Hopping Proofs of Expectation-Based Properties

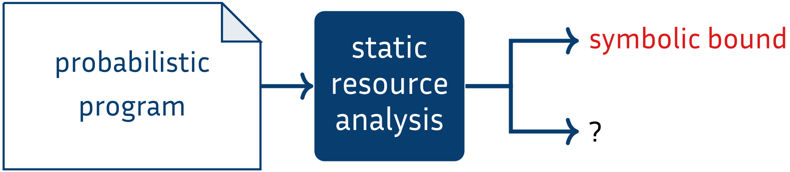
Applications to Skiplists and Security Proofs

Martin Avanzini, Gilles Barthe, Benjamin Grégoire, Georg Moser and Gabriele Vanoni

Journée du PEPR cybersécurité SVP (06 / 02 / 2024)

This work received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0006.

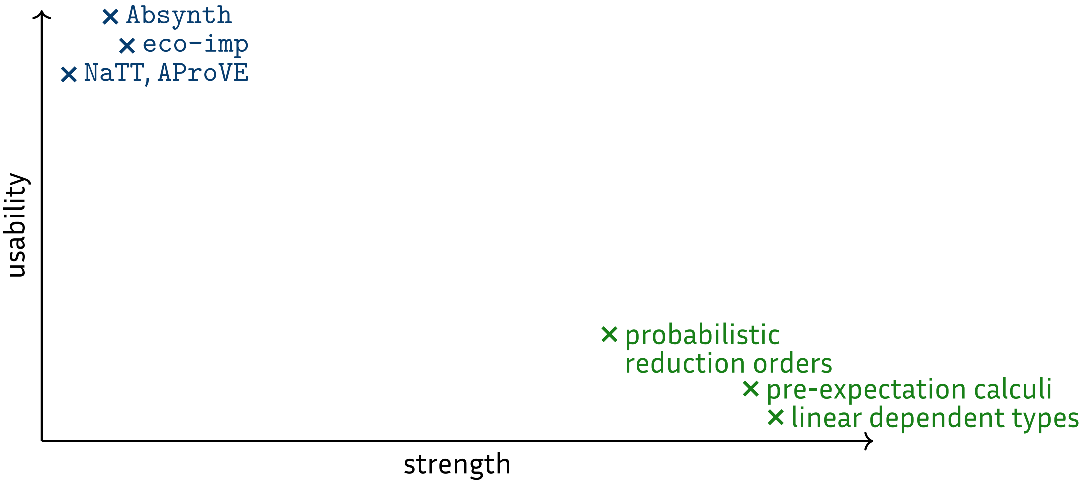
Initial motivation



Properties of interest

1. strength (applicability, precision)
2. usability

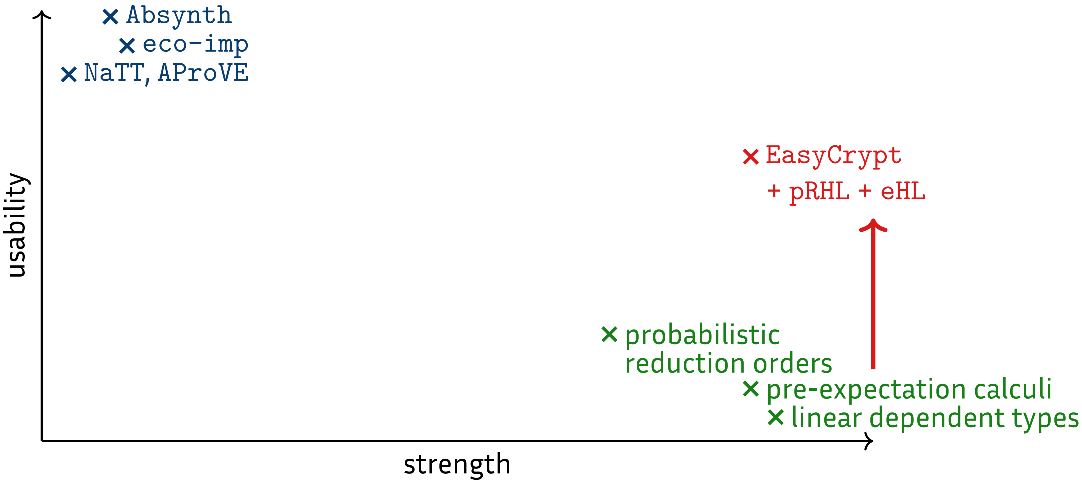
Initial motivation



- – inherently incomplete
- + push-button tools

- + (relative) complete logics
- – tedious, complicated, *pen-and-paper*

Initial motivation



- inherently incomplete
+ push-button tools

+ (relative) complete logics
- tedious, complicated, *pen-and-paper*

Three central ingredients

1. `EasyCrypt`
   - interactive theorem prover for construction and verification of cryptographic proofs
   - built-in imperative programming language with sampling instructions; no heap
   - higher-order ambient logic
   - partial automation (`smt`, tactics `auto`, `wp`, ...)

