ORACLE®

Unifying Access Control & Information Flow

A Security Model for Programs Consisting of Trusted and Untrusted Code

Yi Lu
K. R. Raghavendra
Chenyi Zhang
Paddy Krishnan

ORACLE®

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Background

- Languages designed for internet applications and extensible systems
- Untrusted code may run in the same process as trusted code
- Fine-grained language-based security needed to manage the complex security requirements of program code

Agenda

- Examine stack-inspection based security model
  - Limitations and security requirements
- Propose a new security model to apply access control to enforce secure information flow
  - Dynamic semantics and security property
- Static enforcement of the new security model for OO programs

ORACLE

Stack-based Access Control

- Used in Java and C#, known as sandboxing
  - An implementation of *the principle of least privilege*
- Code attempting sensitive operations may be privileged with permissions
  - Permissions granted to classes by policy files
- **All code on the call stack** must have sufficient privilege to perform specific sensitive operation
  - Permissions tested at runtime

ORACLE

Stack Inspection Example

```
public class A {
    public static void main(String[] args){

        L l = ...;
        ...
        l.createResource(name);
        ...
    }
}
```

```
public class L {
    private Resource resource;

    private Resource create(String name);

    public void createResource(String name)
{

      checkPermission(new
         ResourcePermission(name,"create"));

        resource = create(name);
    }
}
```

ORACLE

Stack Inspection Example

```
public class A {
    public static void main(String[] args){
        L l = ...;
        …
        l.createResource(name);
        ...
    }
}
```

AC.checkPermission	AllPermission
L.createResource	AllPermission
A.main	ResourcePermission("*", "create")

```
public class L {
    private Resource resource;

    private  Resource create(String name);

    public void createResource(String name)
{

        checkPermission(new
          ResourcePermission(name, create"));

         resource = create(name);
    }
}
```

Stack Inspection Unsuccessful: Exception Thrown

```
public class A {
    public static void main(String[] args){
        L l = ...;
        …
        l.createResource(name);
        ...
    }
}
```

```
public class L {
    private Resource resource;

    private  Resource create(String name);

    public void createResource(String name)
{

      checkPermission(new
        ResourcePermission(name,"create"));

        resource = create(name);
    }
}
```

AC.checkPermission	AllPermission
L.createResource	AllPermission
A.main	ϕ

Security Exception

Unauthorised Data Used in Sensitive Operation

```
public class A {
    public static void main(String[] args) {
        L l = ...; B b = ...;
        String name = b.getName();
        l.createResource(name);
        ...
    }
}

public class B {
    public String getName() {
        return "password";
    }
    …
}
```

```
public class L {
    private Resource resource;

    private  Resource create(String name);

    public void createResource(String name)
{

      checkPermission(new
        ResourcePermission(name, "create"));

        resource = create(name);
    }
}
```

B.getName	ϕ
A.main	ResourcePermission("*", "create")

→

AC.checkPermission	AllPermission
L.createResource	AllPermission
A.main	ResourcePermission("*", "create")

ORACLE

Leaked Sensitive Information to Unauthorised Code

```
public class A {
    public static void main(String[] args){
        L l = ...; B b = ...;
        ...
        Resource r = l.getResource();
        b.useResource(r);
    }
}
public class B {
    …
    public void useResource(Resource res) {
... }
}
```

```
public class L {
    private Resource resource;

    …

    public Resource getResource() {

       checkPermission(new
           ResourcePermission("*", "get"));

        return resource;
    }
}
```

AC.checkPermission	AllPermission
L.getResource	AllPermission
A.main	ResourcePermission("*", "create") ResourcePermission("*", "get")

B.useResource	ϕ
A.main	ResourcePermission("*", "create") ResourcePermission("*", "get")

ORACLE

Forbid Desired Operation

```
public class A {
   public static void main(String[] args)  {
      L l = ...;
      …
      l.initResource();
      ...
   }
}
```

```
public class L {
   private Resource resource;

   private Resource create(String name);

   public void createResource(String name) {

     checkPermission(new
         ResourcePermission(name, "create"));

      resource = create(name);
   }
   public void initResource() {
      final String name = "initial";
      createResource(name);
   }
}
```

AC.checkPermission	AllPermission
L.createResource	AllPermission
L.initResource	AllPermission
A.main	ϕ

