Representation Independence, Confinement and Access Control

Anindya Banerjee and David Naumann

ab@cis.ksu.edu and naumann@cs.stevens-tech.edu

Kansas State University and Stevens Institute of Technology
public class Class {
    private Identity[] signers; //authenticated
    public Identity[] getSigners() {
        return signers;
    }
...
}
public class System {
    public Identity[] getKnownSigners() { ... }
    ...
}
class Bad {
    void bad() {
        Identity[] s = getSigners(); //leak
        s[0] = System.getKnownSigners()[0];
        doPrivileged("something bad");
    }
    ...
}
Representation independence

class A {
    private Boolean g;  // rep object
    unit init(){
        g := new Boolean();
        g.set(~true);
    }
    unit setg(bool x){
        g.set(~x);
    }
    bool getg(){
        return ~g.get();
    }
}

Example: abstraction A using representation Boolean to hold current value (or its negation).

*Information hiding*: type safety, visibility and scope rules ensure that clients are not dependent on encapsulated representation.

z := new A(); z.setg(true); b := z.getg();
class A {
    private Boolean g; // rep object
    unit init() { g := new Boolean();
        g.set(~true); }
    unit setg(bool x) { g.set(~x); }
    bool getg() { return ~g.get(); }
    Object bad() { return g; }
}

Client behavior depends on representation:

z := new A(); w := (Boolean) z.bad();
if (w.get()) skip else diverge;
Representation exposure

class A {
    private Boolean g; // rep object
    unit init() { g := new Boolean();
    // g.set(~true); }
    unit setg(bool x) { g.set(~x); }  
    bool getg() { return ~g.get(); } 
    Object bad() { return g; }
}

Client behavior depends on representation:
    z := new A(); w := (Boolean) z.bad();
    if (w.get()) skip else diverge;

Leaks also allow clients to violate invariants, e.g.,
“signers have all been authenticated for this class”.
**Contribution**

Formalization of pointer confinement and proof that it ensures representation independence, for rich fragment of Java.

<table>
<thead>
<tr>
<th>Client objects</th>
<th>Interface objects</th>
<th>Representations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>Boole</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>Boole</td>
</tr>
</tbody>
</table>

allowed

↓

disallowed
Contribution

Formalization of pointer confinement and proof that it ensures representation independence, for rich fragment of Java.

- Justify *component replacement*: in software engineering (e.g., optimizing transformations, refactoring) and in theory (e.g., equivalence of lazy and eager access control).

- *Modular verification*: reason about component in terms of abstract interface spec.

- Secure *information flow* and other program analyses based on abstract interpretation.
Language

- pointers to mutable objects (but no ptr. arithmetic)
- subclassing, dynamic dispatch, type-cast and -test
- class-based visibility control
- recursive types and methods
- privilege-based access control

Major omissions: exceptions, threads, class loading and reflection.
Language

- pointers to mutable objects (but no ptr. arithmetic)
- subclassing, dynamic dispatch, type-cast and -test
- class-based visibility control
- recursive types and methods
- privilege-based access control

Straightforward compositional semantics:
- object state contains locations and prim. vals.
- heap maps locations to object states
- methods bound to classes, not objects
- commands denote functions

\[
\text{method-meanings} \rightarrow \text{envir} \rightarrow \text{heap} \rightarrow (\text{envir} \times \text{heap})^\bot
\]
Heap confinement for $A, \text{Rep}$

$\text{conf } h$ iff $h$ has admissible partition

$h = h_{\text{Out}} \ast h_{A_1} \ast h_{\text{Rep}_1} \ast \ldots \ast h_{A_n} \ast h_{\text{Rep}_n}$ with $h_{\text{Out}} \not\leadsto h_{\text{Rep}_k}, \quad h_{\text{Rep}_k} \not\leadsto h_{\text{Out}}, \quad \text{and}$

$h_{A_k} \ast h_{\text{Rep}_k} \not\leadsto h_{A_j} \ast h_{\text{Rep}_j}$ for $k \neq j$
Confinement

- Commands and method meanings preserve heap confinement; corresponding conditions on expressions and environments.
Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.
- *Semantic definition*; static analysis separate concern.
Confinement

- Commands and method meanings preserve heap confinement; corresponding conditions on expressions and environments.

