



# Link-Time Enforcement of Confined Types for JVM Bytecode

Philip W. L. Fong

`pwl.fong@cs.uregina.ca`

Department of Computer Science

University of Regina

Regina, Saskatchewan, Canada



# Overview

---



- Motivation
- Confined Types
- A Bytecode-level Formulation of Confined Types
- Implementation Efforts
- Secure Cooperation

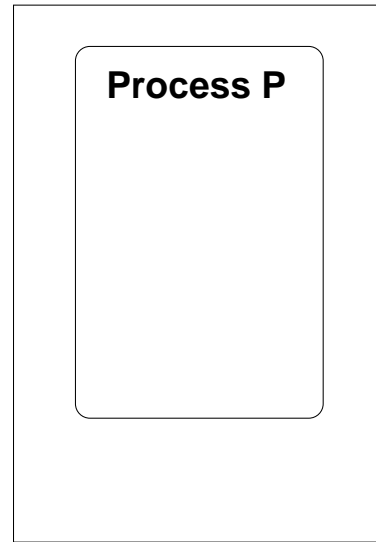




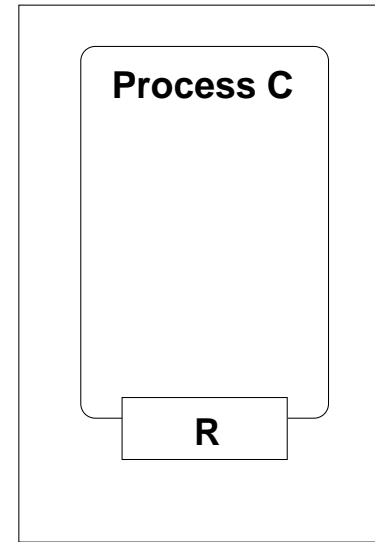
# Motivation



# Dynamically Extensible Software Systems

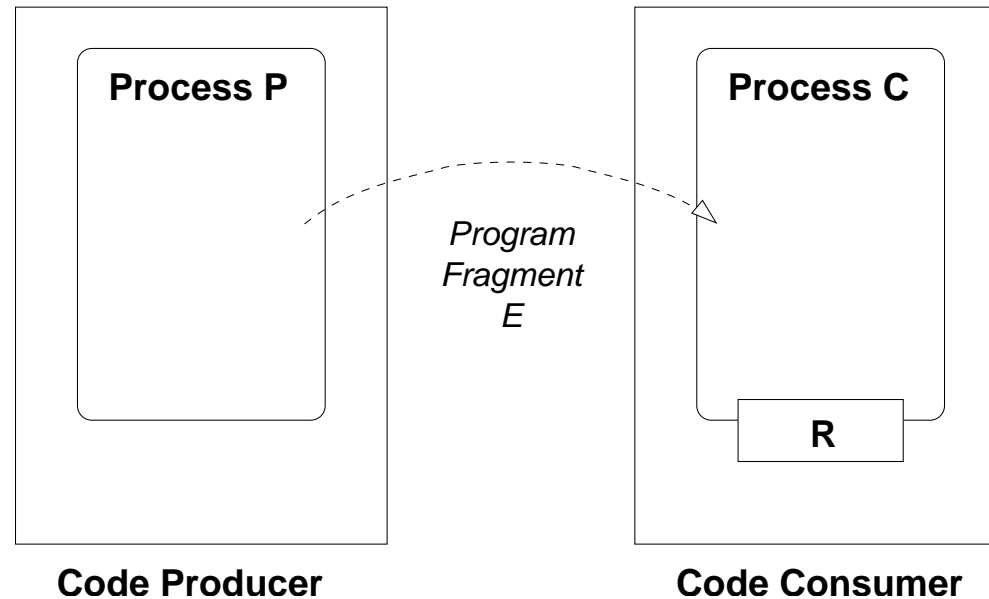


**Code Producer**

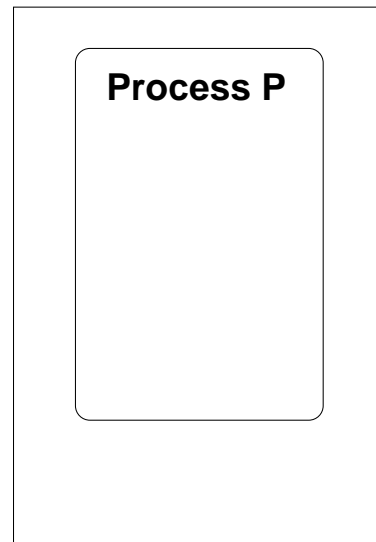