Security Exception

ORACLE

Limitations of Stack Inspection

- Cannot prevent all information flow attacks
  - E.g. *the confused deputy problem*
    - Untrusted code may inject data used by trusted code to perform sensitive operations
    - Data generated from sensitive operations by trusted code received by untrusted code

- Too strong to allow desired information flows
  - Often have to elevate code privilege at runtime

- Rely on programmer discipline
  - No enforceable security model or policy

ORACLE

Related Work

- Stack-based access control
  - Wallach and Felten, S&P'98
  - Fournet and Gordon, POPL'02
- History-based access control
  - Abadi and Fournet, NDSS'03
- Information-based access control
  - Pistoia, Banerjee and Naumann, S&P'07

- *Hard to state a useful security goal that captures the intent for a general class of trusted and untrusted code*

Informal Security Requirements

- Propagation of information needs to be controlled
  - Data from unauthorised code should not reach sensitive operations
  - Sensitive data should not leak to unauthorised code
- Authorisation determined by the privilege assigned to code
  - Code needs sufficient privilege to send/receive data to/from other code
  - Mutual information flows desirable
- Can classic information security models meet the requirements?

Information Flow Security

- Transfer information between variables according to security levels
  - Each variable assigned a security level (e.g. privilege)
  - Security levels form a lattice: $L \leq H$

- Provide guarantees about information propagation
  - Confidentiality: Do not allow information flows from H to L
  - Integrity: Do not allow information flows from L to H

- Transitive information flow policy precludes cyclic flows between levels
  - A richer information flow structure desired

ORACLE

Overview of the New Security Model

- Each code/variable associated with a dual access control specification
  - A pair of partially ordered security levels
- *Capability* or cap(x) determines privilege/trust of variable x
  - e.g. the privilege granted to untrusted code
- *Accessibility* or acc(x) determines secrecy/sensitivity of variable x
  - e.g. the privilege required by sensitive code
- Information is transferred according to access control specification

Security Model and Java

- Java provides access control but also requires information flow security
  - Stack inspection misses certain information flow based issues
- No clear separation of confidentiality and integrity
  - Programmatically expressed using `checkPermission()`
- Our Model identifies security requirements for Java programs
  - JDK : Trusted: All capabilities
  - JDK: `checkPermission()` : Accessibility requirements
  - Application: capability assigned via policy

Example Revisited

```
@requires{}
@holds{ResourcePermission("*", "create")}
public class A {
    public static void main(String[] args){
        L l = ...; B b = ...;
        String name = b.getName();
        l.createResource(name);
        ...
    }
}

@requires{}
@holds{}
public class B {
    public String getName() {
        return "password";
    }
    …
}
```

```
@requires{}
@holds{AllPermission}
public class L {
    private Resource resource;

    @requires{ResourcePermission(name,
"create")}
    private Resource create(String name);

    public void createResource(String name)
{
      checkPermission(new
         ResourcePermission(name,"create"));

        resource = create(name);
    }
}
```

Informal Security Policy

$$x \to y \implies acc(x) \leq cap(y) \land acc(y) \leq cap(x)$$

- $x \to y$: information may flow from x to y
- Both confidentiality and integrity can be guaranteed
- General information flow policy allows richer flow structure
- Transitive policy in classic model a special case
  - Examples: $acc(x) = cap(y)$, $acc(x) \leq acc(y) \leq cap(x) \leq cap(y)$
  - Such relations too strong