- **Semantic definition**; static analysis separate concern.

- Signatures \((C, (x : T) \rightarrow T)\) confined:
  - \(C \leq A\) implies \(\overline{T} \not\subseteq Rep \land \overline{T} \not\subseteq A\)
  - \(C \not\subseteq A \land C \not\subseteq Rep\) implies \(\overline{T} \not\subseteq Rep\)

Methods not satisfying these conditions would violate heap confinement or ignore their arg’s.
Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.

- **Semantic definition**; static analysis separate concern.

- Signatures \((C, (\overline{x}: \overline{T}) \rightarrow T)\) confined:
  - \(C \leq A\) implies \(\overline{T} \not\equiv Rep \land \overline{T} \not\equiv A\)
  - \(C \not\equiv A \land C \not\equiv Rep\) implies \(\overline{T} \not\equiv Rep\)

  Methods not satisfying these conditions would violate heap confinement or ignore their arg’s.

- Semantic confinement can be ensured by simple syntactic checks similar to ones in literature.
**Simulation**

**Basic simulation**

Classes $A$, $Rep$, $Rep'$ and confined class table $CT$ with

$CT(A) = \text{class } A \text{ extends } B \{ \overline{T} \overline{g}; \overline{M} \}$

$CT'(A) = \text{class } A \text{ extends } B \{ \overline{T}' \overline{g}'; \overline{M}' \}$
**Basic simulation**

Classes $A, \text{Rep}, \text{Rep}'$ and confined class table $CT$ with

$CT(A) = \text{class } A \text{ extends } B \{ \overline{T} \, \overline{g}; \overline{M} \}$

$CT'(A) = \text{class } A \text{ extends } B \{ \overline{T}' \, \overline{g}'; \overline{M}' \}$

Relation $R \subseteq [\text{Heap}] \times [\text{Heap}]'$ for a single pair of $A$ objects at same location $\ell$.

$h = hA \ast h\text{Rep}$

$h' = hA' \ast h\text{Rep}'$

![Diagram](image-url)
Simulation

Basic simulation

Classes $A, \text{Rep}, \text{Rep}'$ and confined class table $CT$ with

$CT(A) = \text{class } A \text{ extends } B \{ \overline{Tg}; \overline{M} \}$

$CT'(A) = \text{class } A \text{ extends } B \{ \overline{T'}g'; \overline{M}' \}$

Relation $R \subseteq [Heap] \times [Heap]'$ for a single pair of $A$ objects at same location $\ell$.

$h = hA \ast h\text{Rep}$

$h' = hA' \ast h\text{Rep}'$

Induced relations $\mathcal{R} \theta$

- $\mathcal{R} T d \ d' \iff d = d'$ (primitives and client-visible loc’s)
- $\mathcal{R} \text{Heap} h \ h' \iff \text{partition with } R (hA_k \ast h\text{Rep}_k) (hA'_k \ast h\text{Rep}'_k)$
Main results

Abstraction theorem:
Given basic simulation for confined $CT, CT'$. If every method body of $A$ preserves $R (envir \times Heap)_\bot$ then so does every command.

(Commands in both clients and subclasses of $A$.)
Main results

**Abstraction theorem:**
Given basic simulation for confined \( CT, CT' \). If every method body of \( A \) preserves \( \mathcal{R} (envir \times Heap)_\perp \) then so does every command.

(Commands in both clients and subclasses of \( A \).)

**Identity extension lemma:**
Suppose \( \mathcal{R} (envir \times Heap) (\eta, h) (\eta', h') \). Then

\[
\text{garbage-collect}((\text{rng } \eta), h) = \text{garbage-collect}((\text{rng } \eta'), h'),
\]

if these heaps are both \( A \)-free.

