Separation Logics for Probability, Concurrency, and *Security*

Kwing Hei Li (Heili)

Aarhus University

Doctoral Symposium 2025

Joint work with Alejandro Aguirre, Philipp G. Haselwarter,

Simon Oddershede Gergersen, Markus de Medeiros, Joseph Tassarotti, Lars Birkedal

Example: Password Storage

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{set}\; m\; u\; p \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match}\; \mathsf{get}\; m\; u\; \mathsf{with} \\
&\qquad \mathsf{Some}\; p' \Rightarrow p = p' \\
&\qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

We store passwords p of users u in a mutable map m.

Example: Password Storage

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{set}\ m\ u\ p \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match}\ \mathsf{get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}\ p' \Rightarrow p = p' \\
&\quad \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

We store passwords p of users u in a mutable map m.
This is not secure!

Example: Password Storage *with hash*

$$\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{set}\ m\ u\ (h(p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match}\ \mathsf{get}\ m\ u\ \mathsf{with} \\
&\qquad \mathsf{Some}(x) \Rightarrow x = h(p) \\
&\qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}$$

We now store the hash of the password instead.

Example: Password Storage *with hash*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{set}\ m\ u\ (h(p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(x) \Rightarrow x = h(p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

We now store the hash of the password instead.

People who use same passwords will have same hash stored!

Example: Password Storage *with hash and salt*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

We generate a salt (a random number from $0, \ldots, N$) for each call of setpw

Example: Password Storage *with hash and salt*

$$
\begin{aligned}
\mathsf{setpw}(m,u,p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m,u,p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad\ \mathsf{end}
\end{aligned}
$$

We generate a salt (a random number from $0, \ldots, N$) for each call of setpw

We now store both salt and result after hashing salt and password with hash function h

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathrm{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathrm{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

Randomness occur in two places:

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

Randomness occur in two places:

1. Generation of salt

Example: Password Storage *with salt*

$$\begin{aligned}
\mathsf{setpw}(m,u,p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad\ \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt}\cdot p)) \\
\mathsf{checkpw}(m,u,p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt}\cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad\ \mathsf{end}
\end{aligned}$$

Randomness occur in two places:

1. Generation of salt
2. Modelling hash function as random oracle

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad \mathsf{end}
\end{aligned}
$$

Observation 1:
randomness $\Rightarrow$ more complicated properties

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad\ \mathsf{end}
\end{aligned}
$$

Observation 1:

randomness $\Rightarrow$ more complicated properties

- checkpw with right password returns true

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad\ \mathsf{end}
\end{aligned}
$$

Observation 1:
randomness $\Rightarrow$ more complicated properties

- checkpw with right password returns true
- checkpw with wrong password returns false with *high probability*

Example: Password Storage *with salt*

$$\mathsf{setpw}(m, u, p) \triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in}$$
$$\mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p))$$

$$\mathsf{checkpw}(m, u, p) \triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with}$$
$$\mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p)$$
$$\mid \mathsf{None} \Rightarrow \mathsf{false}$$
$$\mathsf{end}$$

Observation 1:
randomness ⇒ more complicated properties

- checkpw with right password returns true
- checkpw with wrong password returns false with *high probability*
- password storage *appears random* to an outside observer

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{setpw}(m, u, p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad\ \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)) \\
\mathsf{checkpw}(m, u, p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\quad\ \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\
&\quad\ \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad\ \mathsf{end}
\end{aligned}
$$

Observation 2:
many complicated language features

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{setpw}(m,u,p) &\triangleq \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\
&\quad\ \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt}\cdot p)) \\
\mathsf{checkpw}(m,u,p) &\triangleq \mathsf{match\ get}\ m\ u\ \mathsf{with} \\
&\qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt}\cdot p) \\
&\qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\
&\quad\ \ \mathsf{end}
\end{aligned}
$$

Observation 2:
many complicated language features

- Dynamically allocated (potentially higher-order) mutable state

Example: Password Storage *with salt*

$$\mathsf{init} :: \mathsf{unit} \to \left(\begin{array}{l} \mathsf{setpw} : \mathsf{string} \to \mathsf{string} \to \mathsf{unit}, \\ \mathsf{checkpw} : \mathsf{string} \to \mathsf{string} \to \mathsf{bool} \end{array} \right)$$

$$\begin{array}{l} \mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in} \\ \quad \left(\begin{array}{l} \lambda u\, p.\ \mathsf{let\ salt} = \mathsf{rand}\ N\ \mathsf{in} \\ \qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)), \\ \lambda u\, p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ \qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\ \qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \quad \mathsf{end} \end{array} \right) \end{array}$$

Observation 2:
many complicated language features

- Dynamically allocated (potentially higher-order) mutable state
- Higher order functions

Example: Password Storage *with salt*

$$
\begin{aligned}
&\mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in} \\
&\qquad \left(\begin{aligned} &\lambda u\, p.\ \mathsf{let\ salt} = \mathsf{sample}\ N\ \mathsf{in} \\ &\qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)), \\ &\lambda u\, p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ &\qquad\quad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\ &\qquad\quad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ &\qquad \mathsf{end} \end{aligned} \right) \\
&\mathsf{sample}\ N \triangleq\ (\mathsf{rec}\ f\ \_ = \\
&\qquad\qquad \mathsf{let}\ x = \mathsf{rand\ MAX\ in} \\
&\qquad\qquad \mathsf{if}\ x \leqslant N\ \mathsf{then}\ x\ \mathsf{else}\ f\,()\ )\ ()
\end{aligned}
$$

