Summary

● Overview
  - issues in secure service composition
  - safety framings and policies
  - req-by-contract for service request
  - plans for secure orchestration

● A calculus for service composition
  - syntax and operational semantics
  - type & effect system

● Plans & Orchestration
  - constructions of plans and linearization
  - model-checking viable plans
Programming in a world of services

\[ \ell_1 \quad \tau_1 \]

S1

\[ \ell_2 \quad \tau_2 \]

S2

\[ \ell_3 \quad \tau_3 \]

S3

\[ \ell_4 \quad \tau_4 \]

S4

client
Programming in a world of services

client

S1

S2

S3

S4

service location

service code

service interface

service orchestration
Security and service composition

- Two kinds of security concerns:
  - Secrecy of transmitted data, authentication, etc (protocol analysis techniques)
  - Control on computational resources (access control, resource usage analysis, information flow control, etc)

- Need for linguistic mechanisms that:
  - Work in a distributed setting
  - Assume no trust relations among services
  - Can also cope with mobile code
Security and service composition: safety framings

Client wants to protect from untrusted results

Linguistic mechanism: safety framing

The policy $\phi$ is enforced stepwise within its scope
Security and service composition: safety framings

Similarly, services want to protect from clients

(client) \(\rightarrow\) remote exec \(\rightarrow\) service

(result)

(now the safety framing belongs to the service)

(client) \(\rightarrow\) remote exec \(\rightarrow\) \(\varphi'[exec]\)

(result)

Scoped policies check the local execution histories
Security and service composition: service selection

*Req-by-name*: request a *given* service among many

Why **S2** and not **S1** or **S3**, if all functionally equivalent?
Security and service composition: service selection

Problems with “request by name”:

- what if named service $S_2$ becomes unavailable?
- …and if $S_2$ is outperformed by $S_1$ or $S_3$?
- hard reasoning about non-functional properties of services (e.g. security)
- security level independent of the execution context (unless hard-wired in the service code)

From syntax-based to semantics-based invocation

Service names $e,e',..$ tell me nothing about the behaviour!
Security and service composition: service selection

Req-by-contract: request a service respecting the desired behaviour

τ imposes both functional and non-functional constraints
Use cases for Req-by-contract

**Example**: download an applet that obeys the policy $\varphi$

\[ \text{req} \quad \tau_0 \longrightarrow ( \tau_1 \xrightarrow{\varphi[:]} \tau_2 ) \]

**Example**: a remote executer that obeys the policy $\varphi'$

\[ \text{req} \ ( \tau_0 \longrightarrow \tau_1 ) \xrightarrow{\varphi'[:]} \tau_2 \]
Observable behaviour

- **access events** are the actions relevant for security (e.g. read/write local files, invoke/be invoked by a given service, etc)
  - mechanically inferred, or inserted by programmer.
  - their meaning is fixed globally.
  - access events cannot be hidden.

- the (abstract) behaviour observable by the orchestrator over-approximates the run-time histories, i.e. sequences of access events (via a type & effect system).
What kind of policies?

- History-based security
- Policies $\phi$ are regular properties event histories
- Policies $\phi, \phi'$ have a local scope, possibly nested $\phi[\cdots \phi'[\cdots] \cdots]$
- A policy can only control histories of a single site (no trust among services)
- Histories are local to stateless service sites (stateful easy)
Example: the Chinese Wall policy

\( \varphi \) Chinese Wall: cannot write (\( \alpha_w \)) after read (\( \alpha_r \))
Principle of Least Privilege

“Programs should be granted the minimum set of rights needed to accomplish their task”

- A service must always obey all the active policies (no policy override)
- Policies can always inspect the whole past history (no event can be discarded)
- “Privileged calls” implemented by policies that explicitly discard the past
1. infer $\tau_3$ from $S3$
2. mark $S3$ so to prevent spoofing
Service publication (2)

Service Repository

\[ \ell_1 \{S1\}: \tau_1 \]
\[ \ell_2 \{S2\}: \tau_2 \]
\[ \ell_3 \{S3\}: \tau_3 \]

\[ \ell_1 \quad \ell_2 \quad \ell_3 \]

\[ \tau_1 \quad \tau_2 \quad \tau_3 \]

\[ S1 \quad S2 \quad S3 \]
Service orchestration

1. combine $\tau$ with the $\tau_i$ to infer the abstract behaviour $H$
2. extract from $H$ a viable plan $\pi$ for $\ell_0$
Service orchestration

Plan: $\pi = r[\ell_2]$

Names are only known by the orchestrator!
What is a plan?

- A plan drives the execution of an application, by associating each service request with one (or more) appropriate services.
- With a **viable plan**:
  - executions never violate policies
  - there are no unresolved requests
  - you can then *dispose* from any execution monitoring!
- Many kinds of plans:
  - **Simple**: one service for each request
  - **Multi-choice**: more services for each request
  - **Dependent**: one service, and a continuation plan
  - ...
Who do we trust?