ORACLE

Informal Security Policy

$$x \to y \implies acc(x) \leq cap(y) \land acc(y) \leq cap(x)$$

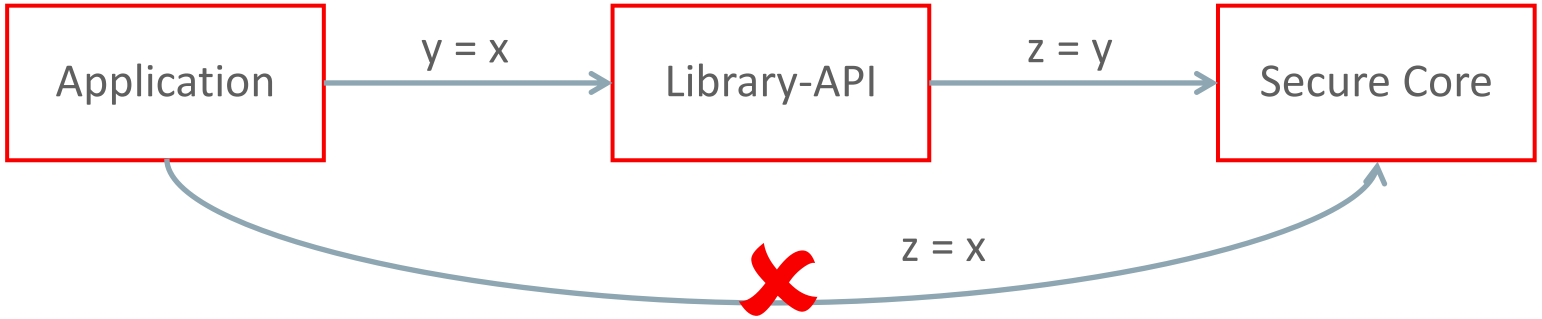
- Confidentiality
  - The receiver must have sufficient privilege to receive the information
- Integrity
  - The sender must have sufficient privilege to send the information
- Mutual information flows supported

ORACLE

Novelty

- Unified treatment of confidentiality and integrity
- Intransitive policy
  - Permits flows across different levels
- Existing lattice-based information flow models use transitive policy
  - Flows only within single level: Anti-symmetry

Example



- Transitive: $\mathrm{label}(x) \leq \mathrm{label}(y) \wedge \mathrm{label}(y) \leq \mathrm{label}(z) \implies \mathrm{label}(x) \leq \mathrm{label}(z)$
- Intransitive Policy
  - $x \rightarrow y \implies acc(x) \leq cap(y), y \rightarrow z \implies acc(y) \leq cap(z)$
  - $acc(x) \nleq cap(z) \implies x \nrightarrow z$

Unauthorised Data Used in Sensitive Operation Revisited

```
@requires{}
@holds{ResourcePermission("*", "create")}
public class A {
    public static void main(String[] args)
{
        L l = ...; B b = ...;
        String name = b.getName();
        l.createResource(name);
        ...
    }
}
@requires{}
@holds{}
public class B {
    public String getName() {
        return "password";
    }
    …
}
```

```
@requires{}
@holds{AllPermission}
public class L {
    private Resource resource;

    @requires{ResourcePermission(name,
"create")}
    private  Resource create(String name);

    public void createResource(String name)
{

      checkPermission(new
        ResourcePermission(name, "create"));

        resource = create(name);
    }
}
```

"password" → name ⟹ {} ≤ {AllPermission} ∧ {ResourcePermission(name,"create")} ≤ {} ✗

Forbid Desired Operation Revisited

```
@requires{}
@holds{}
public class A {
    public static void main(String[] args) {
        L l = ...;
        …
        l.initResource();
        ...
    }
}
```

```
@requires{}
@holds{AllPermission}
public class L {
    private Resource resource;

    @requires{ResourcePermission(name,
"create")}
    private Resource create(String name);

    public void createResource(String name)
{
        checkPermission(new
         ResourcePermission(name, create"));

        resource = create(name);
    }
    public void initResource() {
        final String name = "initial";
        createResource(name);
    }
}
```

"initial" → name ⟹ {} ≤ {AllPermission} ∧ {ResourcePermission(name,"create")} ≤ {AllPermission}

Leaked Sensitive Information Revisited

```
@requires{}
@holds{ResourcePermission("*", "get")}
public class A {
    public static void main(String[] args) {
        L l = ...; B b = ...;
        ...
        Resource r = l.getResource();
        b.useResource(r);
    }
}

@requires{} @holds{}
public class B {
    …
    public void useResource(Resource res) {
... }
}
```

```
@requires{}
@holds{AllPermission}
public class L {
    @requires{ResourcePermission("*",
"get")}
    private Resource resource;

    …

    public Resource getResource() {

        checkPermission(new
           ResourcePermission("*", "get"));
        return resource;
    }
}
```

$$\text{"resource"} \rightarrow \text{res} \Longrightarrow \{\text{ResourcePermission("*","get")}\} \leq \{\} \wedge \{\} \leq \{\text{AllPermission}\}$$

✗

Aims of Formal Security Model

- Extend access control with information flow
- Handle both confidentiality and integrity in intransitive policies
- Proof of security property guaranteed by model