Observation 2:
many complicated language features

- Dynamically allocated (potentially higher-order) mutable state
- Higher order functions
- Unbounded looping

Example: Password Storage *with salt*

$$
\begin{aligned}
\mathsf{init} &\triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in} \\
&\quad \left(\begin{array}{l} \lambda u\, p.\ \mathsf{let}\ \mathsf{salt} = \mathsf{sample}\ N\ \mathsf{in} \\ \qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt}\cdot p)), \\ \lambda u\, p.\ \mathsf{match}\ \mathsf{get}\ m\ u\ \mathsf{with} \\ \qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt}\cdot p) \\ \qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \quad \mathsf{end} \end{array} \right) \\
\mathsf{client} &\triangleq \mathsf{let}\ (\mathsf{setpw}, \mathsf{checkpw}) = \mathsf{init}\,()\ \mathsf{in} \\
&\quad \big(\mathsf{setpw}(u_1, p_1) \mathbin{|||} \mathsf{setpw}(u_2, p_2)\big); \\
&\quad \mathsf{checkpw}(u_1, p_2)
\end{aligned}
$$

Observation 2:
many complicated language features

- Dynamically allocated (potentially higher-order) mutable state
- Higher order functions
- Unbounded looping
- Concurrency in client

Example: Password Storage *with salt*

$$\mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in}$$

$$\left(\begin{array}{l} \lambda u\, p.\ \mathsf{let\ salt} = \mathsf{read}\ \textbf{/dev/random}\ \mathsf{in} \\ \qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)), \\ \lambda u\, p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ \qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\ \qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \quad \mathsf{end} \end{array} \right)$$

$\mathsf{generator} \triangleq$ *repeatedly writes random bits into* ***/dev/random***

Observation 2:
many complicated language features

- Dynamically allocated (potentially higher-order) mutable state
- Higher order functions
- Unbounded looping
- Concurrency in client & implementation...

Verifying real-world security programs $\Rightarrow$ Reasoning about probabilistic properties $+$ Using complicated language features

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, …(equivalences of programs, sensitivity, differential privacy)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, …(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, …(independence, conditioning, relational reasoning, etc.)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, ...(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, ...(independence, conditioning, relational reasoning, etc.)
- Outcome logic (independence, conditioning)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, …(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, …(independence, conditioning, relational reasoning, etc.)
- Outcome logic (independence, conditioning)
- Denotational semantics (contextual refinement)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, …(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, …(independence, conditioning, relational reasoning, etc.)
- Outcome logic (independence, conditioning)
- Denotational semantics (contextual refinement)
- Model checking (safety, liveness)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, …(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, …(independence, conditioning, relational reasoning, etc.)
- Outcome logic (independence, conditioning)
- Denotational semantics (contextual refinement)
- Model checking (safety, liveness)
- Fancy type systems (differential privacy, cost analysis)

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, ...(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, ...(independence, conditioning, relational reasoning, etc.)
- Outcome logic (independence, conditioning)
- Denotational semantics (contextual refinement)
- Model checking (safety, liveness)
- Fancy type systems (differential privacy, cost analysis)
- Refinement based approaches...

Proving complicated probabilistic properties

Various prior work on verifying probabilistic programs:

- Weakest pre-expectation calculi (expectations, error bounds, relational reasoning, etc.)
- Coupling-based logics, pRHL, apRHL, ...(equivalences of programs, sensitivity, differential privacy)
- Probabilistic separation logic, Lilac, Bluebell, ...(independence, conditioning, relational reasoning, etc.)
- Outcome logic (independence, conditioning)
- Denotational semantics (contextual refinement)
- Model checking (safety, liveness)
- Fancy type systems (differential privacy, cost analysis)
- Refinement based approaches...

Though they have various limitations, e.g. no shared state, higher-order functions,

Iris

Iris is a higher-order concurrent separation logic framework, formalized in *Rocq*

Iris

Iris is a higher-order concurrent separation logic framework, formalized in *Rocq*

Used to verify programs with many *challenging features*, e.g. higher-order functions, unstructured concurrency

Iris

Iris is a higher-order concurrent separation logic framework, formalized in *Rocq*

Used to verify programs with many *challenging features*, e.g. higher-order functions, unstructured concurrency

However, less work on using Iris to prove *probabilistic* properties...

Logics developed

PhD goal: Develop *probabilistic extensions* of Iris for highly expressive languages

Logics developed

PhD goal: Develop *probabilistic extensions* of Iris for highly expressive languages

	Unary	**Relational**
Sequential	Eris	Approxis
Concurrent	Coneris	Foxtrot

Logics developed

PhD goal: Develop *probabilistic extensions* of Iris for highly expressive languages

	Unary	**Relational**
Sequential	Eris	Approxis
Concurrent	Coneris	Foxtrot

Stage 1: develop Iris logics for sequential probabilistic programs

Logics developed

PhD goal: Develop *probabilistic extensions* of Iris for highly expressive languages

	Unary	**Relational**
Sequential	Eris	Approxis
Concurrent	Coneris	Foxtrot

Stage 1: develop Iris logics for sequential probabilistic programs

Stage 2: extend those logics to concurrent programs

Idealized collision-free hash

$$
\begin{aligned}
&\mathsf{let}\ x = h\ n\ \mathsf{in}\\
&\mathsf{let}\ y = h\ m\ \mathsf{in}\\
&(x, y)
\end{aligned}
$$