The orchestrator, that:

- certifies the behavioural descriptions of services *(types annotated with effects $H$)*
- composes the descriptions, and ensures that selected services match the requested types
- extracts the *viable plans* (through model-checking)

Also, someone must ensure that services do not change their code on-the-fly
Summing up...

- a calculus for secure service composition:
  - distributed services
  - safety framings scoped policies on localized execution histories
  - req-by-contract service invocation

- static orchestrator:
  - certifies the behavioural interfaces of services
  - provides a client with the viable plans driving secure executions
What’s next

- calculus: syntax and operational semantics
- static semantics: type & effect system
  - types carry annotations $H$ about service behaviour
  - effects $H$ are history expressions, which over-approximate the actual execution histories
- extracting viable plans:
  - linearization: unscrambling the structure of $H$
  - model checking: valid plans are viable
Services $e ::= x$

$\alpha$

if $b$ then $e$ else $e'$

$\lambda z x. e$

$e \ e'$

$\phi[e]$

$\text{req}_r \ \tau$

$\text{wait} \ \ell$

(only in configs)

variable

access event

conditional

abstraction

application

safety framing

service request

wait reply
Networks

\[ N ::= \ell\{e: \tau\}: \eta, e' \]

- location
- service code and published interface
- published service composition
- running code
- execution history
A plan is a function from requests \( r \) to services \( \ell \)

\[
\pi ::= 0 \quad \text{empty} \\
r[\ell] \quad \text{service choice} \\
\pi \mid \pi' \quad \text{composition}
\]

Plans respect the partial knowledge \( \ell < \ell' \) of services about the network (\(<\) is a partial ordering)
Example: delegating code execution

$$\ell_1 \quad \lambda x. \varphi[\alpha_r;\ldots]$$

$$\ell_2 \quad \alpha_c; (\lambda x. \alpha_r;\ldots;\alpha_w)$$

$$\ell_3 \quad f = \text{req}_{r1}()$$

$$\ell_4 \quad \alpha_c; \varphi'[f()]$$

$$\text{req}_{r2}(f)$$

$$f()$$
Example: delegating code execution

\[ \lambda x. \phi[\alpha_r; \cdots] \]

\[ f = \text{req}_{r_1}() \]

\[ \alpha_c; \phi'[f()] \]

\[ \alpha_c; \text{req}_{r_2}(f) \]

\[ \alpha_w; \alpha_r \]

use in certified sites \( \alpha_c \) only

do not write \( \alpha_w \) after a read \( \alpha_r \)

f()
Executing a network of services

\[ \pi = r_1[l_2] \mid r_2[l_3] \]

\[ \ell_1 \]
\[ \lambda x. \varphi[\alpha_r; \ldots] \]

\[ \ell_2 \]
\[ \alpha_c; (\lambda x. \alpha_r; \ldots; \alpha_w) \]

\[ \ell_0 \]
\[ f = \text{req}_{r_1}() \]
\[ \text{req}_{r_2}(f) \]

\[ \ell_3 \]
\[ \alpha_c; \varphi'[f()] \]

\[ \ell_4 \]
\[ \ldots f() \ldots \]
Executing a network of services

\[ \ell_1 \]

\[ \lambda x. \varphi[\alpha_r; \cdots] \]

\[ \ell_2 \quad \varepsilon \]

\[ \alpha_c; (\lambda x. \alpha_r; \cdots; \alpha_w) \]

\[ \ell_0 \]

\[ f = \text{wait} \ \ell_1 \]

\[ \text{req}_{r_2}(f) \]

\[ \ell_3 \]

\[ \alpha_c; \varphi'[f()] \]

\[ \ell_4 \]

\[ \ldots f() \ldots \]

\[ \pi = r_1[\ell_2] \mid r_2[\ell_3] \]
Executing a network of services

\[ \pi = r_1[\ell_2] \mid r_2[\ell_3] \]

\[ f = \text{wait } \ell_1 \]

\[ \text{req}_{r_2}(f) \]

\[ \lambda x. \varphi[\alpha_r; \ldots] \]

\[ \lambda x. \alpha_r; \ldots; \alpha_w \]

\[ \alpha_c; \varphi'[f()] \]

\[ \ldots f() \ldots \]
Executing a network of services

\[ \pi = r_1[l_2] \mid r_2[e_3] \]
Executing a network of services

\[ \pi = r_1[l_2] \mid r_2[e_3] \]
Executing a network of services

\[ \pi = r_1[l_2] \mid r_2[e_3] \]
Executing a network of services

\[ \pi = r_1[l_2] \mid r_2[e_3] \]

not viable!
Semantics of services (1)

- **[App1]**
  \[
  \eta, e_1 \rightarrow \eta', e_1'
  \]
  \[
  \eta, e_1 e_2 \rightarrow \eta', e_1'e_2
  \]