Three central ingredients

1. EasyCrypt
   - interactive theorem prover for construction and verification of cryptographic proofs
   - built-in imperative programming language with sampling instructions; no heap
   - higher-order ambient logic
   - partial automation (smt, tactics auto, wp, ...)

2. Probabilistic relational Hoare-logic (pRHL)

$$\{\, \mathtt{P} \,\}\ \mathtt{C}_1 \sim \mathtt{C}_2\ \{\, \mathtt{Q} \,\} \qquad (\textit{valid if for all } m_1 m_2,\ \mathtt{P}\, m_1\, m_2 \Rightarrow \mathtt{Q}^{\dagger}\, [\![\mathtt{C}_1]\!]_{m_1}\, [\![\mathtt{C}_2]\!]_{m_2})$$

   - variation of Hoare logic for relating programs
   - fascilitates program abstraction, preserving cost

Three central ingredients

1. `EasyCrypt`
   - interactive theorem prover for construction and verification of cryptographic proofs
   - built-in imperative programming language with sampling instructions; no heap
   - higher-order ambient logic
   - partial automation (`smt`, tactics `auto`, `wp`, ...)

2. Probabilistic relational Hoare-logic (`pRHL`)

$$\{\, \mathtt{P} \,\}\ \mathtt{C}_1 \sim \mathtt{C}_2\ \{\, \mathtt{Q} \,\} \qquad \left(\textit{valid if for all } m_1 m_2,\ \mathtt{P}\, m_1\, m_2 \Rightarrow \mathtt{Q}^{\dagger}\ [\![\mathtt{C}_1]\!]_{m_1}\ [\![\mathtt{C}_2]\!]_{m_2}\right)$$

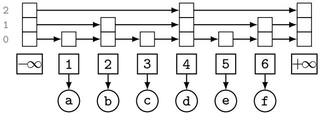
   - variation of Hoare logic for relating programs
   - fascilitates program abstraction, preserving cost

3. Expectation Hoare-logic (`eHL`)

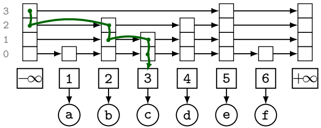
$$\{\, f \,\}\ \mathtt{C}\ \{\, g \,\} \qquad \left(\textit{valid if for all} m,\ \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \leq f\, m\right)$$

   - variation of Hoare Logic for reasoning about expectations
   - facilitates average cost analysis by measuring cost-counter

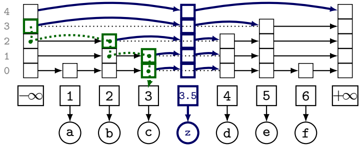
Driving example: skip lists



(a) Perfectly balanced skip list with 3 levels.



(b) Searching for value associated with key $\mathtt{k} = 3$.

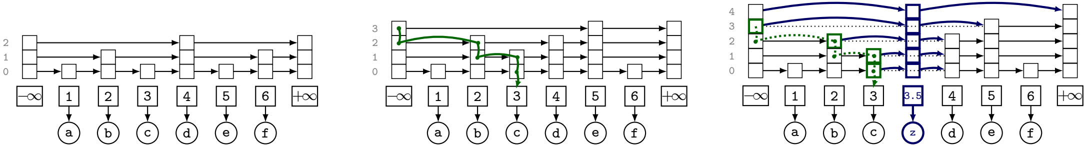


(c) Inserting $\mathtt{z}$ with key $\mathtt{k}$ at sampled height 5 in (b).

- ⋆ probabilistic alternative to binary search trees
- ⋆ good average case complexity $O(\log n)$ for dictionary operations (search, insert, delete)
- ⋆ elegant but informal backward analysis, very difficult to formalise

```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        -- x→key < searchKey   x→forward[i]→key
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        lvl := randomLevel()
        if lvl > list→level then
            for i := list→level + 1 to lvl do
                update[i] := list→header
            list→level := lvl
        x := makeNode(lvl, searchKey, value)
        for i := 1 to level do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x
```

Driving example: skip lists



(a) Perfectly balanced skip list with 3 levels.

(b) Searching for value associated with key k = 3.