Overview of Formal Security Model

- Access control specification $\varphi \quad ::= \quad \mathcal{A} \cdot \mathcal{C}$
- Union $\mathcal{A}_1 \cdot \mathcal{C}_1 \sqcup \mathcal{A}_2 \cdot \mathcal{C}_2 = \mathcal{A}_1 \vee \mathcal{A}_2 \cdot \mathcal{C}_1 \wedge \mathcal{C}_2$
- Security policy

$$\frac{\mathcal{A}_1 \leq \mathcal{C}_2 \qquad \mathcal{A}_2 \leq \mathcal{C}_1}{\mathcal{A}_1 \cdot \mathcal{C}_1 \ \rhd \ \mathcal{A}_2 \cdot \mathcal{C}_2}$$

$$\frac{\varphi_1 \rhd \varphi \qquad \varphi_2 \rhd \varphi}{\varphi_1 \sqcup \varphi_2 \ \rhd \ \varphi}$$

Overview of Formal Security Model

- Access control subsumption

$$\frac{\mathcal{A}_1 \leq \mathcal{A}_2 \qquad \mathcal{C}_2 \leq \mathcal{C}_1}{\mathcal{A}_1 {\cdot} \mathcal{C}_1 \sqsubseteq \mathcal{A}_2 {\cdot} \mathcal{C}_2}$$

$$\frac{\mathcal{A}_1 \leq \mathcal{A} \qquad \mathcal{A}_2 \leq \mathcal{A} \qquad \mathcal{C} \leq \mathcal{C}_1 \qquad \mathcal{C} \leq \mathcal{C}_2}{\mathcal{A}_1 {\cdot} \mathcal{C}_1 \sqcup \mathcal{A}_2 {\cdot} \mathcal{C}_2 \sqsubseteq \mathcal{A} {\cdot} \mathcal{C}}$$

- Derived access control property

$$\frac{\varphi_1 \sqsubseteq \varphi_3 \qquad \varphi_2 \sqsubseteq \varphi_4 \qquad \varphi_3 \rhd \varphi_4}{\varphi_1 \rhd \varphi_2}$$

ORACLE

Dynamic Semantics of the Security Model

- Big step operational semantics
  - Statements

$$s \ \varphi \ E_1 \ \Downarrow \ E_2$$

  - Expressions: No side-effects

$$e \ E_1 \ \Downarrow \ v \ \varphi$$

Explicit Information Flow

- Reading from variable

$$\frac{S(x) = v\ \varphi}{x\ S\ H\ \Downarrow\ v\ \varphi \sqcup label(x)}$$

- Writing to variable

$$\frac{e\ S\ H\ \Downarrow\ v\ \varphi_1 \qquad \varphi \sqcup \varphi_1 \rhd label(x)}{x{=}e\ \varphi\ S\ H\ \Downarrow\ S[x \mapsto (v\ \varphi \sqcup \varphi_1)]\ H}$$

Information Flow via Heap

- Load

$$\frac{S(x) = l\ \varphi_1 \qquad H(l)(f) = v\ \varphi}{x.f\ S\ H\ \Downarrow\ v\ \varphi_1 \sqcup \varphi \sqcup label(f)}$$

- Store

$$\frac{S(x) = l\ \varphi_0 \qquad y\ S\ H\ \Downarrow\ v\ \varphi_1 \qquad \varphi \sqcup \varphi_1 \rhd label(f)}{x.f{=}y\ \varphi\ S\ H\ \Downarrow\ S\ H[l \mapsto H(l)[f \mapsto (v\ \varphi \sqcup \varphi_0 \sqcup \varphi_1)]]}$$

Implicit Information Flow

- Implicit flow via conditional

$$\frac{x\ E \Downarrow l\ \varphi_0 \qquad s_1\ \varphi\sqcup\varphi_0\ E \Downarrow E_1 \quad s_2\ \varphi\sqcup\varphi_0\ E \Downarrow E_2}{(\text{if } x \text{ then } s_1 \text{ else } s_2)\ \varphi\ E \Downarrow E_1 \uplus E_2}$$

$$\frac{x\ E \Downarrow \text{null}\ \varphi_0 \qquad s_1\ \varphi\sqcup\varphi_0\ E \Downarrow E_1 \quad s_2\ \varphi\sqcup\varphi_0\ E \Downarrow E_2}{(\text{if } x \text{ then } s_1 \text{ else } s_2)\ \varphi\ E \Downarrow E_2 \uplus E_1}$$