- **[App2]**
  \[
  \eta, e_2 \rightarrow \eta', e_2'
  \]
  \[
  \eta, v e_2 \rightarrow \eta', v e_2'
  \]

- **[AbsApp]**
  \[
  \eta, (\lambda_x e) v \rightarrow \eta, e[v/x, \lambda_x e/z]
  \]

- **[If]**
  \[
  \eta, \text{if } b \text{ then } e_{\text{true}} \text{ else } e_{\text{false}} \rightarrow \eta, e^{B(b)}
  \]
Semantics of services (2)

**[Event]**
\[ \eta, \alpha \rightarrow \eta\alpha, () \]

**[Framing In]**
\[ \eta, e \rightarrow \eta', e' \quad \eta' \models \varphi \]
\[ \eta, \varphi[e] \rightarrow \eta', \varphi[e'] \]

**[Framing Out]**
\[ \eta \models \varphi \]
\[ \eta, \varphi[v] \rightarrow \eta, v \]
Semantics of networks (1)

[Inject]

\[ \eta, e \rightarrow \eta', e' \]

\[ \ell: \eta, e \rightarrow_\pi \ell: \eta', e' \]

[Par]

\[ N_1 \rightarrow_\pi N_1' \]

\[ N_1 \parallel N_2 \rightarrow_\pi N_1' \parallel N_2 \]

\{e:\tau\} omitted
Semantics of networks (2)

\[ \pi = r[\ell'] | \pi' \text{ -- plan} \]

\[
\ell: \eta, \text{req}, v \parallel \ell'[e'] : \varepsilon, \star \quad \rightarrow_\pi \\
\ell: \eta, \text{wait} \ell' \parallel \ell'[e'] : \varepsilon, e'v
\]

\[ \ell: \eta, \text{wait} \ell' \parallel \ell'[e'] : \eta', v \quad \rightarrow_\pi \\
\ell: \eta, v \quad \parallel \ell'[e'] : \varepsilon, \star \]
Other kinds of plans

- **Simple plans**  \( \pi ::= 0 \mid \pi \mid \pi \mid r[\ell] \)
  
  \( \ell: \text{req}_r \parallel \ell': \{P\} \rightarrow_r[\ell'] \ell: \text{wait} \ell' \parallel \ell': P \)

- **Multi-choice plans**  \( \pi ::= 0 \mid \pi \mid \pi \mid r[\ell_1 \ldots \ell_k] \)
  
  \( \ell: \text{req}_r \parallel \ell': \{P\} \rightarrow_{r[\ell',\ell'']} \ell: \text{wait} \ell' \parallel \ell': P \)

- **Dependent plans**  \( \pi ::= 0 \mid \pi \mid \pi \mid r[\ell. \pi] \)
  
  \( \ell: r[\ell'. \pi] \triangleright \text{req}_r \parallel \ell': \{P\} \rightarrow \ell: r[\ell'. \pi] \triangleright \text{wait} \ell' \parallel \ell': \pi \triangleright P \)

- **...many others: multi-dependent, regular, dynamic,...**
Static semantics

Type & effect system

- **types** carry annotations \( H \) about service abstract behaviour
- **effects** \( H \), namely *history expressions*, over-approximate the actual execution histories
- the type & effect inferred for a service depends on its *partial knowledge* of the network
Types

(pretty standard)

\[ \tau ::= \text{int} \mid \text{bool} \mid 1 \mid \ldots \mid \tau \xrightarrow{H} \tau' \]
Effects (history expressions)

\[ H ::= \begin{align*}
\varepsilon & \quad \text{empty} \\
\alpha & \quad \text{access event} \\
H \cdot H' & \quad \text{sequence} \\
H + H' & \quad \text{choice} \\
h & \quad \text{variable} \\
\mu h.H & \quad \text{recursion} \\
\phi[H] & \quad \text{safety framing} \\
\ell : H & \quad \text{localization} \\
\{\pi_1 \triangleright H_1, \ldots, \pi_k \triangleright H_k\} & \quad \text{planned selection}
\end{align*} \]
Semantics of history expressions

\[
[[\alpha]]^\pi = (?:\alpha)
\]

\[
[[\ell: H]]^\pi = [[H]]^\pi \{\ell/\}.
\]

\[
[[\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}]]^\pi = \bigcup_{i=1..k}[[\{\pi_i \triangleright H_i\}]]^\pi
\]

\[
[[\{\pi' \triangleright H\}]]^\pi = [[H]]^\pi \quad \text{if} \quad \pi' \leq \pi \quad \text{plan} \ \pi' \text{ resolves the requests as} \ \pi
\]

\[
0 \leq \pi \quad r[\ell] \leq r[\ell] \mid \pi \quad \pi_0 \mid \pi_1 \leq \pi \quad \text{if} \quad \pi_0 \leq \pi \& \pi_1 \leq \pi
\]
Semantics of history expressions

