Monitors and Blame Assignment for Higher-Order Session Types

HANNAH GOMMERSTADT

LIMIN JIA & FRANK PFENNING
Anna Problems
Good Communication
Bad Communication
What happened?
Or...
Contributions

• Use **session types** to **dynamically monitor** communication between processes to detect undesirable behavior

• Correctly **blame** the party that violated the communication contract
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. \( \text{proc}(c, P) \)
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

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

• Takes control of a process by replacing it by another

\textcolor{red}{\textbf{havoc}}: \textit{proc}(c, P) \rightarrow \textit{proc}(c, Q)
Attacker Model

• Takes control of a process by replacing it by another

\[ \text{havoc: } \text{proc}(c, P) \mapsto \text{proc}(c, Q) \]

• \( Q \) cannot invent new channels, must have knowledge of existing ones
Communication Contracts

Session types express communication contracts between concurrent processes.
Related Work

• Session types as a model for concurrent communicating processes (Honda 1998)
• Blame theorems for higher-order contracts (Findler & Felleisen 2002)
Session Types

```
proc(a, P) \rightarrow proc(c, send c 5; Q)
```

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

5
Session Types

proc(a, P) \rightarrow \begin{array}{c}
\text{5}
\end{array} \quad proc(c, \text{send } c \ 5; Q) \\
\quad \text{5}
\quad c: \text{int } \land A

proc(a, P) \rightarrow \begin{array}{c}
\text{5}
\end{array} \quad proc(c, Q) \\
\quad c: A
Linearity

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

where \( A \) is a session type

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

Cam = take:

Diagram showing a user interacting with a camera, achieving the action of taking a picture.
Example

Cam = &{take :}
Example

\[ \text{Cam} = \&\{ \text{take : photoPerm} \} \]
Example

Cam = \{ \text{take : photoPerm } \rightarrow \text{picHandle} \}
Example

Cam = &{take : photoPerm → picHandle ⊗ Cam}
Example

User = picRequest:
Example

User = &\{picRequest :
Example

\[ \text{User} = \&\{ \text{picRequest} : \]

\[ \begin{align*}
\text{fail} : \\
\text{success} : \\
\end{align*} \]

\[ \} \]
Example

\[
\text{User} = \&\{ \text{picRequest} : \oplus \{ \text{fail} : \text{success} : \} \}
\]
Example

User = \&\{picRequest:
  \oplus \{fail : User;
    success :
  \}
}
Example

```latex
User = \&\{picRequest :
\oplus \{ fail : User;
   success : photoPerm \}\}
```
Example

\[
\text{User} = \&\{ \text{picRequest} : \\
\quad \oplus \{ \text{fail} : \text{User}; \\
\quad \quad \text{success} : \text{photoPerm} \otimes \text{User} \} \}
\]
Example

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

\[\text{User} = \&\{\text{picRequest} : \oplus \{\text{fail} : \text{User}; \text{success} : \text{photoPerm} \otimes \text{User}\}\}\]
# 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>$c: A \circ B$</td>
<td>Receive channel $d: A$ along $c$, continue $B$</td>
</tr>
<tr>
<td>$c: \oplus {l_i: A_i}$</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>
Monitor Capabilities

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

- All processes are untrusted
- All monitors are trusted
- All message queues are trusted
Simple Monitor

\[ \text{proc}(a, P) \rightarrow M \rightarrow c: \text{int} \land A \rightarrow \text{proc}(c, \text{send c 5; Q}) \]
Simple Monitor

\[
\text{proc}(a, P) \xrightarrow{\text{M}} \text{proc}(c, \text{send } c \ 5; Q)
\]

\[
\text{proc}(a, P) \xrightarrow{\text{M}} \text{proc}(c, \text{send } c \ 5; Q)
\]

\[
\text{proc}(a, P) \xrightarrow{\text{M}} \text{proc}(c, \text{send } c \ 5; Q)
\]
Simple Monitor

\[
\text{proc}(a, P) \xrightarrow{c: \text{int} \land A} \text{proc}(c, \text{send } c \ 5; Q) \\
\text{proc}(a, P) \xrightarrow{c: \text{int} \land A} \text{proc}(c, \text{send } c \ 5; Q) \\
\text{proc}(a, P) \xrightarrow{5: \text{int}} \text{proc}(c, \text{send } c \ 5; Q) \\
\text{proc}(a, P) \xrightarrow{\text{proc}(c, Q)}
\]
Simple Monitor

\[
\text{proc}(a, P) \rightarrow M \rightarrow M \rightarrow \text{proc}(c, \text{send } c \ 5; Q) \quad \text{c: int } \land A
\]

“sloth”
Simple Monitor

\[ \text{proc}(a, P) \quad \text{M} \quad \text{M} \quad \text{proc}(c, \text{send } c \ 5; Q) \]

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

\[ \text{proc}(a, P) \quad \text{M} \quad \text{M} \quad \text{proc}(c, \text{send } c \ 5; Q) \]

