Monitors and Blame Assignment for Higher Order Session Types

LIMIN JIA, HANNAH GOMMERSTADT, FRANK PFENNING
Distributed System
Distributed System
Contributions

- Use *session types* to dynamically monitor communication between processes to detect undesirable behavior
- Correctly *blame* the party that violated the prescribed communication protocol
Static Checking?

• Need to run checker on each node on code written in different languages
• Unrealistic to assume that will have access to whole computing base
• Use session types as invariants to check dynamically
Process Model

• Processes communicate asynchronously over channels by using message queues
• A process provides a service along a single channel, ex. $\text{proc}(c, P)$
Typing

\[ c_1 : A_1 \quad \ldots \quad c_n : A_n \vdash P :: (c : A) \]

where \( A \) is a session type

A process always provides along a single channel, but it may be a client of multiple channels.
## Session Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c: \tau \land A$</td>
<td>Send $v: \tau$ along $c$, continue as $A$</td>
</tr>
<tr>
<td>$c: \tau \rightarrow A$</td>
<td>Receive $v: \tau$ along $c$, continue as $A$</td>
</tr>
<tr>
<td>$c: 1$</td>
<td>Close channel $c$ and terminate</td>
</tr>
<tr>
<td>$c: A \otimes B$</td>
<td>Send channel $d: A$ along $c$, continue as $B$</td>
</tr>
<tr>
<td>$c: A \rightarrow B$</td>
<td>Receive channel $d: A$ along $c$, continue $B$</td>
</tr>
<tr>
<td>$c: \bigoplus {l_i: A_i}$</td>
<td>Send label $l_i$ along $c$, continue as $A_i$</td>
</tr>
<tr>
<td>$c: &amp;{l_i: A_i}$</td>
<td>Receive label $l_i$ along $c$, continue as $A_i$</td>
</tr>
</tbody>
</table>
# Session Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c: \tau \land A$</td>
<td>Send $\nu: \tau$ along $c$, continue as $A$</td>
</tr>
<tr>
<td>$c: \tau \rightarrow A$</td>
<td>Receive $\nu: \tau$ along $c$, continue as $A$</td>
</tr>
<tr>
<td>$c: 1$</td>
<td>Close channel $c$ and terminate</td>
</tr>
<tr>
<td>$c: A \otimes B$</td>
<td>Send channel $d: A$ along $c$, continue as $B$</td>
</tr>
<tr>
<td>$c: A \multimap B$</td>
<td>Receive channel $d: A$ along $c$, continue $B$</td>
</tr>
<tr>
<td>$c: \bigoplus {l_i: A_i}$</td>
<td>Send label $l_i$ along $c$, continue as $A_i$</td>
</tr>
<tr>
<td>$c: &amp;{l_i: A_i}$</td>
<td>Receive label $l_i$ along $c$, continue as $A_i$</td>
</tr>
</tbody>
</table>
# Session Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c: \tau \land A$</td>
<td>Send $v: \tau$ along $c$, continue as $A$</td>
</tr>
<tr>
<td>$c: \tau \rightarrow A$</td>
<td>Receive $v: \tau$ along $c$, continue as $A$</td>
</tr>
<tr>
<td>$c: 1$</td>
<td>Close channel $c$ and terminate</td>
</tr>
<tr>
<td>$c: A \otimes B$</td>
<td>Send channel $d: A$ along $c$, continue as $B$</td>
</tr>
<tr>
<td>$c: A \leftarrow B$</td>
<td>Receive channel $d: A$ along $c$, continue $B$</td>
</tr>
<tr>
<td>$c: \uplus {l_i: A_i}$</td>
<td>Send label $l_i$ along $c$, continue as $A_i$</td>
</tr>
<tr>
<td>$c: &amp;{l_i: A_i}$</td>
<td>Receive label $l_i$ along $c$, continue as $A_i$</td>
</tr>
</tbody>
</table>
Example

\[ Cam = \&\{ \text{take : photoPerm} \rightarrow \text{picHandle} \odot \text{Cam} \} \]

\[ User = \&\{ \text{picRequest} : \]
\[ \oplus \{ \text{fail : User; success : photoPerm} \odot \text{User} \} \]
System Assumptions

• All processes are **untrusted**
• All monitors are **trusted**
• All message queues are **trusted**
Attacker Model

• Takes control of a process by replacing it by another

$$\text{proc}(c, P)$$
Attacker Model

- Takes control of a process by replacing it by another

\[ \text{proc}(c, P) \rightarrow \text{proc}(c, Q) \]
Attacker Model

- Takes control of a process by replacing it by another

\[ \text{havoc}: \ proc(c, P) \rightarrow proc(c, Q) \]
Attacker Model

• Takes control of a process by replacing it by another

  havoc: \textit{proc} (c, P) \rightarrow \textit{proc} (c, Q)

• Q cannot invent new channels, must have knowledge of existing ones
Monitor Capabilities

- Placed at the ends of each queue, check message as it gets enqueued
- Can ONLY observe communicated values
- No access to process internals
- Raise alarms, which stop computation
Simple Monitor

\[
\text{proc}(a, P) \rightarrow M \leftarrow M \rightarrow \text{proc}(c, Q)
\]

\[c : \text{int} \land A\]