\[
[[ H \cdot H' ]]^\pi = [[ H ]]^\pi \cdot [[ H' ]]^\pi
\]

\[
[[ H + H' ]]^\pi = [[ H ]]^\pi + [[ H' ]]^\pi
\]

\[
[[ \mu h.H]]^\pi = \bigcup_{n>0} f^n(\bot)
\]

where \( f(X) = [[ H ]]^\pi \{ X / h \} \)
Example

\[ H = \{ r[e] \triangleright \{ r'[e_1] \triangleright \alpha_1, r'[e_2] \triangleright \alpha_2 \}, \]
\[ r[e'] \triangleright \beta \} \]

\[ \pi = r[e] \mid r'[e_2] \]

\[
[[ H ]]^\pi = \left[ \left[ \left[ r[e] \triangleright \{ r'[e_1] \triangleright \alpha_1, r'[e_2] \triangleright \alpha_2 \} \right] \right] \right]^\pi \cup \left[ \left[ r[e'] \triangleright \beta \right] \right]^\pi \\
= [[ \left[ r'[e_1] \triangleright \alpha_1, r'[e_2] \triangleright \alpha_2 \right] ]]^\pi \\
= [[ \left[ r'[e_1] \triangleright \alpha_1 \right] ]]^\pi \cup [[ \left[ r'[e_2] \triangleright \alpha_2 \right] ]]^\pi \\
= [[ \alpha_2 ]]^\pi = (?: \alpha_2)
Example

\[ H = e: \{ r[\ell_1] \triangleright \ell_1: \alpha_1, \ r[\ell_2] \triangleright \ell_2: \alpha_2 \} \cdot \beta \]

\[ \pi = r[\ell_1] \]

\[ \llbracket H \rrbracket^\pi = \llbracket \{ r[\ell_1] \triangleright \ell_1: \alpha_1, \ r[\ell_2] \triangleright \ell_2: \alpha_2 \} \cdot \beta \rrbracket^\pi \{ \ell/? \} \]

\[ = \llbracket \{ r[\ell_1] \triangleright \ell_1: \alpha_1, \ r[\ell_2] \triangleright \ell_2: \alpha_2 \} \rrbracket^\pi \cdot (\ell: \beta) \]

\[ = \llbracket \ell_1: \alpha_1 \rrbracket^\pi \cdot (\ell: \beta) \]

\[ = (?: \alpha_1) \{ \ell_1/? \} \cdot (\ell: \beta) \]

\[ = (\ell: \beta, \ell_1: \alpha_1) \]
Typing rules (1)

[T-Ev]
\[ \Gamma, \alpha |-_\ell \alpha : 1 \]

[T-Loc]
\[ \Gamma, H |-_\ell e : \tau \]
\[ \frac{\Gamma, \ell : H |- e : \tau}{\Gamma, \ell : H |- e : \tau} \]

[T-Var]
\[ \Gamma, \varepsilon |-_\ell x : \Gamma(x) \]

[T-Wk]
\[ \Gamma, H |-_\ell e : \tau \]
\[ \frac{\Gamma, H \cdot H' |-_\ell e : \tau}{\Gamma, H + H' |-_\ell e : \tau} \]}
Typing rules (2)

[T-If]

\[
\Gamma, H \vdash_{\ell} e : \tau \quad \Gamma, H \vdash_{\ell} e' : \tau \\
\Gamma, H \vdash_{\ell} \text{if } b \text{ then } e \text{ else } e' : \tau
\]

[T-Fr]

\[
\Gamma, H \vdash_{\ell} e : \tau \\
\Gamma, \varphi[H] \vdash_{\ell} \varphi[e] : \tau
\]
Typing rules (3)

[T-Abs]

\[
\frac{\Gamma; x: \tau; z: \tau \quad H \rightarrow \tau', H \vdash e : \tau'}{
\Gamma, \varepsilon \vdash \lambda z x. e : \tau \quad H \rightarrow \tau'}
\]

[T-App]

\[
\frac{\Gamma, H \vdash e : \tau \quad H'' \rightarrow \tau' \quad \Gamma, H' \vdash e' : \tau}{
\Gamma, H \cdot H' \cdot H'' \vdash e e' : \tau'}
\]
Typing Example (1)

\[ \alpha \vdash \epsilon \alpha : 1 \]
\[ \vdash \epsilon (\lambda y. zx) \beta : 1 \]
\[ z : 1 \xrightarrow{H} 1, \alpha + \epsilon \text{ if } b \text{ then } \alpha \text{ else } (\lambda y. zx) \beta : 1 \]
Typing Example (2)

\[ \varepsilon \mid_{\varepsilon} (\lambda y. z x) : 1 \xrightarrow{H} 1 \]