  - $E \uplus F$: Value from E, union of flows from E and F

- Implicit flow via dynamic dispatch supported
  - All potential targets considered

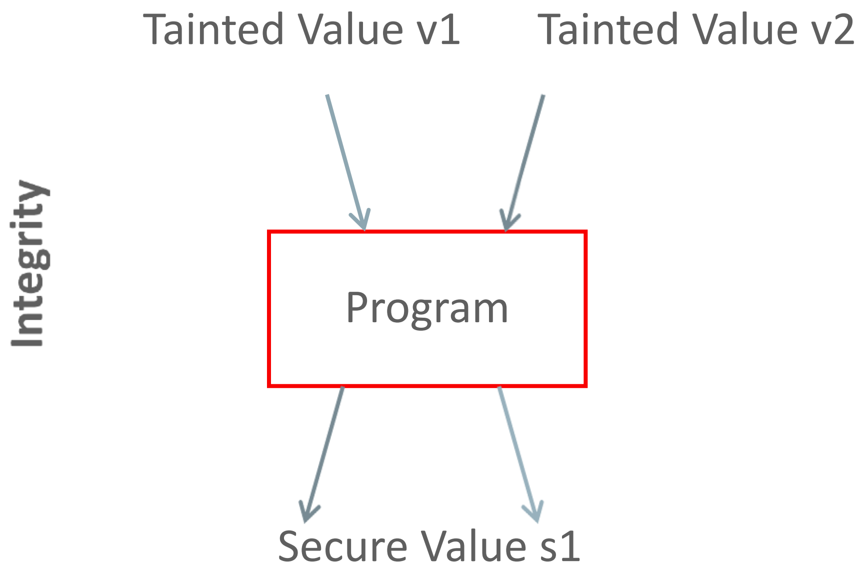
Example: Virtual Dispatch

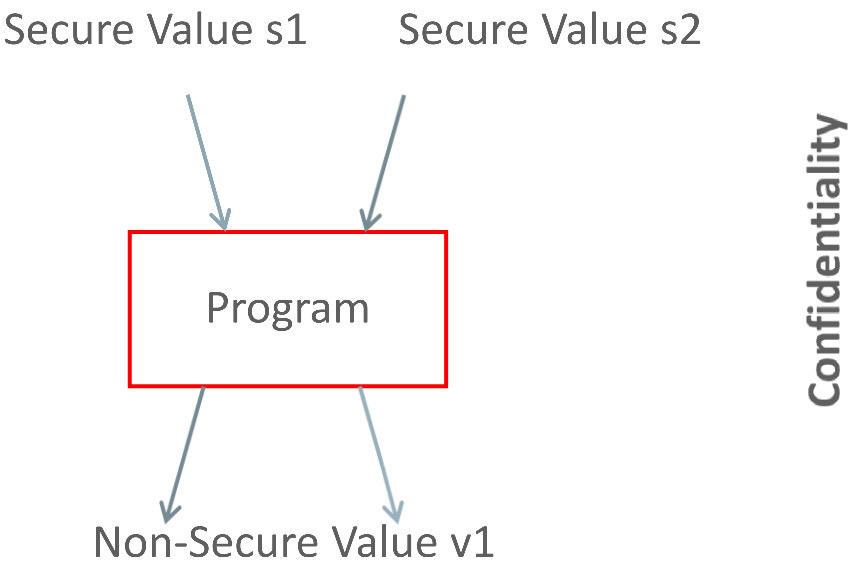
```
class C1 {
   public m(C3 z) {return;}
}
class C2 extends C1 {
  public m(C3 z) {z.f = new T();}
}
class C3 {T f;}
```

```
C1  y = new C2();
if(x)
   y = new C1();
z = new C3();
y.m(z);
// the called m depends on x
// the update on z.f depends on x
```

Noninterference Theorem

- Attacker/system should not be able to distinguish two executions from their outputs with a given access control spec, if they only vary in their inputs with access control specs that are not allowed to access it





