DEFCon: High-Performance Event Processing with Information Security

Matteo Migliavacca, Ioannis Papagiannis, Peter Pietzuch
Imperial College London

David M. Eyers, Jean Bacon
Cambridge Computer Laboratory

Brian Shand
National Health Service, UK

migliava@doc.ic.ac.uk
Event Stream Processing Needs Strong Security

Event processing
- Stream of messages transformed in near real-time by processing units
- Confidential information: healthcare, social networks, finance

Problem: incorrect event flows
- Lead to security violations
- Within application ①, with the environment ②
- Possible causes: bugs, security attacks, third party code, malicious code
Financial Processing: Security and Latency

- Market data processing and local brokering

**Security is important**
- Data is valuable: banks fined for exploiting client information

**Performance constraints**
- Latency, Throughput

**Shared Platform**
- Processing near stock exchanges costly:
  - share resources, reduce entry costs for small firms
- Local brokering to avoid transaction fees and trade anonymously
Security Approach: Information Flow Control

- Protect data end-to-end: Information Flow Control (IFC):
  - Don’t try to eliminate all bugs ① and ② (hard!)
  - Track and control information flows in application
  - Previously applied to operating systems and programming languages

Goal: apply IFC to current high-performance event processing systems
Contributions and Overview

Decentralized Event Flow Control (DEFC) model
- IFC applied to event processing

DEFCon high-performance implementation
- Safe and efficient event flows in Java

Practical isolation methodology
- Secure production-level language runtimes with low effort (OpenJDK 6)

Evaluation
- Throughput and latency overhead
Event Processing in DEFC

DEFC

Bank Investor

Client Investor 77

Client Monitor

1

event

parts

name | data | S (confidentiality)
--- | --- | ---
command | monitor | \{ \}
stock | IBM | \{client77\}

2

unit can input part iff

S(part) ⊆ S(unit)

unit can output part iff

S(unit) ⊆ S(part)

name | data | S (confidentiality)
--- | --- | ---
... | ... | \{client77\}
... | ... | \{client77\}
DEFC Privileges

**Clearance privilege:** receiving confidential information
- Allows units to add tag to its label

**Declassification privilege:** making confidential data public
- Allows units to remove tag from its label

---

**Bank Investor**
- cannot receive confidential information

**Client Investor 77**
- can receive confidential information

**Client Monitor**
- cannot make confidential information public

---

S:{}  

S:{client77}

S:{client77}

---

**client77+**

**client77-**

**client77+**
An Example of Leaks to Avoid

Untainted unit tries to read tainted part
- First try: return access denied
An Example of Leaks to Avoid

- Untainted unit tries to read tainted part
  - First try: return access denied
    - Leaks name of secret parts

### Table: S (confidentiality)

<table>
<thead>
<tr>
<th>name</th>
<th>data</th>
<th>{client77}</th>
</tr>
</thead>
<tbody>
<tr>
<td>FirstLetterIsI</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Diagram:
- Client Monitor
- Bank Investor
- FirstLetterIsI
- Access Denied
- stock=IBM
- S:{client77}
An Example of Leaks to Avoid

- Untainted unit tries to read tainted part
  - First try: return access denied
    - Leaks name of secret parts
  - Second try: update unit label to part label
An Example of Leaks to Avoid

Untainted unit tries to read tainted part
- First try: return access denied
  - Leaks name of secret parts
- Second try: update unit label to part label
  - Secret inferred by absence of communication
An Example of Leaks to Avoid

Untainted unit tries to read tainted part
- First try: return access denied
  - Leaks name of secret parts
- Second try: update unit label to part label
  - Secret inferred by absence of communication
- Solution: avoid implicit label changes, return part not found

Result: all unit label changes must be explicit
- First update label, then read part
Contributions and Overview

Decentralized Event Flow Control (DEFC) model
  – IFC applied to event processing

DEFCCon high-performance implementation
  – Safe and efficient event flows in Java

Practical isolation methodology
  – Secure production-level language runtimes with low effort (OpenJDK 6)

Evaluation
  – Throughput and latency overhead
DEFC assumes units communicate through labelled events

How to control communication between units?
- VM or OS processes: heavy, require copying of data
- Use threads: sharing data in single address space
- Java: mature, pervasive, good performance

How to control communication between Java threads?
Communication: Threads Share Immutable Data

Unit Threads create new objects to put in events
Problem: how to deliver them to receiving units?
  – Copy objects in events
    • Slow
Communication: Threads Share Immutable Data

Unit Threads create new objects to put in events
Problem: how to deliver them to receiving units?
- Copy objects in events
  - Slow
- Transfer references to shared objects
Unit Threads create new objects to put in events
Problem: how to deliver them to receiving units?
- Copy objects in events
  - Slow
- Transfer references to shared objects
  - Problem if unit labels change
Unit Threads create new objects to put in events
Problem: how to deliver them to receiving units?
  – Copy objects in events
    • Slow
  – Transfer references to shared objects
    • Problem if unit labels change