Idealized collision-free hash

$$\left\{ \; m \neq n \; \right\} \quad \begin{array}{l} \text{let } x = h\; n \text{ in} \\ \text{let } y = h\; m \text{ in} \\ (x, y) \end{array} \quad \left\{ \; (x, y).\; x \neq y \; \right\}$$

Useful to model the hash function as a collision-free random oracle

Idealized collision-free hash

$$\left\{\ m \neq n\ \right\} \quad \begin{array}{l} \mathsf{let}\ x = h\ n\ \mathsf{in} \\ \mathsf{let}\ y = h\ m\ \mathsf{in} \\ (x, y) \end{array} \quad \left\{\ (x, y).\ x \neq y\ \right\}$$

Useful to model the hash function as a collision-free random oracle

Hash is collision-free if different inputs map to different outputs

Idealized collision-free hash

$$\left\{ \; m \neq n \; \right\} \quad \begin{array}{l} \mathsf{let}\, x = h\, n\, \mathsf{in} \\ \mathsf{let}\, y = h\, m\, \mathsf{in} \\ (x, y) \end{array} \quad \left\{ \; (x, y).\; x \neq y \; \right\}$$

Useful to model the hash function as a collision-free random oracle

Hash is collision-free if different inputs map to different outputs

But this is not always true! Small probability of error!

Eris @ ICFP 2024

- *Eris* is a unary logic for proving error bounds of probabilistic programs

Eris @ ICFP 2024

- *Eris* is a unary logic for proving error bounds of probabilistic programs
- **KEY IDEA: We internalize error as a separation logic resource, aka *error credit***

Eris @ ICFP 2024

- *Eris* is a unary logic for proving error bounds of probabilistic programs
- **KEY IDEA: We internalize error as a separation logic resource, aka *error credit***
- $\lightning(\varepsilon)$ asserts ownership of ε error credits, with $\varepsilon \in [0, 1]$

Eris @ ICFP 2024

- *Eris* is a unary logic for proving error bounds of probabilistic programs
- **KEY IDEA: We internalize error as a separation logic resource, aka *error credit***
- $\lightning(\varepsilon)$ asserts ownership of ε error credits, with $\varepsilon \in [0, 1]$
- Adequacy: $\{\lightning(\varepsilon)\}\, e\, \{v.\phi(v)\} \Rightarrow \mathrm{Pr}_{\mathsf{exec}\, e}[\neg\phi] \leqslant \varepsilon$

- *Eris* is a unary logic for proving error bounds of probabilistic programs
- **KEY IDEA: We internalize error as a separation logic resource, aka *error credit***
- $⚡(\varepsilon)$ asserts ownership of ε error credits, with $\varepsilon \in [0, 1]$
- Adequacy: $\{⚡(\varepsilon)\}\, e\, \{v.\phi(v)\} \Rightarrow \Pr_{\mathsf{exec}\, e}[\neg\phi] \leqslant \varepsilon$
- Flexible rules to "spend" error credits to avoid undesirable error results:

$$\frac{\textsc{ht-rand-list}}{\vdash \{⚡(\mathsf{length}(xs)/(N+1))\}\ \mathsf{rand}\ N\, \{n\ .\ n \notin xs\}}$$

Eris example: Hash

Idealized collision-free hash function

$$\left\{ \begin{array}{l} \mathsf{collFree}(\mathsf{h}) \ * \\ n \notin \mathrm{dom}\, h \ * \\ \lightning\left(\frac{|\mathrm{dom}\, h|}{2^S}\right) \end{array} \right\}$$

$$h\ n$$

$$\{v.\, \mathsf{collFree}(\mathsf{h})\}$$

Eris example: Hash

Idealized collision-free hash function

$\Downarrow$

Amortized idealized collision-free hash function

$$\left\{ \begin{array}{c} \mathsf{collFreeAm}(\mathsf{h}) \ast \\ n \notin \mathrm{dom}\, h \ast \\ |h| < M \ast \\ ⚡(E_{\mathrm{const}}) \end{array} \right\}$$

$$h\ n$$

$$\{v.\, \mathsf{collFreeAm}(\mathsf{h})\}$$

Eris example: Hash

Idealized collision-free hash function

$\Downarrow$

Amortized idealized collision-free hash function

$$\left\{ \begin{array}{c} \mathsf{collFreeAm}(\mathsf{h}) \ * \\ n \notin \operatorname{dom} h \ * \\ |h| < M \ * \\ \lightning(\mathcal{E}_{\mathsf{const}}) \end{array} \right\}$$

$$h\ n$$

$$\{v.\ \mathsf{collFreeAm}(\mathsf{h})\}$$

Amortized hash specification used in verifying *Merkle tree* and *unreliable data storage system*

$$\mathsf{prf} \triangleq \lambda\_.\ \mathsf{rand}\, N$$

$$\mathsf{prp} \triangleq \begin{array}{l} \mathsf{let}\, l = \mathsf{ref}\, [\,]\ \mathsf{in} \\ \lambda\_.\ \mathsf{let}\, x = \mathsf{unif}\, (\{0, \ldots, N\} \setminus l)\ \mathsf{in} \\ \quad l \leftarrow x \cdot l; \\ \quad x \end{array}$$

$$\text{prf} \triangleq \lambda\_.\ \text{rand}\ N$$

$$\text{prp} \triangleq \begin{array}{l} \text{let}\ l = \text{ref}\ [\,]\ \text{in} \\ \lambda\_.\ \text{let}\ x = \text{unif}\ (\{0, \ldots, N\} \setminus l)\ \text{in} \\ \quad l \leftarrow x \cdot l; \\ \quad x \end{array}$$