5
Simple Monitor

proc(a, P) \rightarrow M \leftarrow M \rightarrow proc(c, Q)

5

c: \text{int} \land A

proc(a, P) \rightarrow M \leftarrow M \rightarrow proc(c, Q)

5: \text{int}
Simple Monitor

\[
\text{proc}(a, P) \rightarrow M \leftarrow M \rightarrow \text{proc}(c, Q)
\]

\[
\text{proc}(a, P) \rightarrow M \leftarrow M \rightarrow \text{proc}(c, Q)
\]

\[
\text{proc}(a, P) \rightarrow M \leftarrow \text{proc}(c, Q)
\]

\[
\text{proc}(a, P) \rightarrow M \leftarrow \text{proc}(c, Q)
\]
Higher-Order Monitor

\[ \text{proc}(a, P) \rightarrow M \leftarrow M \rightarrow \text{proc}(c, Q) \]

\[ c: A \otimes B \]

\[ d \]
Higher-Order Monitor

\[ \text{proc}(a, P) \rightarrow M \leftarrow M \rightarrow \text{proc}(c, Q) \]

\[ \text{proc}(a, P) \rightarrow M \leftarrow d \rightarrow M \rightarrow \text{proc}(c, Q) \]

\[ \text{proc}(a, P) \rightarrow M \leftarrow M \rightarrow M \rightarrow \text{proc}(d, R) \]

\[ d: A \]

\[ c: B \]

\[ c: A \otimes B \]
Monitoring Challenges

• Havoc transitions can cause channels to be duplicated, dropped, etc
• This can create non-linear dependencies
Blame Example

Monitor Record:
- No spawns yet!

Cam

\[
\text{Cam} = \&\{ \text{take : photoPerm} \\
\rightarrow \text{picHandle} \otimes \text{Cam} \}
\]
Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

```latex
Cam = &\{\text{take : photoPerm} \\
\quad \rightarrow \text{picHandle} \otimes \text{Cam}\}
```
Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

- **picHandle** spawned **photoPerm**

\[
Cam = \&\{\text{take} : \text{photoPerm} \rightarrow \text{picHandle} \otimes \text{Cam}\}
\]
Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

- **picHandle** spawned **photoPerm**

\[
\text{Cam} = \&\{ \text{take} : \text{photoPerm} \\
\rightarrow \text{picHandle} \otimes \text{Cam} \}\]
Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

- **picHandle** spawned **photoPerm**

\[
\text{Cam} = \&\{ \text{take : photoPerm} \Rightarrow \text{picHandle} \otimes \text{Cam} \}
\]
Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

- **picHandle** spawned **photoPerm**

\[
Cam = \&\{\text{take : photoPerm} \\
\rightarrow \text{picHandle} \otimes Cam\}\]
Blame Example

Monitor Record:

- **Cam** spawned **picHandle**

- **picHandle** spawned **photoPerm**

\[ \text{Cam} = \&\{ \text{take} : \text{photoPerm} \rightarrow \text{picHandle} \otimes \text{Cam} \} \]
Big Spawn Tree
Havoced Spawn Tree
Havoced Spawn Tree
Blame Path
Theoretical Results

- Correctness of blame
- Well typed configurations do not raise alarms
- Monitor transparency
- Minimality*
Correctness of Blame

• In case of an alarm, one of the indicated set of possible culprits must have been compromised.

Definition 1 (Correctness of blame). A set of processes $\mathcal{N}$ is correct to be blamed w.r.t. the execution trace $\mathcal{T} = \Omega, G \xrightarrow{*} \Omega'$, alarm$(a)$ with $\models \Omega : \text{wf}$ if there is a $b \in \mathcal{N}$ such that $b$ has made a havoc transition in $\mathcal{T}$. 
Well Typed Configurations

- A havoc transition is necessary for the monitor to halt execution and assign blame

**Definition 2** (Well-typed configurations do not raise alarms).

Given any $\mathcal{T} = \Omega, G \xrightarrow{*} \Omega', G'$ such that $\models \Omega : \text{wf}$ and $\mathcal{T}$ does not contain any havoc transitions, there does not exist an $a$ such that $\text{alarm}(a) \in \Omega'$. 
Monitor Transparency

- Dynamic monitoring does not change system behavior for well-typed processes

**Definition 3 (Monitor transparency).** Given any \( T = \Omega, G \xrightarrow{*} \Omega', G' \) such that \( \models \Omega : \text{wf} \) and \( T \) does not contain any havoc transitions. Then \( \Omega(\longrightarrow^-)^*\Omega'' \), where \( \Omega'' \) is obtained from \( \Omega' \) by removing typing information from queues.
Minimality*

- The set of processes is as minimal as possible with respect to the observational model of the monitor
- This is a conjecture
Technical Challenges

• Execution may continue for havoced processes for many steps before an observable type violation occurs

• Rogue process configurations may violate invariants such as linearity
Summary

- System of monitoring and blame assignment for session-type asynchronous communication model
- Adversary model allows process to transition to ill typed code in a havoc step

Tech Report: https://www.cylab.cmu.edu/research/techreports/2015/tr_cylab15004.html
Related Work


- Multiparty Session Types: Bocchi et al. (2013), Chen at al. (2011), Thiemann (2014)
Future Work

• Dependent types
• Computational contracts
• More expressive security properties
Questions?