Underlying Concepts

- Indistinguishability

$$\frac{\begin{array}{lcl} label(x) = \varphi & \implies & S_1(x) = S_2(x) \\ label(f) = \varphi & \implies & H_1(l)(f) = H_2(l)(f) \end{array}}{S_1\ H_1 \overset{\varphi}{\approx} S_2\ H_2}$$

$$\frac{\begin{array}{lcl} label(x) \rhd \varphi & \implies & S_1(x) = S_2(x) \\ label(f) \rhd \varphi & \implies & H_1(l)(f) = H_2(l)(f) \end{array}}{S_1\ H_1 \overset{\rhd\varphi}{\approx} S_2\ H_2}$$

Noninterference Theorem

- Start states indistinguishable
- States are well-formed
- Executing the same statement in E_1 and E_2 results in indistinguishable states

$$\left.\begin{array}{c} E_1 \overset{\varphi}{\approx} E_2 \\ E_1 \overset{\triangleright\varphi}{\approx} E_2 \\ \vdash E_1 \\ \vdash E_2 \\ s\ \varphi_0\ E_1 \Downarrow E_3 \\ s\ \varphi_0\ E_2 \Downarrow E_4 \end{array}\right\} \Longrightarrow E_3 \overset{\varphi}{\approx} E_4$$

Overview of Static Semantics

- To prove noninterference by static analysis
  - Approximate dynamic semantics with abstract domains
  - Enforce access control policy on the abstract domains

- Defined in type inference rules by $\Gamma\ \Sigma \vdash e : \tau\ \varphi$

- Assignment

$$\frac{\Gamma(x) = \tau\ \varphi_1 \qquad \Gamma\ \Sigma \vdash e : \tau\ \varphi_1 \qquad \varphi \sqsubseteq \varphi_1}{\Gamma\ \Sigma \vdash x{=}e : \varphi}$$

ORACLE

Static Semantics: Field-sensitive

- Load

$$\frac{\Gamma(x) = \tau_0\ \varphi_1 \qquad o \in \tau_0 \qquad \Sigma(o)(f) = \tau\ \varphi}{\Gamma\ \Sigma \vdash x.f : \tau\ \varphi_1 \sqcup \varphi \sqcup label(f)}$$

- Store

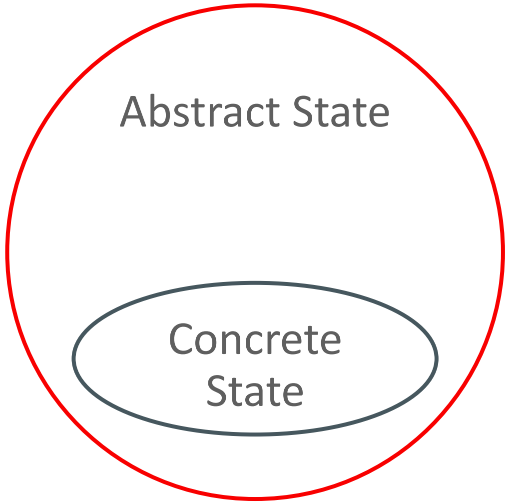
$$\frac{\Gamma(x) = \tau_0\ \varphi_1 \qquad o \in \tau_0 \qquad \Sigma(o)(f) = \tau\ \varphi \qquad \Gamma\ \Sigma \vdash y : \tau\ \varphi}{\Gamma\ \Sigma \vdash x.f{=}y : \varphi_1 \sqcup \varphi}$$

ORACLE

Static Guarantee

- Correspondence between concrete and abstract state

$$
\frac{
\begin{array}{c}
S(x) = v\ \varphi_0 \implies \begin{cases} \Gamma(x) = \tau\ \varphi \\ \{v\} \subseteq \tau \\ \varphi_0 \sqsubseteq \varphi \end{cases} \\
H(l^o)(f) = v\ \varphi_0 \implies \begin{cases} \Sigma(o)(f) = \tau\ \varphi \\ \{v\} \subseteq \tau \\ \varphi_0 \sqsubseteq \varphi \end{cases}
\end{array}
}{
\Gamma\ \Sigma \vdash S\ H
}
$$



Observations

- Dynamic checking impractical
  - Need to track all branches including virtual calls
- Static program analysis provides guarantee
  - Conservative: Can reject safe programs

Summary

- DAC security model: Combines access control and secure information flow
  - General class of trusted and untrusted code
  - Intransitive security policy allows a richer information flow structure