(Can also express in terms of heap visible to clients.)
Access control

Access matrix: $\mathcal{A}(\text{user}) = \{p\}$ and $\mathcal{A}(\text{sys}) = \{p, w\}$.

class Sys signer sys {
    unit writepass(String x) {
        check $w$; write(x, "passfile");
    }
    unit passwd(String x) {
        check $p$; dopriv $w$ in writepass(x);
    }
}

class User signer user {
    Sys s ...
    unit use() { dopriv $p$ in s.passwd("me");
    }
    unit try() { dopriv $w$ in s.writepass("me");
    }
}
Conclusion

Contribution: analysis of information hiding for pointers, subclassing, etc., using simple, extensible denotational semantics.

Ongoing and future work:

- polymorphism (essential to avoid Object)
- static analysis and transformation for access control (proved Fournet&Gordon [POPL02] equiv’s in a denotational semantics for the funct. lang.)
- information flow
- static checking of confinement (sans annotation)
- proof rules for simulation (A’s methods)
- other confinement disciplines (e.g., read-only)
Related work

This paper, with other proof cases: http://www.cs.stevens-tech.edu/~naumann/absApp.ps

A static analysis for instance-based confinement in Java: http://.../static.ps

A simple semantics and static analysis for Java security: http://.../tr2001.ps


D. Clarke, J. Noble, J. Potter: Simple ownership types for object containment, ECOOP’01.


J. Reynolds: Types, abstraction, and parametric polymorphism, Info. Processing ’83

Appendix: static confinement

Signatures: \( C \leq \text{Rep} \Rightarrow U \leq A \lor U \leq \text{Rep} \) for all \( U \in \overline{T} \)

Phrases:

\[
\begin{align*}
C \leq A & \Rightarrow U \not\subseteq A \\
\Gamma; \ C \triangleright e : U & \Rightarrow \Gamma; \ C \triangleright x.f := e \\
C \not\subseteq A & \Rightarrow B \not\subseteq \text{Rep} \\
C \leq A & \Rightarrow B \not\subseteq A \\
\Gamma; \ C \triangleright x := \text{new } B( ) & \Rightarrow \Gamma; \ C \triangleright x := \text{new } B( )
\end{align*}
\]

\[
\begin{align*}
\text{mtype}(m, D) & = (\overline{x} : \overline{T}) \rightarrow T \\
C \leq A & \Rightarrow T \not\subseteq A \\
\Gamma; \ C \triangleright e : D \quad \Gamma; \ C \triangleright \overline{e} : \overline{U} & \Rightarrow \Gamma; \ C \triangleright e.m(\overline{e}) : T
\end{align*}
\]

These suffice for semantic condition stronger than needed for abstraction theorem.
Appendix: parametricity

Simulation is made unsound by rep exposure and also by non-parametric constructs like unchecked casts, &x < &y, sizeof(A), etc. which Java lacks.

Our results hold for any parametric allocator fresh:
- \( \text{loctype}(\text{fresh}(C, h)) = C \) and \( \text{fresh}(C, h) \notin \text{dom } h \)
- \( \text{dom } h_1 \cap \text{locs } C = \text{dom } h_2 \cap \text{locs } C \Rightarrow \text{fresh}(C, h_1) = \text{fresh}(C, h_2) \)

Equal heaps aren’t enough for some equivalences:
\[
\begin{align*}
x &:= \text{new } C(); \\
y &:= \text{new } C(); \\
y &:= \text{new } C(); \\
x &:= \text{new } C();
\end{align*}
\]
So take heaps up to isomorphism, in def of equivalence or in model. Or model with non-det. allocator.
var $x := 0$ in $P(x := x + 2);\ if\ even(x)\ diverge\ else\ skip$

diverge

O-O version with closure as explicit object (with method $x := x + 2$ or $skip$).