(c) Inserting z with key k at sampled height 5 in (b).

```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        -- x→key < searchKey    x→forward[i]→key
        update[i] := x
    x→forward[1]
    →key = searchKey then x→value := newValue

    vl := randomLevel()
    f lvl > list→level then
        for i := list→level + 1 to lvl do
            update[i] := list→header
        list→level := lvl
    := makeNode(lvl, searchKey, value)
    or i := 1 to level do
        x→forward[i] := update[i]→forward[i]
        update[i]→forward[i] := x
```

- ⋆ probabilistic alternative to binary search trees
- ⋆ good average case complexity $O(\log n)$ for dictionary operations (search, insert, delete)

Final contribution

optimal (up-to small constant) *bound* on avg. search complexity of *concrete implementation*, fully verified (~2.500 lines proof)

$$\{\, 2\log_2(|\mathtt{lst}| + 1) + 4 \,\}\ \mathtt{sl} \xleftarrow{\$} \mathtt{from\_list}(\mathtt{lst});\ \mathtt{find}(\mathtt{k}, \mathtt{sl})\ \{\, \mathtt{ct} \,\}$$

Expectation Hoare-logic (eHL)

- ⋆ generalises classical Hoare Logic, inspired by probabilistic pre-condition calculi
  - \+ complete, relative to transient logic of EasyCrypt
  - \+ encompasses reasoning about probabilities of events
  - \+ pRHL rule for “game-hopping proofs”
  - \+ adversary rule for game-based cryptographic proofs

Dexter Kozen. *“A probabilistic PDL”*. *Journal of Computer and System Sciences, Vol. 30*, pp. 162–178, 1985.

Annabelle McIver and Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. *Abstraction, refinement and proof for probabilistic systems*, 2005.

Expectation Hoare-logic (eHL)

- ⋆ generalises classical Hoare Logic, inspired by probabilistic pre-condition calculi
  - + complete, relative to transient logic of EasyCrypt
  - + encompasses reasoning about probabilities of events
  - + pRHL rule for "game-hopping proofs"
  - + adversary rule for game-based cryptographic proofs
  - − only upper-bounds (lower-bounds, only indirectly for lossless programs)
  - − only non-negative functions

Dexter Kozen. *"A probabilistic PDL"*. *Journal of Computer and System Sciences, Vol. 30*, pp. 162–178, 1985.

Annabelle McIver and Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. *Abstraction, refinement and proof for probabilistic systems*, 2005.

Expectation Hoare-logic (eHL)

- ⋆ generalises classical Hoare Logic, inspired by probabilistic pre-condition calculi
  - + complete, relative to transient logic of `EasyCrypt`
  - + encompasses reasoning about probabilities of events
  - + pRHL rule for "game-hopping proofs"
  - + adversary rule for game-based cryptographic proofs
  - – only upper-bounds (lower-bounds, only indirectly for lossless programs)
  - – only non-negative functions
- ⋆ available in `EasyCrypt` since version `r2023.09`
  - – many derived rules supporting `EasyCrypt`'s bottom-up style reasoning
  - – procedure declarations and frame rule for modular reasoning
  - – `xreal` library formalising extended reals
    - ◦ finite & infinite sums
    - ◦ expectations and laws (e.g., linearity, Jensen's inequality, ...)
    - ◦ ...

Judgements

$$\vDash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, \mathtt{Q} \mid g \,\}$$

for all initial states m satisfying classical pre-condition $\mathtt{P}$,

1. f is an upper-bound to expected value of g after running $\mathtt{C}$ $\qquad \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \leq f\,m$
2. output satisfies post-condition $\mathtt{Q}$ with certainty $\qquad m' \in \mathrm{supp}([\![\mathtt{C}]\!]_m) \Rightarrow \mathtt{Q}\,m'$

Judgements

$$\vDash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, \mathtt{Q} \mid g \,\}$$

for all initial states m satisfying classical pre-condition $\mathtt{P}$,

1. f is an upper-bound to expected value of g after running $\mathtt{C}$ $\qquad \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \leq f\,m$
2. output satisfies post-condition $\mathtt{Q}$ with certainty $\qquad m' \in \mathrm{supp}([\![\mathtt{C}]\!]_m) \Rightarrow \mathtt{Q}\,m'$

Notes

- $f, g : \mathrm{Mem} \to [0, \infty]$ are non-negative
- shallow embedding of classical conditions: $(\,\mathtt{P} \mid f\,)\,m \triangleq \mathtt{if}\ \mathtt{P}\,m\ \mathtt{then}\ f\,m\ \mathtt{else}\ \infty$

Judgements

$$\vDash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, \mathtt{Q} \mid g \,\}$$

for all initial states m satisfying classical pre-condition $\mathtt{P}$,

1. f is an upper-bound to expected value of g after running $\mathtt{C}$ $\qquad \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \leq f\, m$
2. output satisfies post-condition $\mathtt{Q}$ with certainty $\qquad m' \in \mathsf{supp}([\![\mathtt{C}]\!]_m) \Rightarrow \mathtt{Q}\, m'$

Notes

- ⋆ $f, g : \mathsf{Mem} \to [0, \infty]$ are non-negative
- ⋆ shallow embedding of classical conditions: $(\, \mathtt{P} \mid f \,)\, m \triangleq \mathtt{if}\ \mathtt{P}\, m\ \mathtt{then}\ f\, m\ \mathtt{else}\ \infty$