- Prove a general intransitive noninterference property
  - Handles implicit information flow including dynamic dispatch
  - Provide both confidentiality and integrity guarantees

- Security model enforced by static program analysis

ORACLE

Integrated Cloud

Applications & Platform Services

ORACLE®

Distinct Integral/Confidential Requirements

```
@requires{} @holds{AllPermission}
public class A {
   public static void main(String[] args) throws Exception {
      L l = ...; B b = ...; C c = ...;
      l.setResource(b.get());
      Resource r = l.getResource();
      c.use(r);
   }
}

@requires{} @holds{ResourcePermission("*", "set")}
public class B {
   public Resource get() { return new Resource("password"); }

}
@requires{} @holds{ResourcePermission("*", "get")}
public class C {
   public void use(Resource res) { ... }
}
```

```
@requires{} @holds{AllPermission}
public class L {
   @requires_conf{ResourcePermission("*", "get")}
   @requires_inte{ResourcePermission("*", "set")}
   private Resource resource;

   ...
   public Resource getResource() {
      AccessController.checkPermission(
         new ResourcePermission("*", "get"));
      return resource;
   }
   public Resource setResource(Resource r) {
      AccessController.checkPermission(
         new ResourcePermission("*", "set"));
      resource = r;
   }
}
```

ORACLE

Security Policy for Distinct Integrity/Confidentiality

$$x \rightarrow y \implies conf(x) \leq cap(y) \wedge inte(y) \leq cap(x)$$

- The receiver must satisfy the **confidential requirement** of the sender
- The sender must satisfy the **integral requirement** of the receiver

ORACLE

Distinct Integral/Confidential Requirements

```
@requires{} @holds{AllPermission}
public class A {
   public static void main(String[] args) throws Exception {
      L l = ...; B b = ...; C c = ...;
      l.setResource(b.get());
      Resource r = l.getResource();
      c.use(r);
   }
}

@requires{} @holds{ResourcePermission("*", "set")}
public class B {
   public Resource get() { return new Resource("password"); }

}
@requires{} @holds{ResourcePermission("*", "get")}
public class C {
   public void use(Resource res) { ... }
}
```

```
@requires{} @holds{AllPermission}
public class L {
   @requires_conf{ResourcePermission("*", "get")}
   @requires_inte{ResourcePermission("*", "set")}
   private Resource resource;

   ...
   public Resource getResource() {
      AccessController.checkPermission(
         new ResourcePermission("*", "get"));
      return resource;
   }
   public Resource setResource(Resource r) {
      AccessController.checkPermission(
         new ResourcePermission("*", "set"));
      resource = r;
   }
}
```

$$\text{new Resource("password")} \rightarrow \text{resource} \Longrightarrow \{\} \leq \{\text{AllPermission}\}$$
$$\wedge \{\text{ResourcePermission("}*\text{","set")}\} \leq \{\text{ResourcePermission("}*\text{","set")}\}$$

✓

Distinct Integral/Confidential Requirements

```
@requires{} @holds{AllPermission}
public class A {
   public static void main(String[] args) throws Exception {
      L l = ...; B b = ...; C c = ...;
      l.setResource(b.get());
      Resource r = l.getResource();
      c.use(r);
   }
}

@requires{} @holds{ResourcePermission("*", "set")}
public class B {
   public Resource get() { return new Resource("password"); }

}
@requires{} @holds{ResourcePermission("*", "get")}
public class C {
   public void use(Resource res) { ... }
}
```

```
@requires{} @holds{AllPermission}
public class L {
   @requires_conf{ResourcePermission("*", "get")}
   @requires_inte{ResourcePermission("*", "set")}
   private Resource resource;

   ...
   public Resource getResource() {
      AccessController.checkPermission(
         new ResourcePermission("*", "get"));
      return resource;
   }
   public Resource setResource(Resource r) {
      AccessController.checkPermission(
         new ResourcePermission("*", "set"));
      resource = r;
   }
}
```

$$\text{resource} \rightarrow \text{res} \Longrightarrow \{\text{ResourcePermission}(\text{"}*\text{"},\text{"get"})\} \leq \{\text{ResourcePermission}(\text{"}*\text{"},\text{"get"})\}$$
$$\wedge \{\} \leq \{\text{AllPermission}\}$$

✓