Approxis re-introduce error credits to the relational setting for proving *approximate refinements*

$$\text{prf} \triangleq \lambda\_.\ \text{rand}\ N$$

$$\text{prp} \triangleq \begin{array}{l} \text{let}\ l = \text{ref}\,[\,]\ \text{in} \\ \lambda\_.\ \text{let}\ x = \text{unif}\,(\{0, \ldots, N\} \setminus l)\ \text{in} \\ \quad l \leftarrow x \cdot l; \\ \quad x \end{array}$$

Approxis re-introduce error credits to the relational setting for proving *approximate refinements*

Used in security-related examples: PRP/PRF switching lemma and IND$-CPA security of an encryption scheme

$$\mathsf{prf} \triangleq \lambda\_.\ \mathsf{rand}\ N$$

$$\mathsf{prp} \triangleq \begin{array}{l} \mathsf{let}\ l = \mathsf{ref}\,[\,]\ \mathsf{in} \\ \lambda\_.\ \mathsf{let}\ x = \mathsf{unif}\,(\{0, \ldots, N\} \setminus l)\ \mathsf{in} \\ \quad l \leftarrow x \cdot l; \\ \quad x \end{array}$$

Approxis re-introduce error credits to the relational setting for proving *approximate refinements*

Used in security-related examples: PRP/PRF switching lemma and IND$-CPA security of an encryption scheme

Built a logical refinement relation for contextual refinement, used to prove correctness of a B+ tree sampling scheme

Logics for Concurrency and Probability

Eris and Approxis are logics for sequential probabilistic programs

Logics for Concurrency and Probability

Eris and Approxis are logics for sequential probabilistic programs

We now extend them for concurrent probabilistic programs

Logics for Concurrency and Probability

Eris and Approxis are logics for sequential probabilistic programs

We now extend them for concurrent probabilistic programs

- Eris $\Rightarrow$ *Coneris* @ ICFP 2025
- Approxis $\Rightarrow$ *Foxtrot* (WIP)

Challenges in extending to Concurrency

These extensions to concurrency are non-trivial:

Challenges in extending to Concurrency

These extensions to concurrency are non-trivial:

1. In Coneris, we need to capture randomized logical atomicity to support modular specifications (More on this at my ICFP talk on Wednesday!)

Challenges in extending to Concurrency

These extensions to concurrency are non-trivial:

1. In Coneris, we need to capture randomized logical atomicity to support modular specifications (More on this at my ICFP talk on Wednesday!)
2. Some rules in Approxis are unsound in Foxtrot

Challenges in extending to Concurrency

These extensions to concurrency are non-trivial:

1. In Coneris, we need to capture randomized logical atomicity to support modular specifications (More on this at my ICFP talk on Wednesday!)
2. Some rules in Approxis are unsound in Foxtrot

We need to redesign the model of the logics and introduce new logical facilities and proof techniques

Examples of Coneris and Foxtrot

- Modular specifications of *thread-safe* hashes

Examples of Coneris and Foxtrot

- Modular specifications of *thread-safe* hashes
- Strict error bounds of *concurrent* Bloom filter

Examples of Coneris and Foxtrot

- Modular specifications of *thread-safe* hashes
- Strict error bounds of *concurrent* Bloom filter
- Sodium sampling function:

$$
\begin{array}{l}
\lambda N.\ \text{if } N < 2 \text{ then } 0 \\
\qquad \text{else let min} = \text{MAX mod } N \text{ in} \\
\qquad \text{let } r = \text{ref } 0 \text{ in} \\
\qquad \left(\begin{array}{l} \text{rec} f\ \_ = r \leftarrow \text{rand}(\text{MAX} - 1); \\ \ \text{if } !r < \text{min then } f() \\ \ \text{else } (!r \text{ mod } N) \end{array} \right) ()
\end{array}
\quad \simeq_{\text{ctx}} \quad \lambda N.\ \text{if } N = 0 \text{ then } 0 \text{ else } \text{rand}(N - 1)
$$

Revisiting Password Storage

$$
\begin{array}{l}
\mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in} \\
\qquad \left(\begin{array}{l} \lambda u\, p.\ \mathsf{let\ salt} = \mathsf{sample}\ N\ \mathsf{in} \\ \qquad\quad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)), \end{array} \right. \\
\qquad\ \ \left. \begin{array}{l} \lambda u\, p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ \qquad\quad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\ \qquad\quad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \qquad\ \mathsf{end} \end{array} \right)
\end{array}
$$

- checkpw with wrong password returns false with *high probability* $\Rightarrow$ Eris

Revisiting Password Storage

$$
\begin{aligned}
\text{init} \triangleq \lambda\_.\ &\text{let } m = \text{init}\,()\ \text{in} \\
&\left(\begin{aligned} &\lambda u\, p.\ \text{let salt} = \text{sample}\ N\ \text{in} \\ &\qquad \text{set}\ m\ u\ (\text{salt}, h(\text{salt} \cdot p)), \end{aligned} \right. \\
&\quad \left. \begin{aligned} &\lambda u\, p.\ \text{match get}\ m\ u\ \text{with} \\ &\qquad \text{Some}(\text{salt}, x) \Rightarrow x = h(\text{salt} \cdot p) \\ &\qquad \mid \text{None} \Rightarrow \text{false} \\ &\quad \text{end} \end{aligned} \right)
\end{aligned}
$$