**Code Consumer**

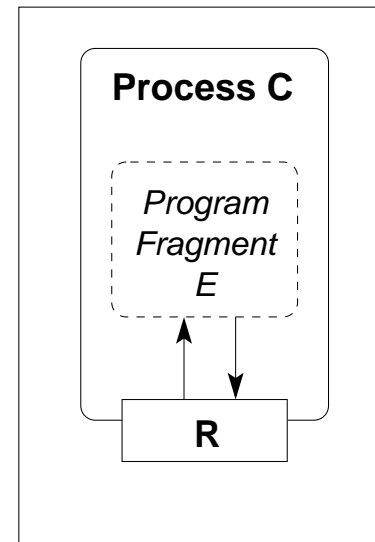
# Dynamically Extensible Software Systems



# Dynamically Extensible Software Systems

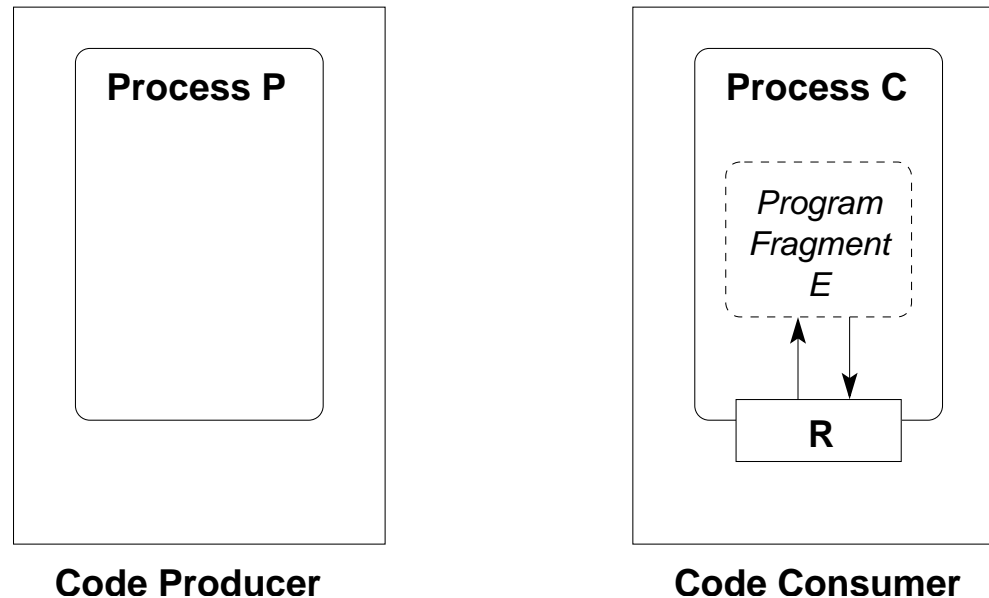


Code Producer



Code Consumer

# Dynamically Extensible Software Systems



## Examples:

mobile code, OS kernel extensions, application plug-ins, scriptable software

# Language-Based Security



## Language-based Security:

- Use a safe language to encode untrusted software extensions
- Protection via programming language facilities
  - e.g., type systems, program rewriting, interpreters
- **Examples:** JVM, CLR