Examples

$$\{\, 0 \leq \mathtt{y} \mid {}^1\!/_2 + \mathtt{y} \,\}\ \mathtt{x} \xleftarrow{\$} \{0, 1\}\ \{\, 0 \leq \mathtt{y} \wedge 0 \leq \mathtt{x} \leq 1 \mid \mathtt{x} + \mathtt{y} \,\}$$

Judgements

$$\models \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, \mathtt{Q} \mid g \,\}$$

for all initial states m satisfying classical pre-condition $\mathtt{P}$,

1. f is an upper-bound to expected value of g after running $\mathtt{C}$ $\qquad \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \leq f\, m$
2. output satisfies post-condition $\mathtt{Q}$ with certainty $\qquad m' \in \mathrm{supp}([\![\mathtt{C}]\!]_m) \Rightarrow \mathtt{Q}\, m'$

Notes

- ⋆ $f, g : \mathrm{Mem} \to [0, \infty]$ are non-negative
- ⋆ shallow embedding of classical conditions: $(\, \mathtt{P} \mid f \,)\, m \triangleq \mathtt{if}\ \mathtt{P}\, m\ \mathtt{then}\ f\, m\ \mathtt{else}\ \infty$

Examples

$$\{\, 0 \leq y \mid 1/2 + y \,\}\ x \xleftarrow{\$} \{0, 1\}\ \{\, 0 \leq y \wedge 0 \leq x \leq 1 \mid x + y \,\}$$

$$\{\, 1/2 \,\}\ x \xleftarrow{\$} \{0, 1\};\ y \xleftarrow{\$} \{0, 1\}\ \{\, x = y \,\}$$

Judgements

$$\models \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, \mathtt{Q} \mid g \,\}$$

for all initial states m satisfying classical pre-condition $\mathtt{P}$,

1. f is an upper-bound to expected value of g after running $\mathtt{C}$ $\qquad \mathbb{E}_{[\![\mathtt{C}]\!]_m}[g] \leq f\,m$
2. output satisfies post-condition $\mathtt{Q}$ with certainty $\qquad m' \in \mathsf{supp}([\![\mathtt{C}]\!]_m) \Rightarrow \mathtt{Q}\,m'$

Notes

- $f, g : \mathsf{Mem} \to [0, \infty]$ are non-negative
- shallow embedding of classical conditions: $(\,\mathtt{P} \mid f\,)\,m \triangleq \mathtt{if}\ \mathtt{P}\,m\ \mathtt{then}\ f\,m\ \mathtt{else}\ \infty$

Examples

$$\{\, 0 \leq \mathtt{y} \mid 1/2 + \mathtt{y} \,\}\ \mathtt{x} \xleftarrow{\$} \{0,1\}\ \{\, 0 \leq \mathtt{y} \wedge 0 \leq \mathtt{x} \leq 1 \mid \mathtt{x} + \mathtt{y} \,\}$$

$$\{\, 1/2 \,\}\ \mathtt{x} \xleftarrow{\$} \{0,1\};\ \mathtt{y} \xleftarrow{\$} \{0,1\}\ \{\, \mathtt{x} = \mathtt{y} \,\}$$

$$\{\, 0 \leq \mathtt{ct} \mid \mathtt{ct} + 1 \,\}\ \mathtt{x} \xleftarrow{\$} \{0,1\};\ \mathtt{while}\ \mathtt{x}\ \mathtt{do}\ \mathtt{ct} \leftarrow \mathtt{ct} + 1;\ \mathtt{x} \xleftarrow{\$} \{0,1\}\ \mathtt{od}\ \{\, \mathtt{x} = 0 \wedge 0 \leq \mathtt{ct} \mid \mathtt{ct} \,\}$$

Core rules (simplified, no procedures)

Probabilistic rule:

$$\frac{}{\vdash \{\, \mathbb{E}_{\mathtt{F}}[\lambda v.\, f[\mathtt{x}{:=}v]] \,\}\ \mathtt{x} \xleftarrow{\$} \mathtt{F}\ \{\, f \,\}}\ \text{[sample]}$$