- checkpw with wrong password returns false with *high probability* $\Rightarrow$ Eris
- password storage *appears random* to an outside observer $\Rightarrow$ Approxis

Revisiting Password Storage

$$
\begin{aligned}
&\text{init} \triangleq \lambda\_.\ \text{let } m = \text{init}\,()\ \text{in} \\
&\quad \left(\begin{array}{l} \lambda u\, p.\ \text{let salt} = \text{sample } N \text{ in} \\ \qquad \text{set } m\, u\, (\text{salt}, h(\text{salt} \cdot p)), \\ \lambda u\, p.\ \text{match get } m\, u \text{ with} \\ \qquad \text{Some}(\text{salt}, x) \Rightarrow x = h(\text{salt} \cdot p) \\ \qquad \mid \text{None} \Rightarrow \text{false} \\ \quad \text{end} \end{array} \right) \\
&\text{client} \triangleq \text{let } (\text{setpw}, \text{checkpw}) = \text{init}\,()\ \text{in} \\
&\qquad (\text{setpw}(u_1, p_1) \mid\mid\mid \text{setpw}(u_2, p_2)); \\
&\qquad \text{checkpw}(u_1, p_2)
\end{aligned}
$$

- checkpw with wrong password returns false with *high probability* $\Rightarrow$ Eris
- password storage *appears random* to an outside observer $\Rightarrow$ Approxis
- concurrency in the client side $\Rightarrow$ Coneris or Foxtrot

Revisiting Password Storage

$$\mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in}$$
$$\left(\begin{array}{l} \lambda u\, p.\ \mathsf{let\ salt} = \mathsf{read}\ \textbf{/dev/random}\ \mathsf{in} \\ \qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)), \\ \lambda u\, p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ \qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\ \qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \quad \mathsf{end} \end{array} \right)$$

$$\mathsf{generator} \triangleq \textit{repeatedly writes random bits into } \textbf{/dev/random}$$

- checkpw with wrong password returns false with *high probability* $\Rightarrow$ Eris
- password storage *appears random* to an outside observer $\Rightarrow$ Approxis
- concurrency in the client side $\Rightarrow$ Coneris or Foxtrot
- concurrency in implementation side $\Rightarrow$ work in progress!

Revisiting Password Storage

$$\mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in}$$
$$\left(\begin{array}{l} \lambda u\, p.\ \mathsf{let\ salt} = \mathsf{read}\ \textbf{/dev/random}\ \mathsf{in} \\ \qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt}\cdot p)), \\ \lambda u\, p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ \qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt}\cdot p) \\ \qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \quad \mathsf{end} \end{array} \right)$$

$$\mathsf{generator} \triangleq \textit{repeatedly writes random bits into}\ \textbf{/dev/random}$$

- checkpw with wrong password returns false with *high probability* ⇒ Eris
- password storage *appears random* to an outside observer ⇒ Approxis
- concurrency in the client side ⇒ Coneris or Foxtrot
- concurrency in implementation side ⇒ work in progress!

Why? The schedulers are too powerful. (Well-known issue in various security models)

Revisiting Password Storage

$$\mathsf{init} \triangleq \lambda\_.\ \mathsf{let}\ m = \mathsf{init}\,()\ \mathsf{in}$$

$$\left(\begin{array}{l} \lambda u\,p.\ \mathsf{let\ salt} = \mathsf{read}\ \textbf{/dev/random}\ \mathsf{in} \\ \qquad \mathsf{set}\ m\ u\ (\mathsf{salt}, h(\mathsf{salt} \cdot p)), \end{array} \right.$$

$$\left. \begin{array}{l} \lambda u\,p.\ \mathsf{match\ get}\ m\ u\ \mathsf{with} \\ \qquad \mathsf{Some}(\mathsf{salt}, x) \Rightarrow x = h(\mathsf{salt} \cdot p) \\ \qquad \mid \mathsf{None} \Rightarrow \mathsf{false} \\ \quad \mathsf{end} \end{array} \right)$$

$\mathsf{generator} \triangleq$ *repeatedly writes random bits into* **/dev/random**

- checkpw with wrong password returns false with *high probability* ⇒ Eris
- password storage *appears random* to an outside observer ⇒ Approxis
- concurrency in the client side ⇒ Coneris or Foxtrot
- concurrency in implementation side ⇒ work in progress!

Why? The schedulers are too powerful. (Well-known issue in various security models)

Can we develop logics for reasoning about more restricted schedulers?