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

\[ \text{“sloth”}: \text{int} \]

\[ \text{“sloth”}: \text{int} \]

\[ \text{“sloth”}: \text{int} \]
Simple Monitor

```
proc(a, P)  M  M  c: int ∧ A  proc(c, send c 5; Q)  “sloth”
```

```
proc(a, P)  M  M  c: int ∧ A  proc(c, send c 5; Q)  “sloth”: int
```
Higher-Order Monitor - Cut

\[ \text{proc}(a, P) \quad M \quad \text{M} \quad c : B \quad \text{proc}(c, d \leftarrow R; Q) \]

d : A
Higher-Order Monitor - Cut

\[
\text{proc}(a, P) \xrightarrow{M} \text{proc}(c, d \leftarrow R; Q) \quad c : B
\]

\[
\text{proc}(a, P) \xrightarrow{M} \text{proc}(c, Q) \quad c : B
\]

\[
\text{proc}(a, P) \xrightarrow{M} \text{proc}(d, R) \quad d : A
\]
Higher-Order Monitor - \(\otimes\)

\[
\text{proc}(a, P) \quad \text{M} \quad \text{M} \quad \text{proc}(d, R) \\
\text{proc}(c, \text{send } c d; Q) \quad \text{M} \quad \text{M} \\
\]

\(d: A\)  \(c: A \otimes B\)
Higher-Order Monitor - $\otimes$

$\mathsf{proc(a, P)}$ $\rightarrow$ $\mathsf{proc(c, send c d; Q)}$

$\mathsf{proc(a, P)}$ $\rightarrow$ $\mathsf{proc(c, Q)}$

$\mathsf{proc(a, P)}$ $\rightarrow$ $\mathsf{proc(d, R)}$
Blame Challenges

• Passing channels along channels complicates the communication
• If an alarm is raised during higher-order communication, it is not clear which processes are to be blamed
Non-higher Order Case

Process configuration
Non-higher Order Case

Process configuration
Non-higher Order Case

Process configuration
Higher-Order Case

Process configuration
Higher-Order Case

Process configuration

Communication configuration
Higher-Order Case

Process configuration

Communication configuration
Incorrect Blame

Process configuration

Communication configuration
Correct Blame

Process configuration

Communication configuration
Intuition

Process configuration

Communication configuration
Intuition

Process configuration

Communication configuration
Incorrect Blame

Process configuration

Communication configuration
Correct Blame

Process configuration

Communication configuration

Root
User
Perm
Pic
Cam
Perm
Pic
User
Cam
Root
Blame Summary

Regardless of communication configuration, our higher-order monitors are able to produce a correct blame set by using the process configuration.
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 $\mathcal{T} = \Omega, G \rightarrow^*$ $\Omega'$, 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 $T = \Omega, G \rightarrow^* \Omega', G'$ such that $\models \Omega : \text{wf}$ and $T$ does not contain any havoc transitions, there does not exists 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 $\mathcal{T} = \Omega, G \xrightarrow{*} \Omega', G'$ such that $\models \Omega : \text{wf}$ and $\mathcal{T}$ does not contain any havoc transitions. Then $\Omega(\xrightarrow{-})^*\Omega''$, 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
Blame Proof Sketch

- (Giant messy) Preservation proof over the monitoring rules (operational semantics augmented with monitor actions)
- Track havoc processes in a graph to later blame if alarm is raised
Blame Proof Challenges

- Havoc transitions can violate the linearity condition
- Can now have channels being dropped or duplicated which complicates the communication even further
Minimality Revisited

• The set of processes is as minimal as possible with respect to the observational model of the monitor
Can we do better?

• How much more power do the monitors have to have to make our blame assignment more precise?
New monitor capabilities

- The monitor verifies that any new spawned process has the appropriate type
- Processes are no longer black boxes
Higher-Order Monitor - Cut

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

\( c : B \)

\( d : A \)
Higher-Order Monitor - Cut

proc(a, P)

\[ \text{M} \quad \text{M} \quad \text{c: B} \quad \text{proc(c, d ← R; Q)} \]

\[ \text{d : A} \]

proc(a, P)

\[ \text{M} \quad \text{M} \quad \text{c: B} \quad \text{proc(c, Q)} \]

\[ \text{d : A} \quad \text{proc(d, R)} \]

R :: (d : A)
Example Take 2

Process configuration

Communication configuration
Take 2

Process configuration

Communication configuration
Incorrect Blame

Process configuration

Communication configuration
Correct Blame

Process configuration

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

**Definition 1** (Correctness of blame). A set $N$ is correct to be blamed w.r.t. the execution trace $T = \Gamma, \Omega, H \rightarrow^* \text{alarm}(a)$ with $\models \Omega : \text{wf}$ if $N$ contains a single process that has made a havoc transition in $T$. 
Blame Proof Sketch

• (Not messy) Preservation proof over the monitoring rules (operational semantics augmented with monitor actions)

• Track types of channels at each step of the communication to typecheck new spawned processes
## Monitoring Tradeoffs

<table>
<thead>
<tr>
<th>Monitor Capabilities</th>
<th>Maximum Size of the Blame Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Monitors can’t access process internals</td>
<td>The longest path from the root of the process configuration tree to a leaf</td>
</tr>
<tr>
<td>- Communications over channels are monitored</td>
<td></td>
</tr>
<tr>
<td>- Every new spawned process is typechecked</td>
<td>One process</td>
</tr>
<tr>
<td>- Communications over channels are monitored</td>
<td></td>
</tr>
</tbody>
</table>
Takeaways

• Using session types as communication contracts enables dynamic monitoring
• The monitors assign blame correctly which is a significant theoretical result
• There are tradeoffs between monitor capability and blame precision
Future Work

• Communication contracts
• More expressive security properties via dependent types
Questions?

User

Camera

Android OS

perm

picRequest

take
Related Work


• Multiparty Session Types: Bocchi et al. (2013), Chen et al. (2011), Thiemann (2014)