# Encapsulation and Security



## Data Encapsulation

- Protecting object states from undisciplined access
- Well-supported in mainstream OO languages

## Reference Encapsulation

- Preventing accidental reference leaking
- Not supported in mainstream OO languages
- Reference leaking has led to a security breach in JDK 1.1



# Confined Types



## Confined Types (Vitek *et al* 2001, 2003)

- a recently proposed lightweight annotation system for supporting reference encapsulation in Java-like languages
- existing formulations target Java-like source languages
- enforceable only by code producer at compile time
- not qualified as language-based protection mechanism for code consumers



# Contributions



1. the first formulation of confined types for JVM bytecode
2. the first implementation to enforce confined types at link-time on behalf of the code consumer
3. employing the bytecode-level formulation of confined types to facilitate a form of secure cooperation





# Confined Types



# JDK 1.1 Security Breach

```
public class Class {  
    private Identity[] signers;  
    public Identity[] getSigners() {  
        return signers;  
    }  
}
```

# Manual Fix

```
public class Class {
    private Identity[] signers;
    public Identity[] getSigners() {
        Identity[] dup =
            new Identity[signers.length];
        for (int i = 0; i < signers.length; i++)
            dup[i] = signers[i];
        return dup;
    }
}
```

# A New Type Qualifier

- A class can be qualified as being **confined**.
- References to **confined** class instances are not allowed to escape outside of the package in which the class is declared.

- **Examples:**

```
confined class ConfinedIdentity { ... }
```

# Solution (1)

```
public class Identity {
    ConfinedIdentity rep;
    Identity(ConfinedIdentity si) {
        rep = si;
    }
    ...
}
```



# Solution (2)

```
public class Class {
    private ConfinedIdentity[] signers;
    public Identity[] getSigners() {
        Identity[] dup =
            new Identity[signers.length];
        for (int i = 0; i < signers.length; i++)
            dup[i] = new Identity(signers[i]);
        return dup;
    }
}
```



# **A Bytecode-Level Formulation of Confined Types**



# Confined Types as Capabilities (1)

## Capability Types (Boyland *et al* 2001):

- A capability is an unforgeable pair:  
     $\langle \textit{object-reference}, \textit{access-rights} \rangle$
- In a strongly typed programming language, a type qualifier plays the role of the *access-rights* component of a capability:

```
const char *p;
```

# Confined Types as Capabilities (2)

## A Capability-based Formulation of Confined Types:

- In our bytecode-level type system, **confined-ness** is not just the property of a class, it is a capability type.
- Every object reference is associated with a capability type to indicate **where it can be propagated**.
- Subtype hierarchy:

$\perp <: \text{confined} <: \text{anonymous}$

Supertypes are more restrictive than subtypes.

- Greatly simplifies the formulation of type rules.

# Confined Type Interface

- Code safety is a whole-program notion, but ...
  - Lazy, dynamic linking  $\Rightarrow$  not all application classes are loaded at all times.
- Every classfile carries a confined type interface to facilitate **modular type checking**.
- Designed to be backward compatible:
  - Existing classfiles in the Java platform library does not need further annotation.

# Type Rules for Bytecode Instructions

*invokevirtual*  $\langle B.m \rangle$

**Operand Stack:**

$\dots, o, a_1, a_2, \dots, a_k \longrightarrow \dots, v$

**Operation:** Invoke method  $\langle B.m \rangle$  on object instance  $o$ , passing arguments  $a_1, a_2, \dots, a_k$ . Any return value  $v$  is pushed into the operand stack.

**Type Constraints:**

Suppose  $\langle B.m \rangle : T_0(T_1, T_2, \dots, T_k)T \in \mathcal{I}_A$ .

Suppose further that  $o : T_o$ ,  $a_i : T_{a_i}$ , and  $v : T_v$ .