Conclusion

Two challenges in verifying real-world security programs:

1. Complicated probabilistic properties
2. Programs use complicated language features

Conclusion

Two challenges in verifying real-world security programs:

1. Complicated probabilistic properties
2. Programs use complicated language features

Much success with implementing logics within *Iris*:

	Unary	**Relational**
Sequential	Eris	Approxis
Concurrent	Coneris	Foxtrot

Conclusion

Two challenges in verifying real-world security programs:

1. Complicated probabilistic properties
2. Programs use complicated language features

Much success with implementing logics within *Iris*:

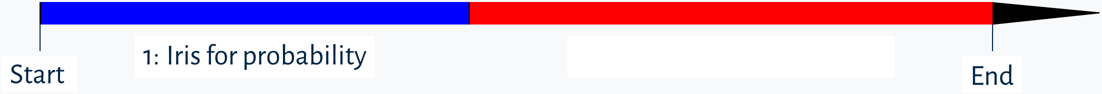
	Unary	**Relational**
Sequential	Eris	Approxis
Concurrent	Coneris	Foxtrot

Future work:

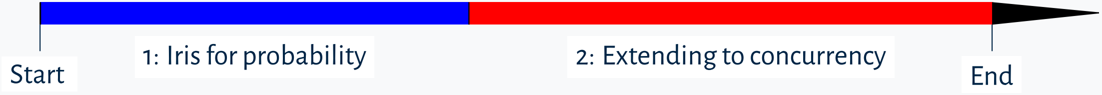
1. Improving concurrency model of Coneris and Foxtrot
2. Applying it to verify actual implementations of cryptographic libraries and protocols

APPENDIX

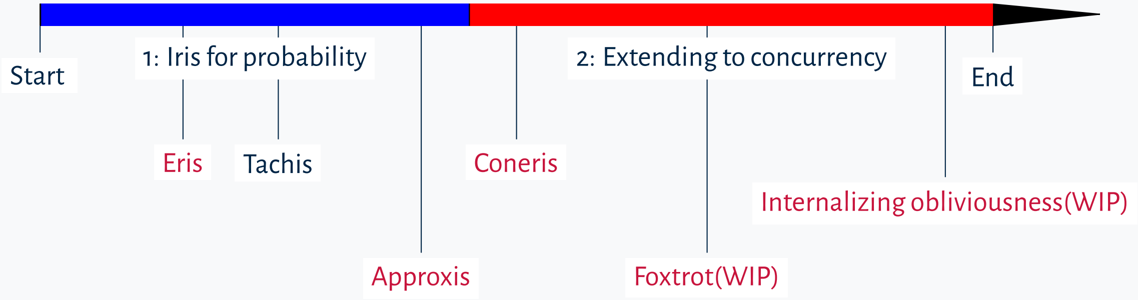
Timeline



Timeline


Start
1: Iris for probability
2: Extending to concurrency
End

Timeline



Start
1: Iris for probability
Eris
Tachis
Approxis
Coneris
2: Extending to concurrency
Foxtrot(WIP)
Internalizing obliviousness(WIP)
End

Eris rules

Eris rules

1. $\lightning(\varepsilon_1) * \lightning(\varepsilon_2) \dashv\vdash \lightning(\varepsilon_1 + \varepsilon_2)$

Eris rules

1. $\lightning(\varepsilon_1) * \lightning(\varepsilon_2) \dashv\vdash \lightning(\varepsilon_1 + \varepsilon_2)$
2. $\lightning(1) \vdash \bot$

Eris rules

1. $\lightning(\varepsilon_1) * \lightning(\varepsilon_2) \dashv\vdash \lightning(\varepsilon_1 + \varepsilon_2)$
2. $\lightning(1) \vdash \bot$
3. $$\frac{\sum_{i=0}^{N} \dfrac{\mathcal{F}(i)}{N+1} \leqslant \varepsilon}{\vdash \{\lightning(\varepsilon)\}\ \mathsf{rand}\, N\, \{n\,.\, \lightning(\mathcal{F}(n))\}}\ \textsc{ht-rand-exp}$$

Eris rules

1. $\lightning(\varepsilon_1) * \lightning(\varepsilon_2) \dashv\vdash \lightning(\varepsilon_1 + \varepsilon_2)$
2. $\lightning(1) \vdash \bot$
3. $$\dfrac{\sum_{i=0}^{N} \dfrac{\mathcal{F}(i)}{N+1} \leqslant \varepsilon}{\vdash \{\lightning(\varepsilon)\}\ \mathsf{rand}\, N\, \{n\,.\, \lightning(\mathcal{F}(n))\}}\ \textsc{ht-rand-exp}$$

$$\begin{tikzcd}
 & \boxed{0\,;\ \lightning(\mathcal{F}(0))} \\
\boxed{\mathsf{rand}\, 1\,;\ \lightning(\varepsilon)} \arrow[ur] \arrow[dr] & \\
 & \boxed{1\,;\ \lightning(\mathcal{F}(1))}
\end{tikzcd}$$

$$\frac{\mathcal{F}(0) + \mathcal{F}(1)}{2} \leqslant \varepsilon$$

Logical Relations in Approxis

We build a logical refinement relation in Approxis for proving contextual refinement

Logical Relations in Approxis

We build a logical refinement relation in Approxis for proving contextual refinement

You can assume ownership of some non-zero amount of error credits with the logical refinement relation!

$$\frac{\forall \varepsilon > 0.\, \lightning(\varepsilon) \twoheadrightarrow \Delta \vDash e \precsim e' : \tau}{\Delta \vDash e \precsim e' : \tau}$$

Logical Relations in Approxis

We build a logical refinement relation in Approxis for proving contextual refinement

You can assume ownership of some non-zero amount of error credits with the logical refinement relation!