\[ \beta \mid_{\varepsilon} \beta : 1 \]

\[ \alpha \mid_{\varepsilon} \alpha : 1 \]

\[ \varepsilon \cdot \beta \cdot H \mid_{\varepsilon} (\lambda y. z x) \beta : 1 \]

\[ z : 1 \xrightarrow{H} 1, \alpha + \beta \cdot H \mid_{\varepsilon} if \ b \ then \ \alpha \ else \ (\lambda y. z x) \beta : 1 \]
Typing Example (3)

\[
\begin{array}{c}
x:1; z:1 \xrightarrow{H} 1, \text{H} \mid_{\varepsilon} zx:1 \\
\hline
\varepsilon \mid_{\varepsilon} (\lambda y. zx):1 \xrightarrow{H} 1 \\
\alpha \mid_{\varepsilon} \alpha:1 \\
\beta \cdot \text{H} \mid_{\varepsilon} (\lambda y. zx) \beta:1 \\
z:1 \xrightarrow{H} 1, \alpha + \beta \cdot \text{H} \mid_{\varepsilon} \text{if } b \text{ then } \alpha \text{ else } (\lambda y. zx) \beta:1
\end{array}
\]
Typing Example (4)

\[
\begin{align*}
z:1 & \xrightarrow{\text{H}} 1, \varepsilon \vdash_{\varepsilon} z:1 \xrightarrow{\text{H}} 1 & x:1, \varepsilon \vdash_{\varepsilon} x:1 \\
\hline
x:1;z:1 & \xrightarrow{\text{H}} 1, \varepsilon \cdot \varepsilon \cdot \text{H} \vdash_{\varepsilon} zx:1 \\
\hline
\varepsilon \vdash_{\varepsilon} (\lambda y.zx):1 & \xrightarrow{\text{H}} 1 & \beta \vdash_{\varepsilon} \beta:1 \\
\hline
\alpha \vdash_{\varepsilon} \alpha:1 & \beta \cdot \text{H} \vdash_{\varepsilon} (\lambda y.zx)\beta:1 \\
\hline
z:1 & \xrightarrow{\text{H}} 1, \alpha+ \beta \cdot \text{H} \vdash_{\varepsilon} \text{if } b \text{ then } \alpha \text{ else } (\lambda y.zx)\beta:1
\end{align*}
\]
Typing Example

\[ z:1 \xrightarrow{H} 1, \alpha + \beta \cdot H \mid_- \varepsilon \text{ if } b \text{ then } \alpha \text{ else } (\lambda y.zx)\beta : 1 \]

\[ \varepsilon \mid_- \varepsilon \lambda z x. \text{ if } b \text{ then } \alpha \text{ else } (\lambda y.zx)\beta : \tau \xrightarrow{H} \tau' \]

To use rule [T-Abs] the latent and actual effects must be unified, i.e. \( H = \alpha + \beta \cdot H \)

A history expression that satisfies the above is:

\( H = \mu h. \alpha + \beta \cdot h \)
Typing rules (3)

\[
\begin{align*}
\tau &= U \{ \rho +_{r[\ell]} \tau' \mid A \& B \& C \} \\
A &\equiv \emptyset, \varepsilon \mid -_{\ell'} \text{ e : } \tau' \\
B &\equiv \rho \approx \tau' \\
C &\equiv \ell < \ell' \{ \text{ e : } \tau' \}
\end{align*}
\]

\[\Gamma, \varepsilon \mid -_{\ell} \text{ req } \rho : \tau\]
Typing rules (3)

\[
\begin{align*}
\tau &= \bigcup \{ \rho + r_{[\ell]} \tau' \mid A \& B \& C \} \\
A &\equiv \emptyset, \varepsilon | -\varepsilon \ e : \tau' \\
B &\equiv \rho \approx \tau' \\
C &\equiv \ell < \ell' \ \{ e : \tau' \} \\
\Gamma, \varepsilon | -\varepsilon \ req_r \rho : \tau
\end{align*}
\]
Typing rules (3)

\[ \tau = \bigcup \{ \rho +_{r[\ell]} \tau' \mid A \& B \& C \} \]

\[ A \equiv \emptyset, \varepsilon \mid -\ell' \ e : \tau' \quad B \equiv \rho \approx \tau' \]

\[ C \equiv \ell < \ell' \ \{ e : \tau' \} \]

\[ \Gamma, \varepsilon \mid -\ell \ \text{req}_\tau \rho : \tau \]
Typing rules (3)

\[ [T-Req] \]

\[
\begin{align*}
\tau &= \bigcup \{ \rho +_{r[\ell]} \tau' \mid A \& B \& C \} \\
A &\equiv \emptyset, \varepsilon \mid \ell' \quad e : \tau' \\
B &\equiv \rho \approx \tau' \\
C &\equiv \ell < \ell' \{ e : \tau' \} \\
\Gamma, \varepsilon \mid \ell \text{ req}_r \rho : \tau
\end{align*}
\]
Certified published interfaces

