [14] provides ownership transfer within the ownership tree wherein the exported object occupies the same location in the ownership tree as its former owner, which makes several objects to jointly own an object. Since our work in this paper focuses on the trusted information sharing in the ownership types environment, it can be adapted to UTT[24] and ConstrainedJava[14] by similarly adding neighbourhood relationship at the owners' level.

6 Conclusion and Future Work

In this paper, we have presented a trusted ownership mechanism to enable application developers to specify sharing of objects among trusted neighbourhood at run-time. Illustrative example captures the environmental factors requiring object sharing and trusted access to the shared objects.

As a future work we need to investigate how to add unanticipated object evolution and delegation logics [28] into the proposed framework. By combining unanticipated delegation logics and ownership security, one could enable both software design flexibility and ownership-based security.

References

1. Aldrich, Kastadinov V. and Chambers C.: Alias annotations for program understanding. In Proceedings of the 17th ACM SIGPLAN conference, OOPSLA, ACM, Vol. 37(11), pp. 311-330 (2002)
2. Aldrich, J. and Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In Proceedings of ECOOP, LNCS Vol.3086, Springer, Berlin, pp.1-25 (2004).
3. Almeida P.S.: Balloon Types: Controlling Sharing of State in Data Types. In Proceedings of ECOOP, Springer-Verlag, pp. 32-59 (1997).
4. Andrew C. Myers., Barbara Liskov.: A Decentralized Model for Information Flow Control. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, October (1997). pp. 387-411 (2005).
5. Bornat R., Calcagno C., O'Hearn P. and Parkinson M.: Permission Accounting in Separation Logic. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on POPL, ACM, pp. 259-270 (2005).
6. Boyapati C., Liskov B. and Shrira L.: Ownership Types for Object Encapsulation. In ACM SIGPLAN Notices, ACM, Vol. 38(1), pp. 213-223 (2003).
7. Boyapati C. and Rinard M.C.: Safejava: a unified type system for safe programming. MIT (2004).
8. Boyland J. and Retert W.: Connecting Effects and Uniqueness with Adoption. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on POPL, ACM, pp. 283-295 (2005).
9. Clarke D.G., Potter J., and Noble J.: Ownership Types for Flexible Alias Protection. In Proceedings of the 13th ACM SIGPLAN conference on OOPSLA, ACM, pp. 48-64 (1998).
10. Clarke D.G. and Drossopoulou S.: Ownership, Encapsulation and the Disjointness of Type and Effect. In Proceedings of the 17th ACM SIGPLAN conference on OOPSLA, ACM, pp. 292-310 (2002).
11. Clarke D.G. and Wrigstsad T.: External Uniqueness is Unique Enough. In Proceedings of European Conference on ECOOP, Springer, LNCS Vol. 2743, pp. 59-67 (2003).
12. Clarke D., Östlund J., Sergey I., Wrigstad T.: Ownership Types: A Survey. In Aliasing in Object-Oriented Programming. Types, Analysis and Verification, LNCS, Vol. 7850, pp. 15-58 (2013).
13. Dietl W. and Muller P.: Universes: Lightweight Ownership for JML. In JOT, Vol. 4(8), pp. 5-32 (2005).

14. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In DLS 2007: Proceedings of the 2007 Symposium on Dynamic Languages, pp. 41-52 (2007).
15. Hogg J.: Islands: Aliasing Protection in Object-Oriented Languages. In ACM SIGPLAN Notices, ACM, Vol. 26(11),pp. 271-285 (1991)
16. Hogg J., Lea D., Wills A., deChampeaux D. and Holt R.: The Geneva Convention on the Treatment of Object Aliasing. In ACM SIGPLAN OOPS Messenger, ACM, Vol.3(2), pp. 11-16 (1992).
17. Leino K.R.M. and Muller P.: Object Invariants in Dynamic Contexts. In Proceedings of ECOOP, LNCS Vol. 3086, Springer-Verlag, pp. 491-516 (2004).
18. Leino K.R.M. and Muller P.: Modular Verification of Static Class Invariants. In FM 2005: Formal Methods, International Symposium of Formal Methods Europe, LNCS Vol. 3582, Springer, pp. 26-42 (2005).
19. Li, P., Cameron, N., Noble, J.: Mojojojo - more ownership for multiple owners. In International Workshop on FOOL (2010).
20. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In Proceedings of OOPSLA, pp. 441-460 (2007).
21. Lu, Y., Potter, J.: On Ownership and Accessibility. In: ECOOP 2006. LNCS Vol. 4067, pp 99-123 (2006).
22. Lu, Y., Potter, J., Xue, J.: Ownership Downgrading for Ownership Types. In: APLAS 2009. LNCS Vol. 5904, pp. 144-160 (2009).
23. Muller P. and Poetzsch-Heffter A.: Universes: A Type System for Controlling Representation Exposure. In Programming Languages and Fundamentals of Programming, Poetzsch-Heffter A. and Meyer J. (eds.), Fer-nuiversitat Hagen, Technical Report 263 (1999).
24. Muller P. and Arsenii Rudich.: Ownership transfer in universe types. In Proceedings of the 22nd OOPSLA, pp.461-478 (2007).
25. Noble J., Vitek J. and Potter J.: Flexible Alias Protection. In Proceedings of the 12th ECOOP, LNCS Vol. 1445, pp. 158-185 (1998).
26. Östlund, J., Wrigstad, T.: Multiple Aggregate Entry Points for Ownership Types. In ECOOP 2012, LNCS Vol. 7313, pp.156-180 (2012).
27. Potter J., Noble J., Clarke D.:The Ins and Outs of Objects. In Proceedings of ASWEC, p.80 (1998).
28. Salah Sadou., Hafedh Mili.: A delegation-based approach for the unanticipated dynamic evolution of distributed objects. In Journal of Systems and Software, Vol. 82(6), pp.932-946 (2009).
29. Vitek J. and Bokowski B.: Confined Types. In Proceedings of the 14th ACM SIGPLAN conference on OOPSLA, ACM, pp. 82-96 (1999).