Standard Hoare logic rules:

$$\frac{}{\vdash \{\, f \,\}\ \mathtt{skip}\ \{\, f \,\}}\ \text{[skip]}$$

$$\frac{}{\vdash \{\, f[x{:=}\mathtt{E}] \,\}\ \mathtt{x} \leftarrow \mathtt{E}\ \{\, f \,\}}\ \text{[assign]}$$

$$\frac{\vdash \{\, f \,\}\ \mathtt{C}\ \{\, h \,\} \qquad \vdash \{\, h \,\}\ \mathtt{D}\ \{\, g \,\}}{\vdash \{\, f \,\}\ \mathtt{C};\ \mathtt{D}\ \{\, g \,\}}\ \text{[seq]}$$

$$\frac{\vdash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, g \,\} \qquad \vdash \{\, \neg\mathtt{P} \mid f \,\}\ \mathtt{D}\ \{\, g \,\}}{\vdash \{\, f \,\}\ \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}\ \{\, g \,\}}\ \text{[if]}$$

$$\frac{\vdash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, f \,\}}{\vdash \{\, f \,\}\ \mathtt{while}\ \mathtt{P}\ \mathtt{do}\ \mathtt{C}\ \{\, \neg\mathtt{P} \mid f \,\}}\ \text{[while]}$$

$$\frac{\vDash f' \leq f \qquad \vdash \{\, f' \,\}\ \mathtt{C}\ \{\, g' \,\} \qquad \vDash g \leq g'}{\vdash \{\, f \,\}\ \mathtt{C}\ \{\, g \,\}}\ \text{[weaken]}$$

Core rules (simplified, no procedures)

Probabilistic rule:

$$\frac{}{\vdash \{\, \mathbb{E}_{\mathtt{F}}[\lambda v.\, f[\mathtt{x}{:=}v]] \,\}\ \mathtt{x} \xleftarrow{\$} \mathtt{F}\ \{\, f \,\}}\ \text{[sample]}$$

Standard Hoare logic rules:

$$\frac{}{\vdash \{\, f \,\}\ \mathtt{skip}\ \{\, f \,\}}\ \text{[skip]} \qquad \frac{}{\vdash \{\, f[\mathtt{x}{:=}\mathtt{E}] \,\}\ \mathtt{x} \leftarrow \mathtt{E}\ \{\, f \,\}}\ \text{[assign]}$$

$$\frac{\vdash \{\, f \,\}\ \mathtt{C}\ \{\, h \,\} \quad \vdash \{\, h \,\}\ \mathtt{D}\ \{\, g \,\}}{\vdash \{\, f \,\}\ \mathtt{C};\ \mathtt{D}\ \{\, g \,\}}\ \text{[seq]} \qquad \frac{\vdash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, g \,\} \quad \vdash \{\, \neg\mathtt{P} \mid f \,\}\ \mathtt{D}\ \{\, g \,\}}{\vdash \{\, f \,\}\ \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}\ \{\, g \,\}}\ \text{[if]}$$

$$\frac{\vdash \{\, \mathtt{P} \mid f \,\}\ \mathtt{C}\ \{\, f \,\}}{\vdash \{\, f \,\}\ \mathtt{while}\ \mathtt{P}\ \mathtt{do}\ \mathtt{C}\ \{\, \neg\mathtt{P} \mid f \,\}}\ \text{[while]} \qquad \frac{\vDash f' \le f \quad \vdash \{\, f' \,\}\ \mathtt{C}\ \{\, g' \,\} \quad \vDash g \le g'}{\vdash \{\, f \,\}\ \mathtt{C}\ \{\, g \,\}}\ \text{[weaken]}$$

Theorem (Soundness and Completeness)

$$\vdash \{\, f \,\}\ \mathtt{C}\ \{\, g \,\} \quad \Leftrightarrow \quad \vDash \{\, f \,\}\ \mathtt{C}\ \{\, g \,\}$$

A simple example

```
var ct;  // loop counter

proc rw(n)
  ct ← 0;
  while 0 < n do
    b ←$ {0, 1};
    if b then n ← n − 1 else skip;
    ct ← ct + 1
  od
  return ct;
```

avg. # coin flips reaching *n* heads

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;

  while 0 < n do

    b ←$ {0, 1};

    if b then n ← n − 1 else skip;