\[ \ell_1: 1 \rightarrow (1 \xrightarrow{\varphi[\alpha_r]} 1) \]

\[ \lambda x. \varphi[\alpha_r; \ldots] \]

\[ f = \text{req}_{r1}() \]

\[ \alpha_c; \lambda x. \alpha_r; \ldots; \alpha_w \]

\[ \text{req}_{r2}(f) \]

\[ \ell_2: 1 \xrightarrow{\alpha_c} (1 \xrightarrow{\alpha_r \cdot \alpha_w} 1) \]

\[ \ell_3: (1 \xrightarrow{h} 1) \xrightarrow{\alpha_c \cdot \varphi'[h]} 1 \]

\[ \alpha_c; \varphi'[f()] \]

\[ \ell_4: (1 \xrightarrow{h} 1) \xrightarrow{h} 1 \]

\[ f() \]
Abstracting client behaviour

\[
\begin{align*}
\ell_1 & \quad 1 \xrightarrow{\varepsilon} (1 \xrightarrow{\phi[\alpha_r]} 1) \\
\lambda x.\phi[\alpha_r;\ldots] \\
\ell_2 & \quad 1 \xrightarrow{\alpha_c} (1 \xrightarrow{\alpha_r \cdot \alpha_w} 1) \\
\alpha_c; \lambda x.\alpha_r;\ldots;\alpha_w \\
f & = \text{req}_{r_1}() \\
\text{req}_{r_2}(f) \\
\ell_3 & \quad (1 \xrightarrow{h} 1 \xrightarrow{\alpha_c \cdot \phi'[h]} 1) \\
\alpha_c; \phi'[f()] \\
\ell_4 & \quad (1 \xrightarrow{h} 1 \xrightarrow{h} 1) \\
f()
\end{align*}
\]

\{ r_1[\ell_1] \triangleright \ell_1: \varepsilon, \ r_1[\ell_2] \triangleright \ell_2: \alpha_c \} .
Abstracting client behaviour

\[ f = \text{req}_{r1}() \]

\[ \alpha_c; \varphi'[f()] \]

\[ \{ r_2[e_3] \triangleright e_3 : \alpha_c \cdot \varphi'[\{ r_1[e_1] \triangleright \varphi[\alpha_r], r_1[e_2] \triangleright \alpha_r \cdot \alpha_w \}] \}, \]

\[ r_2[e_4] \triangleright e_4 : \{ r_1[e_1] \triangleright \varphi[\alpha_r], r_1[e_2] \triangleright \alpha_r \cdot \alpha_w \} \]
Summing up ...

Calculus: **operational semantics** and **type & effect system**

- **effects** are history expressions, and over-approximate the actual execution histories
- **planned selections** therein hinder information about which **plans** to choose for secure compositions
What’s next: the road to viable plans

- **linearization**: extracting plans and their “pure” effects by unscrambling the structure of history expressions
- **validity**: defining when an effect denotes histories that “never go wrong”
- **model checking**: valid plans are viable
  - transform **history expression** into BPAs
  - transform **policies** into FSAs
- **orchestrator**: uses viable plans to drive safe service composition
Which are the viable plans?

\[
\{ r_1[e_1] \triangleright e_1 : \varepsilon, r_1[e_2] \triangleright e_2 : \alpha_c \} \cdot \\
\{ r_2[e_3] \triangleright e_3 : \alpha_c \cdot \varphi'[\{ r_1[e_1] \triangleright \varphi[\alpha_r], r_1[e_2] \triangleright \alpha_r \cdot \alpha_w \}], \\
r_2[e_4] \triangleright e_4 : \{ r_1[e_1] \triangleright \varphi[\alpha_r], r_1[e_2] \triangleright \alpha_r \cdot \alpha_w \}\}
\]

Difficult to tell: the planned selections are nested!

\[
\{ r_1[e_1] | r_2[e_3] \triangleright e_1 : \varepsilon, e_3 : \alpha_c \cdot \varphi'[\varphi[\alpha_r]], \quad \text{viable} \\
\{ r_1[e_2] | r_2[e_4] \triangleright e_2 : \alpha_c, e_4 : \alpha_r \cdot \alpha_w, \quad \text{viable} \\
\{ r_1[e_1] | r_2[e_4] \triangleright e_1 : \varepsilon, e_4 : \varphi[\alpha_r], \quad \text{not viable} \\
\{ r_1[e_2] | r_2[e_3] \triangleright e_2 : \alpha_c, e_3 : \alpha_c \cdot \varphi'[\alpha_r \cdot \alpha_w] \}\}
\]

not viable
Linearization

- transform $H$ into a *semantically equivalent* $H' \equiv H$ such that $H'$ is in linear form, i.e.:
  $$H' = \{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$$
  and the $H_i$ have no planned selections.