Then  $T_o <: T_0$ ,  $T_{a_i} <: T_i$ , and  $T <: T_v$ .

# Intermodular Type Checking



- Lazy, dynamic linking  $\Rightarrow$  intermodular type checking must be performed incrementally.
- Intermodular type checking is carefully **staged** to dovetail with dynamic linking events.
- Special consideration to preserve **laziness** in dynamic linking.



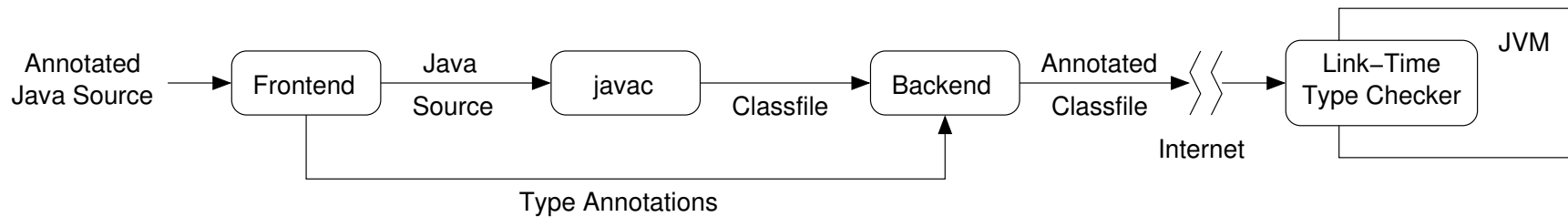


# Implementation Efforts





# Set-Up



## Implementation Experiences:

- Linux command-line tool for annotating classfiles
- Link-time type checker

# Pluggable Verification Modules



## Aegis VM

- an open source research VM for Java
- bytecode verification is a pluggable service
- third-party verification services can be safely incorporated into the dynamic linking process as a **Pluggable Verification Module (PVM)** [OOPSLA'04]

## PVM-based Implementation of Confined Types

- for both intra- and inter-modular type checking
- enforces confined types at link time
- $\approx$  3000 lines of moderately commented C code





# Secure Cooperation



# Secure Cooperation



- Enabling a form of secure cooperation among mutually suspicious code units.
  1. Protection by access contracts
  2. Trust inspiration
  3. Secure software extensions



# Protection via Import Type Annotations

- **Problem:** Alice wants to share a Resource with Bob, but worries that the sharing leads to resource leaking ...

```
package domain;
confined class Resource { ... }
public class Alice {
    static Resource resource = new Resource();
    public static void main(String[] args)
        throws Throwable {
        Class C = Class.forName(args[0]);
        Bob b = (Bob) C.newInstance();
        b.share(resource);
    }
}
public interface Bob {
    void share(Resource r);
}
```

# Protection via Import Type Annotations

- **Solution:** Annotate the classfile of `Bob` with the following export type assertion:

`Bob.share : confined`  $\rightarrow \perp$

Subtypes of `Bob` must conform to this access contract.

# Non-Compliant Extension

```
package domain;
```

```
public class Charlie implements Bob {  
    public static Resource leak;  
    public void share(Resource r) {  
        leak = r;  
    }  
}
```

# Robustness of Trust Inspiration

1. **What if `charlie` falsely asserts a matching export type assertion?**

**Consequence:** PVM fails to confirm compliance of `Charlie.sum` to its promised export type.

⇒ Definition of class `Charlie` will fail.

2. **What if `charlie` does not supply a matching export type assertion?**

**Consequence:** Intermodular type checking will fail.

⇒ Preparation of class `Charlie` will fail.




# Summary & Future Work



## Summary

- the first formulation of confined types for JVM bytecode
- A first implementation to enforce confined types at link time
- Application to secure cooperation

## Future Work

- a static capability type system, **Discretionary Capability Confinement (DCC)**, for Java bytecode
  - a generic framework for defining capability type systems over the Java platform
- 



**Thank You**