Holds because locals $\neq$ objects and name spaces flat. Need confinement if the integer is itself an object.
Appendix: semantic domains

\[ \theta ::= T \mid \Gamma \mid C \text{ state} \mid \text{Heap} \mid (C, (\overline{x} : \overline{T}) \rightarrow T) \mid MEnv \]
Appendix: semantic domains

\[ \theta ::= T | \Gamma | C \text{ state} | \text{Heap} | (C, (\overline{x} : \overline{T}) \to T) | \text{MEnv} \]

\[ \llbracket \text{bool} \rrbracket = \{T, F\} \]
\[ \llbracket C \rrbracket = \{\text{nil}\} \cup \{\ell \in \text{Loc} | \text{loctype } \ell \leq C\} \]

\( \eta \in [\Gamma] \) maps each identifier \( x \) to its value \( \eta x \in [\Gamma x] \)

\( s \in [C \text{ state}] \) maps (declared\&inherited) fields to values

\( h \in [\text{Heap}] \) is partial function on \( \text{Loc} \), with \( h\ell \in \llbracket (\text{loctype } \ell) \text{ state} \rrbracket \)
Appendix: semantic domains

\[ \theta ::= T \mid \Gamma \mid C \text{ state} \mid \text{Heap} \mid (C, (\overline{x} : \overline{T}) \rightarrow T) \mid MEnv \]

\[ [\text{bool}] = \{T, F\} \]
\[ [C] = \{\text{nil}\} \cup \{\ell \in \text{Loc} \mid \text{loctype} \ell \leq C\} \]

\( \eta \in [\Gamma] \) maps each identifier \( x \) to its value \( \eta x \in [\Gamma x] \)

\( s \in [C \text{ state}] \) maps (declared\&inherited) fields to values

\( h \in [\text{Heap}] \) is partial function on \( \text{Loc} \), with \( h\ell \in [((\text{loctype} \ell) \text{ state})] \)

\[ [C, (\overline{x} : \overline{T}) \rightarrow T] = [\overline{x} : \overline{T}, \text{this} : C] \rightarrow [\text{Heap}] \rightarrow ([T] \times [\text{Heap}]) \perp \]

\( \mu \in [MEnv] \) maps each \( C, m \) to \( \mu Cm \in [C, (\overline{x} : \overline{T}) \rightarrow T] \).
Appendix: semantic domains

\[ \theta ::= T \mid \Gamma \mid C \text{ state} \mid \text{Heap} \mid (C, (\overline{x} : \overline{T}) \rightarrow T) \mid M\text{Env} \]

\[[\text{bool}] = \{T, F\}\]
\[[C] = \{\text{nil}\} \cup \{\ell \in \text{Loc} \mid \text{loctype } \ell \leq C\}\]

\(\eta \in [\Gamma]\) maps each identifier \(x\) to its value \(\eta x \in [\Gamma x]\)
\(s \in [C \text{ state}]\) maps (declared\&inherited) fields to values
\(h \in [\text{Heap}]\) is partial function on \(\text{Loc}\), with \(h\ell \in [ (\text{loctype } \ell) \text{ state} ]\)

\[[C, (\overline{x} : \overline{T}) \rightarrow T] = [\overline{x} : \overline{T}, \text{this} : C] \rightarrow [\text{Heap}] \rightarrow ([\Gamma] \times [\text{Heap}])_{\perp}\]

\(\mu \in [M\text{Env}]\) maps each \(C, \text{m}\) to \(\mu C\text{m} \in [C, (\overline{x} : \overline{T}) \rightarrow T]\).

\[[\Gamma; C \vdash e : T] \in [M\text{Env}] \rightarrow [\Gamma] \rightarrow [\text{Heap}] \rightarrow [T]_{\perp}\]
\[[\Gamma; C \vdash S : \text{com}] \in [M\text{Env}] \rightarrow [\Gamma] \rightarrow [\text{Heap}] \rightarrow ([\Gamma] \times [\text{Heap}])_{\perp}\]