$$\frac{\forall \varepsilon > 0.\ \lightning(\varepsilon) \twoheadrightarrow* \Delta \vDash e \precsim e' : \tau}{\Delta \vDash e \precsim e' : \tau}$$

$$\begin{array}{l} \mathsf{rec}\, f\, \_ = \\ \ \mathsf{let}\, x = \mathsf{rand}\, N\, \mathsf{in} \\ \ \ \mathsf{if}\, x \leqslant M\, \mathsf{then}\, x\, \mathsf{else}\, f\,() \end{array} \quad \simeq_{\mathsf{ctx}} \quad \lambda\, \_.\ \mathsf{rand}\, M$$

Logical Relations in Approxis

We build a logical refinement relation in Approxis for proving contextual refinement

You can assume ownership of some non-zero amount of error credits with the logical refinement relation!

$$\frac{\forall \varepsilon > 0.\ \lightning(\varepsilon) \twoheadrightarrow \Delta \vDash e \precsim e' : \tau}{\Delta \vDash e \precsim e' : \tau}$$

$$\begin{array}{lcr} \mathsf{rec}\, f\ \_ = & & \\ \quad \mathsf{let}\ x = \mathsf{rand}\ N\ \mathsf{in} & \simeq_{\mathsf{ctx}} & \lambda\,\_.\ \mathsf{rand}\ M \\ \quad \mathsf{if}\ x \leqslant M\ \mathsf{then}\ x\ \mathsf{else}\ f\,() & & \end{array}$$

Used in proving correctness of a rejection sampling scheme from B+ tree (developed by Olken and Rotem 1989s)

- Can we use error credits to reason about error bounds of *concurrent* probabilistic programs?

Coneris @ ICFP 2025

- Can we use error credits to reason about error bounds of *concurrent* probabilistic programs?
- Yes! With *Coneris*!

- Can we use error credits to reason about error bounds of *concurrent* probabilistic programs?
- Yes! With *Coneris*!
- $\{\lightning(\varepsilon)\}\, e\, \{v.\phi(v)\} \Rightarrow$ ***for all possible schedulers*** s, $\mathrm{Pr}_{\mathsf{exec}\, s,e}[\neg\phi] \leqslant \varepsilon$

- Can we use error credits to reason about error bounds of *concurrent* probabilistic programs?
- Yes! With *Coneris*!
- $\{↯(\varepsilon)\}\, e\, \{v.\phi(v)\} \Rightarrow$ ***for all possible schedulers*** s, $\Pr_{\mathsf{exec}\, s,e}[\neg\phi] \leqslant \varepsilon$
- Inherits all the error credit rules of Eris

- Can we use error credits to reason about error bounds of *concurrent* probabilistic programs?
- Yes! With *Coneris*!
- $\{\lightning(\varepsilon)\}\, e\, \{v.\phi(v)\} \Rightarrow$ ***for all possible schedulers*** s, $\Pr_{\mathsf{exec}\, s, e}[\neg\phi] \leqslant \varepsilon$
- Inherits all the error credit rules of Eris
- Error credits can be placed in invariants!

$$
\begin{array}{l}
\{\lightning(1/16)\} \\
\qquad \mathsf{let}\ l = \mathsf{ref}\, 0\ \mathsf{in} \\
\qquad (\mathsf{faa}\ l\ (\mathsf{rand}\ 3) \mathrel{|||} \mathsf{faa}\ l\ (\mathsf{rand}\ 3))\,; \\
\qquad !\, l \\
\{v.v > 0\}
\end{array}
$$

Coneris – Modularity

- Writing modular specifications for concurrent modules is known to be challenging

Coneris – Modularity

- Writing modular specifications for concurrent modules is known to be challenging
- Traditional Iris logics use $\mathrel{|}\!\!\!\Rrightarrow$ to capture logical atomicity (linearization point).

Coneris – Modularity

- Writing modular specifications for concurrent modules is known to be challenging
- Traditional Iris logics use $\Rrightarrow$ to capture logical atomicity (linearization point). But this is not enough if we also have *probability*!

Coneris – Modularity

- Writing modular specifications for concurrent modules is known to be challenging
- Traditional Iris logics use $\mid\!\!\Rrightarrow$ to capture logical atomicity (linearization point). But this is not enough if we also have *probability*!
- We introduce the probabilistic update modality $\mid\!\!\rightsquigarrow$ to capture *randomized logical atomicity*

Coneris – Modularity

- Writing modular specifications for concurrent modules is known to be challenging
- Traditional Iris logics use $\mid\!\Rrightarrow$ to capture logical atomicity (linearization point). But this is not enough if we also have *probability*!
- We introduce the probabilistic update modality $\mid\!\rightsquigarrow$ to capture *randomized logical atomicity*
- Used to prove specification of a thread safe hash module and concurrent bloom filter (novel result)

Foxtrot (WIP)

- Can we also extend Approxis to reason about approximate equivalence of *concurrent* probabilistic programs?

Foxtrot (WIP)

- Can we also extend Approxis to reason about approximate equivalence of *concurrent* probabilistic programs?
- Yes! With *Foxtrot*!