    ct ← ct + 1

  od

  return ct;
// res
```

avg. # coin flips reaching n heads

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;

  while 0 < n do



    b ←$ {0, 1};


    if b then n ← n − 1 else skip;

    ct ← ct + 1

  od

  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do



    b ←$ {0, 1};


    if b then n ← n − 1 else skip;

    ct ← ct + 1

  od

  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct

    b ←$ {0, 1};

    if b then n ← n − 1 else skip;

    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct


    b ←$ {0, 1};


    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct


    b ←$ {0, 1};
    // 0 < n ∧ 0 ≤ ct + 1 |
       if b then 2(n − 1) + (ct + 1) else 2n + (ct + 1)
    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching n heads

$$\frac{\vdash \{\, g \,\}\ \mathtt{C}\ \{\, f \,\} \qquad \vdash \{\, h \,\}\ \mathtt{D}\ \{\, f \,\}}{\vdash \{\, \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ g\ \mathtt{else}\ h \,\}\ \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}\ \{\, f \,\}}\ \text{[WP if]}$$

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
    // 0 < n ∧ 0 ≤ ct + 1 |
      1/2 · (2(n − 1) + (ct + 1)) + 1/2 · (2n + (ct + 1))
    b ←$ {0, 1};
    // 0 < n ∧ 0 ≤ ct + 1 |
      if b then 2(n − 1) + (ct + 1) else 2n + (ct + 1)
    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

$$\frac{\vdash \{g\}\, C\, \{f\} \qquad \vdash \{h\}\, D\, \{f\}}{\vdash \{\, \mathtt{if}\ P\ \mathtt{then}\ g\ \mathtt{else}\ h \,\}\ \mathtt{if}\ P\ \mathtt{then}\ C\ \mathtt{else}\ D\ \{f\}}\ \text{[WP if]}$$

$$\frac{}{\vdash \{\, 1/2 \cdot f[b{:=}0] + 1/2 \cdot f[b{:=}1] \,\}\ b \xleftarrow{\$} \{0,1\}\ \{f\}}\ \text{[Unif]}$$

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)

  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
    // 0 < n ∧ 0 ≤ ct + 1 |
       1/2 · (2(n − 1) + (ct + 1)) + 1/2 · (2n + (ct + 1))
    b ←$ {0, 1};
    // 0 < n ∧ 0 ≤ ct + 1 |
       if b then 2(n − 1) + (ct + 1) else 2n + (ct + 1)
    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

$$\frac{\vdash \{\, g \,\}\ \mathtt{C}\ \{\, f \,\} \qquad \vdash \{\, h \,\}\ \mathtt{D}\ \{\, f \,\}}{\vdash \{\ \mathtt{if\ P\ then}\ g\ \mathtt{else}\ h\ \}\ \mathtt{if\ P\ then\ C\ else\ D}\ \{\, f \,\}}\ \text{[WP if]}$$

$$\frac{}{\vdash \{\ 1/2 \cdot f[\mathtt{b}{:=}0] + 1/2 \cdot f[\mathtt{b}{:=}1]\ \}\ \mathtt{b} \overset{\$}{\leftarrow} \{0,1\}\ \{\, f \,\}}\ \text{[Unif]}$$

since

$$0 < \mathtt{n} \wedge 0 \le \mathtt{n} \wedge 0 \le \mathtt{ct} \ \Rightarrow\ 0 < \mathtt{n} \wedge 0 \le \mathtt{ct} + 1$$

and

$$2\mathtt{n} + \mathtt{ct} \ge 1/2 \cdot (2(\mathtt{n} - 1) + (\mathtt{ct} + 1)) + 1/2 \cdot (2\mathtt{n} + (\mathtt{ct} + 1))$$

A simple example

$$\frac{\vdash \{\, g \,\}\ \mathtt{C}\ \{\, f \,\} \qquad \vdash \{\, h \,\}\ \mathtt{D}\ \{\, f \,\}}{\vdash \{\, \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ g\ \mathtt{else}\ h \,\}\ \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D}\ \{\, f \,\}}\ \text{[WP if]}$$

$$\frac{}{\vdash \{\, 1/2 \cdot f[\mathtt{b}{:=}0] + 1/2 \cdot f[\mathtt{b}{:=}1] \,\}\ \mathtt{b} \xleftarrow{\$} \{0,1\}\ \{\, f \,\}}\ \text{[Unif]}$$

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)
  // 0 ≤ n ∧ 0 ≤ 0 | 2n + 0
  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
    // 0 < n ∧ 0 ≤ ct + 1 |
       1/2 · (2(n − 1) + (ct + 1)) + 1/2 · (2n + (ct + 1))
    b ←$ {0, 1};
    // 0 < n ∧ 0 ≤ ct + 1 |
       if b then 2(n − 1) + (ct + 1) else 2n + (ct + 1)
    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

since

$$0 < \mathtt{n} \wedge 0 \leq \mathtt{n} \wedge 0 \leq \mathtt{ct} \ \Rightarrow\ 0 < \mathtt{n} \wedge 0 \leq \mathtt{ct} + 1$$

and

$$2\mathtt{n} + \mathtt{ct} \geq 1/2 \cdot (2(\mathtt{n} - 1) + (\mathtt{ct} + 1)) + 1/2 \cdot (2\mathtt{n} + (\mathtt{ct} + 1))$$

A simple example

