Portable Resource Control in Java: Application to Mobile Agent Security

Walter Binder
CoCo Software Engineering GmbH
Austria

Jarle Hulaas, Alex Villazón, Rory Vidal
University of Geneva
Switzerland
Requirements for Mobile Agent Platforms

• Security
  – protect platform and mobile agents
  – ensure secrecy
  – ensure integrity
  – prevent denial-of-service attacks
  – prevent resource leaks

• Portability

• High performance
Why Java for Mobile Agents?

• Portable code
• Language safety
  – type safety
  – automatic memory management
  – memory protection
  – bytecode verification
• Multithreading
• Class-loader namespaces
• Serialization (weak mobility)
• Availability and performance
Caveats and Pitfalls

- JVM is not an operating system
- No protection domains
- Uncontrolled aliasing
- No ownership information in objects
- Covert channels (e.g., public static variables)
- No resource control
- Single heap
Solution: Portable Resource Control Layer for Java

• Resource control
  – resource accounting
  – enforcing resource limits

• Resources
  – physical (e.g., memory, CPU)
  – logical (e.g., threads)
  – communication (e.g., with services)

• Pure Java
  – no dependence on operating system
  – standard JVMs (state-of-the-art JIT compilers)
  – Java processors (e.g., embedded systems)
Benefits

• Prevention against denial-of-service attacks
• Billing for resource consumption
• Quality of service guarantees
• Monitoring and profiling of mobile agents
Implementation

• Accountability (objects belong to 1 domain)
• Reification of physical resources
  – bytecode rewriting
  – memory: check before allocation (JRes)
  – CPU: count bytecode instructions
• Compensation for native code
  – class-loading
  – deserialization
  – reflection
Memory Account (Simplified)

- Memory limit for multiple threads
- Synchronization
- Weak references to allocated objects

```java
public final class Mem {
    public static Mem getOrCreate();

    public void setLimit(int limit);  
    public void checkAllocation(int size)
        throws ResourceOveruseException;
    public void register(Object o);
}
```
CPU Account

• Separate account for each thread
• No synchronization
• Periodic scheduler thread (high priority)

```java
public final class CPU {
    public static CPU getOrCreate();

    public volatile int usage;
}
```
Passing Accounting Objects

• Unmodified method

Object f(int x) {
    if (x < 0) return null;
    else return new Foo(g(x));
}

• Added accounting objects

Object f(int x, Mem mem, CPU cpu) {
    if (x < 0) return null;
    else return new Foo(g(x, mem, cpu), mem, cpu);
}
Object f(int x, Mem mem, CPU cpu) {
    if (x < 0) return null;
    else {
        int y = g(x, mem, cpu);
        mem.checkAllocation(SIZEOF_FOO);
        Object o = new Foo(y, mem, cpu);
        mem.register(o);
        return o;
    }
}
Rewriting for CPU Accounting

```java
Object f(int x, Mem mem, CPU cpu) {
    cpu.usage += 8;
    if (x < 0) {
        cpu.usage += 8;
        return null;
    }
    else {
        cpu.usage += 26;
        int y = g(x, mem, cpu);
        mem.checkAllocation(SIZEOF_FOO);
        Object o = new Foo(y, mem, cpu);
        mem.register(o);
        return o;
    }
}
```
Accounting Block Analysis

```
0 bipush 10
1 istore_1
2 iconst_1
3 istore_3
4 goto 25
5 invokestatic #2 <Method java.lang.Thread currentThread()>
6 pop
7 ldc2_w #3 <Long 100>
8 bipush 100
9 invokestatic #5 <Method void sleep(long, int)>
10 iinc 3 1
11 iload_3
12 iconst_1
13 if_icmple 10
14 iinc 2 -1
15 goto 44
16 astore_2
17 iconst_0
18 istore 3
19 return

Exception table:
from to target type
3 38 41 <Class java.lang.Exception>
```
class A {
    void x()
    {
        B b = new B();
        b.y();
    }
}

class B {
    void y()
    {
        ...
        ...
    }
}

class A {
    void x()
    {
        B b = new B(CPU.getOrCreate());
        b.y();
    }
}

class B {
    void y()
    {
        ...
        ...
    }
}
Overhead of CPU Accounting

relative execution time

- mtrt
- jess
- compr.
- db
- mpeg
- jack
- javac
- geo. mean

- Ubench-Ujdk
- Rbench-Ujdk
- Rbench-Rjdk
Case study: The J-SEAL2 Mobile Agent Kernel

- Operating system structure
  - isolated protection domains
  - safe domain termination
  - mediated communication
  - resource control
- Small micro-kernel (< 150 KB)
- Portable (pure Java)
- Flexible and extensible
- Efficient and scalable
Nested Protection Domains in J-SEAL2

- Hierarchy of protection domains
- Parent domain controls children
  - communication control
  - resource control
  - termination of sub-hierarchies
- Sandboxes with different privileges
J-SEAL2 System Structure

Services:
- Net
- GUI
- Mail

Root

Stationary Domains:
- Manager
  - Sandbox Trusted
  - Sandbox Anonym

Mobile Agents:
- Agent 1
- Agent 2
- Agent 3
- Agent 4
Hierarchical Resource Control in J-SEAL2

- **Startup**
  - root domain owns all resources

- **Subdomain creation**
  - parent may donate resources to child
  - parent may share resources with child
Resource Donation and Sharing in J-SEAL2

**Trusted domains**

- **CPU**: ∞
- **MEM**: ∞

**Untrusted applications**

- **CPU**: 5%
- **MEM**: 40MB
- **CPU**: 15%
- **MEM**: 10MB

_Resources are shared and split as follows:_

- **CPU**: split 20%
- **MEM**: split 50MB
- **MEM**: split 10MB
- **CPU**: split 75%
Summary

- Mobile agent platform requirements
  - security
  - portability
  - high performance and scalability

- Resource control layer for Java
  - portable (pure Java)
  - reification of physical resources
  - bytecode rewriting
  - moderate overhead