Foxtrot (WIP)

- Can we also extend Approxis to reason about approximate equivalence of *concurrent* probabilistic programs?
- Yes! With *Foxtrot*!
- $\{\lightning(\varepsilon) \ast \circ \Rrightarrow e'\}\, e\, \{v.\exists v'.\, \circ \Rrightarrow v'\} \implies \mathsf{exec}^{\sqcup\Downarrow}(e, \sigma) \leqslant \mathsf{exec}^{\sqcup\Downarrow}(e', \sigma) + \varepsilon$

Foxtrot (WIP)

- Can we also extend Approxis to reason about approximate equivalence of *concurrent* probabilistic programs?
- Yes! With *Foxtrot*!
- $\{\lightning(\varepsilon) * \circ \Rrightarrow e'\}\, e\, \{v.\exists v'.\, \circ \Rrightarrow v'\} \implies \mathsf{exec}^{\sqcup\Downarrow}(e,\sigma) \leqslant \mathsf{exec}^{\sqcup\Downarrow}(e',\sigma) + \varepsilon$
- Does not inherit all the rules of Approxis!

THIS-IS-UNSOUND

$$\frac{\kappa' \hookrightarrow_s (N,\vec{m}) \qquad \begin{array}{c}\kappa \hookrightarrow (N,\vec{n}) \\ \forall v.\, \kappa \hookrightarrow (N,\vec{n} +\!\!+ v) * \kappa' \hookrightarrow_s (N,\vec{m} +\!\!+ v) \mathrel{-\!\!*} \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}\end{array}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}$$

UNSOUND-HT-COUPLE-RAND-LBL-EXACT

$$\frac{\forall n \leqslant N.\, \{\kappa \hookrightarrow_s (N,\vec{n}\cdot[n])\}\, n\, \{\Phi\}}{\{\kappa \hookrightarrow_s (N,\vec{n})\}\ \mathsf{rand}\, N\, \{\Phi\}}$$

Foxtrot (WIP)

- Can we also extend Approxis to reason about approximate equivalence of *concurrent* probabilistic programs?
- Yes! With *Foxtrot*!
- $\{\lightning(\varepsilon) * \circ \Mapsto e'\}\, e\, \{v. \exists v'. \circ \Mapsto v'\} \implies \mathsf{exec}^{\sqcup\Downarrow}(e, \sigma) \leqslant \mathsf{exec}^{\sqcup\Downarrow}(e', \sigma) + \varepsilon$
- Does not inherit all the rules of Approxis!

THIS-IS-UNSOUND

$$\frac{\kappa' \hookrightarrow_s (N, \vec{m}) \qquad \begin{array}{c}\kappa \hookrightarrow (N, \vec{n}) \\ \forall v.\, \kappa \hookrightarrow (N, \vec{n} \mathbin{+\!\!+} v) * \kappa' \hookrightarrow_s (N, \vec{m} \mathbin{+\!\!+} v) \mathrel{-\!\!*} \mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}\end{array}}{\mathsf{rwp}\ e_1 \precsim e_2\ \{\Phi\}}$$

UNSOUND-HT-COUPLE-RAND-LBL-EXACT

$$\frac{\forall n \leqslant N.\, \{\kappa \hookrightarrow_s (N, \vec{n} \cdot [n])\}\, n\, \{\Phi\}}{\{\kappa \hookrightarrow_s (N, \vec{n})\}\ \mathsf{rand}\ N\, \{\Phi\}}$$

Challenge: Model of Foxtrot is very different from that of Approxis.

Foxtrot examples

Algebraic theory:

$$(e_1 \oplus_p e_2) \oplus_q e_3 \simeq_{\mathsf{ctx}} e_1 \oplus_{pq} (e_2 \oplus_{\frac{q-pq}{1-pq}} e_3)$$

$$e_1 \mathbf{\ or\ } (e_2 \mathbf{\ or\ } e_3) \simeq_{\mathsf{ctx}} (e_1 \mathbf{\ or\ } e_2) \mathbf{\ or\ } e_3$$

$$e_1 \mathbf{\ or\ } (\mathsf{diverge}\,()) \simeq_{\mathsf{ctx}} e_1$$

Foxtrot examples

Algebraic theory:

$$(e_1 \oplus_p e_2) \oplus_q e_3 \simeq_{\mathsf{ctx}} e_1 \oplus_{pq} (e_2 \oplus_{\frac{q-pq}{1-pq}} e_3)$$

$$e_1 \;\mathbf{or}\; (e_2 \;\mathbf{or}\; e_3) \simeq_{\mathsf{ctx}} (e_1 \;\mathbf{or}\; e_2) \;\mathbf{or}\; e_3$$

$$e_1 \;\mathbf{or}\; (\mathsf{diverge}\,()) \simeq_{\mathsf{ctx}} e_1$$

Libsodium random sampling implementation:

$$\begin{array}{l} \lambda N.\ \mathsf{if}\ N < 2\ \mathsf{then}\ 0 \\ \qquad \mathsf{else\ let\ min} = \mathsf{MAX\ mod}\ N\ \mathsf{in} \\ \qquad \mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in} \\ \qquad \left(\begin{array}{l} \mathsf{rec}\, f\ \_ = r \leftarrow \mathsf{rand}(\mathsf{MAX} - 1); \\ \ \mathsf{if}\ !\, r < \mathsf{min\ then}\ f() \\ \ \ \mathsf{else}\ (!\, r\ \mathsf{mod}\ N) \end{array} \right) () \end{array} \quad \simeq_{\mathsf{ctx}} \quad \lambda N.\ \mathsf{if}\ N = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{rand}(N-1)$$

Oblivious scheduler example

$$\text{let } x = \text{rand } 1 \text{ in}$$
$$\text{choose}(x = 0, x = 1)$$

Oblivious scheduler example

$$\text{let } x = \text{rand } 1 \text{ in}$$
$$\text{choose}(x = 0, x = 1)$$