- defined through oriented equations $\equiv$ that groups $r[\ell]$ in topmost position
Linearization

\[ H \equiv \{0 \triangleright H\} \]

\[ \{\pi_i \triangleright H_i\}_i \cdot \{\pi'_j \triangleright H'_j\}_j \equiv \{\pi_i \mid \pi'_j \triangleright H_i \cdot H'_j\}_{i,j} \]

\[ \{\pi_i \triangleright H_i\}_i + \{\pi'_j \triangleright H'_j\}_j \equiv \{\pi_i \mid \pi_j \triangleright H_i + H'_j\}_{i,j} \]

\[ \phi[\{\pi_i \triangleright H_i\}_i] \equiv \{\pi_i \triangleright \phi[H_i]\}_i \]

\[ \mu h. \{\pi_i \triangleright H_i\}_i \equiv \{\pi_i \triangleright \mu h. \ H_i \}_i \]

\[ \{\pi_i \triangleright \{\pi'_{i,j} \triangleright H_{i,j}\}_j\}_i \equiv \{\pi_i \mid \pi'_{i,j} \triangleright H_{i,j}\}_{i,j} \]
Example

\[ H = \varphi[ \mu h. \{ r[\varepsilon_1] \triangleright \alpha, r[\varepsilon_2] \triangleright \beta \} \cdot h ] \]

\[ H = \varphi[ \lambda z x. \text{req}_r \rho; z x ] \]
Example

\[ H = \varphi[ \mu h. \{ r[e_1] \triangleright \alpha, r[e_2] \triangleright \beta \} \cdot h ] \]

\[ \equiv \varphi[ \mu h. \{ r[e_1] \triangleright \alpha, r[e_2] \triangleright \beta \} \cdot \{0 \triangleright h\} ] \]

\[ \equiv \varphi[ \mu h. \{ r[e_1] \mid 0 \triangleright \alpha \cdot h, r[e_2] \mid 0 \triangleright \beta \cdot h\} ] \]

\[ \equiv \varphi[ \mu h. \{ r[e_1] \triangleright \alpha \cdot h, r[e_2] \triangleright \beta \cdot h\} ] \]

\[ \equiv \varphi[ \{ r[e_1] \triangleright \mu h. \alpha \cdot h, r[e_2] \triangleright \mu h. \beta \cdot h\} ] \]

\[ \equiv \{ r[e_1] \triangleright \varphi[ \mu h. \alpha \cdot h], r[e_2] \triangleright \varphi[ \mu h. \beta \cdot h]\} \]
Simple vs multi-choice plans

With simple plans:

\[ H \equiv \{ r[\ell_1] \triangleright \varphi[\mu h. \alpha \cdot h], r[\ell_2] \triangleright \varphi[\mu h. \beta \cdot h] \} \]

With multi-choice plans:

\[ H \equiv \{ r[\ell_1] \triangleright \varphi[\mu h. \alpha \cdot h], r[\ell_2] \triangleright \varphi[\mu h. \beta \cdot h], r[\ell_1, \ell_2] \triangleright \varphi[\mu h. (\alpha + \beta) \cdot h] \} \]

Plan \( r[\ell_1, \ell_2] \) useful when \( \ell_1 \) or \( \ell_2 \) unavailable
Example: bottleneck service

\[ \varphi = \text{never } \alpha \alpha \text{ nor } \beta \beta \]

\[ H = \varphi[ \{ r[e_1] \triangleright \{ r_1[e_2] \triangleright \alpha, r_1[e_3] \triangleright \beta \} \} \cdot \{ r'[e_1] \triangleright \{ r_1[e_2] \triangleright \alpha, r_1[e_3] \triangleright \beta \} \} ] \]
Simple vs dependent plans

With simple plans:

\[ H \equiv \{ r[\ell_1] | r_1[\ell_2] | r'[\ell_1] \triangleright \varphi[\alpha \cdot \alpha], \not \text{viable} \}
\]

\[ r[\ell_1] | r_1[\ell_3] | r'[\ell_1] \triangleright \varphi[\beta \cdot \beta] \not \text{viable} \}
\]

With dependent plans:

\[ H \equiv \{ r[\ell_1 \cdot r_1[\ell_2]] | r'[\ell_1 \cdot r_1[\ell_2]] \triangleright \varphi[\alpha \cdot \alpha], \not \text{viable} \}
\]

\[ r[\ell_1 \cdot r_1[\ell_2]] | r'[\ell_1 \cdot r_1[\ell_3]] \triangleright \varphi[\alpha \cdot \beta], \text{viable} \]

\[ r[\ell_1 \cdot r_1[\ell_3]] | r'[\ell_1 \cdot r_1[\ell_2]] \triangleright \varphi[\beta \cdot \alpha], \text{viable} \]

\[ r[\ell_1 \cdot r_1[\ell_3]] | r'[\ell_1 \cdot r_1[\ell_3]] \triangleright \varphi[\beta \cdot \beta] \not \text{viable} \} \]
Validity