```
var ct; // loop counter
// 0 ≤ n | 2n
proc rw(n)
  // 0 ≤ n ∧ 0 ≤ 0 | 2n + 0
  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
    // 0 < n ∧ 0 ≤ ct + 1 |
       1/2 · (2(n − 1) + (ct + 1)) + 1/2 · (2n + (ct + 1))
    b ←$ {0, 1};
    // 0 < n ∧ 0 ≤ ct + 1 |
       if b then 2(n − 1) + (ct + 1) else 2n + (ct + 1)
    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

$$\frac{\vdash \{\, g \,\}\; \mathtt{C} \;\{\, f \,\} \qquad \vdash \{\, h \,\}\; \mathtt{D} \;\{\, f \,\}}{\vdash \{\, \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ g\ \mathtt{else}\ h \,\}\; \mathtt{if}\ \mathtt{P}\ \mathtt{then}\ \mathtt{C}\ \mathtt{else}\ \mathtt{D} \;\{\, f \,\}}\ \text{[WP if]}$$

$$\frac{}{\vdash \{\, 1/2 \cdot f[\mathtt{b}:=0] + 1/2 \cdot f[\mathtt{b}:=1] \,\}\; \mathtt{b} \xleftarrow{\$} \{0,1\} \;\{\, f \,\}}\ \text{[Unif]}$$

since

$$0 < n \wedge 0 \le n \wedge 0 \le ct \;\Rightarrow\; 0 < n \wedge 0 \le ct + 1$$

and

$$2n + ct \ge 1/2 \cdot (2(n-1) + (ct+1)) + 1/2 \cdot (2n + (ct + 1))$$

A simple example

```
var ct;  // loop counter
// 0 ≤ n | 2n
proc rw(n)
  // 0 ≤ n ∧ 0 ≤ 0 | 2n + 0
  ct ← 0;
  // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  while 0 < n do
    // 0 < n ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
    // 0 < n ∧ 0 ≤ ct + 1 |
       1/2 · (2(n − 1) + (ct + 1)) + 1/2 · (2n + (ct + 1))
    b ←$ {0, 1};
    // 0 < n ∧ 0 ≤ ct + 1 |
       if b then 2(n − 1) + (ct + 1) else 2n + (ct + 1)
    if b then n ← n − 1 else skip;
    // 0 ≤ n ∧ 0 ≤ ct + 1 | 2n + (ct + 1)
    ct ← ct + 1
    // 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  od
  // ¬(0 < n) ∧ 0 ≤ n ∧ 0 ≤ ct | 2n + ct
  // ct
  return ct;
// res
```

avg. # coin flips reaching *n* heads

```
(* { 0 <= n | 2 * n } rw { ct } *)
ehoare rw_time : Ex.rw :
  (0 <= n) `|` (2 * n)%xr ==> res%xr.
proof.
  proc.
  while ((0 <= n /\ 0 <= ct) `|` (2 * n + ct)).
  (* ct <=  !(n < 0) | I *)
  + by smt().
  (* { 0 < n | I } Loop-Body { I } *)
  + auto => &hr />.
    rewrite Ep_dbool /=.
    apply xle_cxr_r => /> *.
    have -> /=: 0 <= n{hr} - 1 by smt().
    have -> /=: 0 <= ct{hr} + 1 by smt().
    by smt().
  by auto.
```

728 mavanzin/slides/Pepr24/ex.ec 32:37 Bot   EasyCrypt script (+3)

```
Current goal (remaining: 2)

Type variables: <none>

------------------------------------------------------------------------
Context : {b : bool, n : int}

pre = (0 < n) `|` ((0 <= n /\ 0 <= ct) `|` (2 * n + ct))

(1--)  b <$ {0,1}
(2--)  if (b) {
(2.1)    n <- n - 1
(2--)  }
(3--)  ct <- ct + 1

post = (0 <= n /\ 0 <= ct) `|` (2 * n + ct)
```

425 *goals* 1:0 All   EasyCrypt goals (+2)

1

A more interesting example

```
var ct;  // cost counter
// partition array a[l ...h] with pivot a[h]
proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p ←$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

// select k-th largest element from array a
proc qselect(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition(a, l, h);
    if i = k then l ← i; h ← i          // exit loop
    elseif i < k then l ← i + 1 // descent right
    else h ← i − 1                      // descent left
  return a[k]
```

randomised quickselect

$\mathtt{cost}(l, h) =$

$(h - l)$ *// cost partition*

$+ \sum_{i=l}^{k-1} \frac{1}{h-l+1} \mathtt{cost}(i+1, h)$ *// right descents*

$+ \sum_{i=k}^{h} \frac{1}{h-l+1} \mathtt{cost}(l, i-1)$ *// left descents*

$\Rightarrow$ solution has upper-bound $4(h - l)$

A more interesting example

```
var ct;  // cost counter
// partition array a[l ...h] with pivot a[h]
proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p <-$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

// select k-th largest element from array a
proc qselect(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition(a, l, h);
    if i = k then l ← i; h ← i          // exit loop
    elseif i < k then l ← i + 1 // descent right
    else h ← i − 1                      // descent left
  return a[k]
```

randomised quickselect

$$\begin{aligned} \mathtt{cost}(l, h) = {} & \\ & (h - l) \quad \textit{// cost partition} \\ & + \sum_{i=l}^{k-1} \frac{1}{h-l+1} \mathtt{cost}(i+1, h) \quad \textit{// right descents} \\ & + \sum_{i=k}^{h} \frac{1}{h-l+1} \mathtt{cost}(l, i-1) \quad \textit{// left descents} \end{aligned}$$