Shared state allows unrestricted communication
  – Solution: only allow immutable objects in event parts
Communication: Shared State in Runtimes

- Bank Investor
- Client Investor
- Client Monitor

DEFCon

- Class Library
- Native
- JVM
- OS

OpenJDK 6
- static fields ~4000
- native methods ~2000
Isolation Methodology Overview

Goal
- Provide isolation between Java Threads
- Secure potentially dangerous targets: static fields and native methods

Previous Java isolation approaches
- Do not support fast message passing between isolates (MVM)
- Use custom Class Libraries and/or JVMs (I-JVM)
- Require extensive analysis of Class Library (KaffeOS, Joe-E)

Our approach
1. Identify potentially dangerous targets using static analysis
2. Modify runtime behaviour of targets using aspect oriented programming (AOP)
3. White-list safe targets
1. Static Analysis

OpenJDK 6

- static fields ~4000
- native methods ~2000
1. Static Analysis

- Bank Investor
- Client Investor
- Client Monitor

DEFCon

OpenJDK 6

- static fields: ~4000, ~2000
- native methods: ~2000, ~1000

Class Library

removed

Native → JVM → OS
1. Static Analysis

- Bank Investor
- Client Investor
- Client Monitor

DEFCon

OpenJDK 6

- static fields
  - ~4000
  - ~2000
  - ~900

- native methods
  - ~2000
  - ~1000
  - ~300

Class Library

Native

JVM

OS
2. AOP Runtime Injection

- Bank Investor
- Client Investor
- Client Monitor

DEFCon

OpenJDK 6
- static fields: ~4000, ~2000, ~900
- native methods: ~2000, ~1000, ~300

Class Library

Native fields: ~4000, ~2000, ~900
Native methods: ~2000, ~1000, ~300

transparent duplication
security checks
3. White-listing

- Bank Investor
- Client Investor
- Client Monitor

OpenJDK 6

- static fields
  ~4000
  ~2000
  ~900

- native methods
  ~2000
  ~1000
  ~300

- transparent duplication
- security checks
- white-listing
# 3. White-listing

<table>
<thead>
<tr>
<th>Target type</th>
<th>Manually white-listed</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for unit execution</td>
<td>for performance</td>
</tr>
<tr>
<td>static fields</td>
<td>27</td>
<td>6</td>
</tr>
<tr>
<td>native methods</td>
<td>15</td>
<td>9</td>
</tr>
</tbody>
</table>

OpenJDK 6

- static fields: ~4000
- ~2000
- ~900
- native methods: ~2000
- ~1000
- ~300

- transparent duplication
- security checks
- white-listing
Isolation Summary

What we achieved
- Secured OpenJDK 6 for running financial scenario
- Required few days of manual work
- Easily applicable to new versions/different JDKs

Limitations
- Assumes knowledge of unit bytecode for static analysis
  - Might need additional effort for new units
- Manual code auditing subject to human errors
Contributions and Overview

Decentralized Event Flow Control (DEFC) model
- IFC applied to event processing

DEFCon high-performance implementation
- Safe and efficient event flows in Java

Practical isolation methodology
- Secure production-level language runtimes with low effort (OpenJDK 6)

Evaluation
- Throughput and latency overhead
Evaluation: Performance Overhead

Measure overhead
- Rate of processed ticks
- Latency of produced deals

Synthetic traces on 6k stock symbols
- Prices set to trigger a deal every 10 ticks

Experiments on dual Intel Xeon E5540 2.53GHz
Acceptable Reduction on Throughput

- Label checks: marginal overhead
- Isolation: $\sim 20\%$ overhead
Low Impact on Latency

Label checks: ~0.5 ms overhead
Isolation: ~1ms overhead
Isolation with Separate JVMs

Comparison with Marketcetera (Open Source trading platform)
- One JVM per investor

Throughput:
- Comparable with DEFCon with few investors
- Does not scale

Latency: around 8 ms
Future Work

Distribution
- Performance limited by number of cores
- Scale DEFCon to multiple engines

Usability
- Correctly assigning labels is hard
- Tools to help design and automatically check labelling

Performance isolation
- Units compete for resources
- Prevent uncooperative behaviours
Conclusion

Event processing requires security and low latency

DEFC model
- Provides strong and fine-grained security by applying Information Flow Control to event processing

DEFCon implementation
- Processes events in single address space for performance
- Provides isolation on production-level language runtimes

Tracking and enforcing security of event flows can be done with reasonable overhead

Thank You! ... Questions?
(migliava@doc.ic.ac.uk)