- histories are enriched with $[\phi$ and $]_{\phi}$ to point out the scope of policies.
- a history is **valid** when all the policies are respected, within their scopes
  - ex: $\alpha_w \alpha_r [\phi \alpha_w]_{\phi}$ not valid (write after read)
  - ex: $\alpha_w [\phi \alpha_r]_{\phi} \alpha_w$ valid (write outside scope of $\phi$)
- a history expression $H$ is $\pi$-valid when all the histories in $[[H]]_{\pi}$ are valid.
Validity, formally

- **Safe sets:**
  - $S(\varepsilon) = 0 \quad S(\eta \alpha) = S(\eta)$
  - $S(\eta_0 [\varphi \eta_1 ]\varphi) = S(\eta_0 \eta_1 ) \cup \varphi[\text{flat}(\eta_0 ) \text{ flat pref}(\eta_1 ) ]$

- **Example:**
  \[
  S([\varphi \alpha [\psi \beta ]\psi \gamma ]\varphi) = S(\alpha [\psi \beta ]\psi \gamma ) \cup \varphi[\{\varepsilon, \alpha, \alpha\beta, \alpha\beta\gamma\}]
  = \Psi[\{\alpha, \alpha\beta\}], \varphi[\{\varepsilon, \alpha, \alpha\beta, \alpha\beta\gamma\}]
  \]

- **\( \eta \) is valid if, for each \( \varphi[\{\eta_1 , \ldots, \eta_k \}] \) in \( S(\eta) \):**
  \[
  \eta_i \models \varphi \quad \text{for} \ 1 \leq i \leq k
  \]
Verifying validity

Model checking: valid plans are viable
(drive executions that never go wrong)

- transform linearized history expression into BPAs (Basic Process Algebras)
- transform policies into scoped policies (in the form of Finite State Automata)
From history expressions to BPAs

Example

\[ H = \beta \cdot (\mu h. \alpha + h \cdot h + \varphi[h]) \]

\[ \text{BPA}(H) = \beta \cdot X, \quad \{ X = \alpha + X \cdot X + [\varphi \cdot X \cdot ]_\varphi \} \]

Theorem: \[ [[ H ]] = [[ \text{BPA}(H) ]] \]
From policies to scoped policies

Example

Chinese Wall policy: no write after read
From policies to scoped policies

Example

\[ A_\phi[ ] \]

[Diagram of a state transition graph with states q0, q1, q2, q3, and transitions labeled with \( \alpha_r, \alpha_w \), \( [\phi] \), and \( ]_{\phi} \).]
Theorem:

\( \eta \) valid iff

\( \eta \) accepted by \( A_{\phi[]} \)

for all \( \phi \) occurring in \( \eta \)

\( \eta \) w/o “redundant” framings \( \phi[...\phi[...]...] = \phi[... \ ... \ ...] \)
Model-checking BPAs with FSAs

Theorem:

H valid iff

\[
[[\text{BPA}(H)]] = \bigwedge_{\varphi \text{ in } H} A_{\varphi[\cdot]}
\]
Main result

Network $N = \ell_1\{e_1: \tau_1\} \parallel \ldots \parallel \ell_k\{e_k: \tau_k\}$

$\emptyset, H_i |- e_i: \tau_i$ for $1 \leq i \leq k$

If $H_i$ is $\pi$-valid then $\pi$ is viable for $e_i$
Summing up ...

- **hypothesis**: client with history expression $H$
- **linearization**: transform $H$ into a semantically equivalent $H'$ in linear form, i.e.:
  \[ H' = \{ \pi_1 \triangleright H_1 \ldots \pi_k \triangleright H_k \} \]
  and the $H_i$ have no planned selections.
- **verification**: model-check the $H_i$ for validity
- **theorem**: if $H_i$ is valid, then $\pi_i$ is viable
Conclusions

A linguistic framework for secure service composition

- safety framings, policies, req-by-contract
- type & effect system
- verification of effects, to extract viable plans

The orchestrator securely composes and runs service-based applications
Other issues considered

- **instrumentation**: how to compile local policies into local checks, in case that some policy may fail
- **resource creation**: how to create fresh resources
- **liveness**: how to deal with properties of the form “something good will eventually happen”
- **multi-choice and dependent plans**
Future work

- other kinds of plans (e.g. dynamic)
- other kinds of effects (e.g. sessions)
- safety framings and security protocols
- safety framings for information flow
- incremental analysis, when new services can be discovered at run-time
- trust relations between services
- spatial types and logics
References

- M. Bartoletti, P. Degano, G.L. Ferrari. Checking risky events is enough for local policies. *ICTCS’05*.

www.di.unipi.it/~bartolet/pub