⇒ solution has upper-bound $4(h - l)$

relies on functional correctness of partitioning

A more interesting example

```
var ct;  // cost counter

// partition array a[l ...h] with pivot a[h]
proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p ←$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

// select k-th largest element from array a
proc qselect(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition(a, l, h);
    if i = k then l ← i; h ← i              // exit loop
    elseif i < k then l ← i + 1 // descent right
    else h ← i − 1                          // descent left
  return a[k]
```

randomised quickselect

$$\begin{aligned}&\mathtt{cost}(l,h) = \\ &\quad (h - l) \quad \textit{// cost partition} \\ &\quad + \textstyle\sum_{i=l}^{k-1} \frac{1}{h-l+1} \mathtt{cost}(i+1, h) \quad \textit{// right descents} \\ &\quad + \textstyle\sum_{i=k}^{h} \frac{1}{h-l+1} \mathtt{cost}(l, i-1) \quad \textit{// left descents}\end{aligned}$$

$\Rightarrow$ solution has upper-bound $4(h-l)$

relies on functional correctness of partitioning

- ⋆ tedious to carry through eHL proof
- ⋆ program abstraction facilitate analysis
- ⋆ formalised via pRHL

A more interesting example

```
var ct; // cost counter
// partition array a[l ...h] with pivot a[h]
proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p ←$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

// select k-th largest element from array a
proc qselect(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition(a, l, h);
    if i = k then l ← i; h ← i          // exit loop
    elseif i < k then l ← i + 1 // descent right
    else h ← i − 1                        // descent left
  return a[k]
```

randomised quickselect

```
var ct;
proc rpartition_abs(l, h)
  ct ← ct + (h − l);
  i ←$ unif(l, h);
  return i

proc qselect_abs(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition_abs(l, h);
    if i = k then l ← i; h ← i
    elseif i < k then l ← i + 1
    else h ← i − 1
  return a[k]
```

abstraction focusing on counter

A more interesting example

```
var ct;  // cost counter
// partition array a[l ...h] with pivot a[h]
proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p ←$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

// select k-th largest element from array a
proc qselect(a, k)
  (ct, l, h) ← (
  while l < h d
    i ← rpartition(a, l, h);
    if i = k then l ← i; h ← i        // exit loop
    elseif i < k then l ← i + 1 // descent right
    else h ← i − 1                   // descent left
  return a[k]
```

randomised quickselect

```
var ct;
proc rpartition_abs(l, h)
  ct ← ct + (h − l);
  i ←$ unif(l, h);
  return i

proc qselect_abs(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition_abs(l, h);
    if i = k then l ← i; h ← i
    elseif i < k then l ← i + 1
    else h ← i − 1
  return a[k]
```

abstraction focusing on counter

$$\{\,\text{unique}\ a\,\}\ \texttt{rpartition}(a, l, h) \sim \texttt{rpartition\_abs}(l, h)\ \{\, ct^{\langle 1 \rangle} \leq ct^{\langle 2 \rangle}\,\}$$

pRHL

A more interesting example

```
var ct; // cost counter
// partition array a[l ...h] with pivot a[h]
proc partition(a, l, h)
  (p, i) ← (a[h], l − 1);
  for j = l to h − 1 do
    if a[j] < p then i++; swap(a, i, j)
    ct++;
  i++; swap(a, i, h);
  return i

proc rpartition(a, l, h)
  p ←$ unif(l, h);
  swap(a, p, h);
  i ← partition(a, l, h);
  return i

// select k-th largest element from array a
proc qselect(a, k)
  (ct, l, h) ← (
  while l < h d
    i ← rpartition(a, l, h);
    if i = k then l ← i; h ← i          // exit loop
    elseif i < k then l ← i + 1        // descent right
    else h ← i − 1                     // descent left
  return a[k]
```

randomised quickselect

```
var ct;
proc rpartition_abs(l, h)
  ct ← ct + (h − l);
  i ←$ unif(l, h);
  return i

proc qselect_abs(a, k)
  (ct, l, h) ← (0, 0, size(a) − 1);
  while l < h do
    i ← rpartition_abs(l, h);
    if i = k then l ← i; h ← i
    elseif i < k then l ← i + 1
    else h ← i − 1
  return a[k]
```

abstraction focusing on counter

$$\{ \text{unique } a \} \; \texttt{rpartition}(a, l, h) \sim \texttt{rpartition\_abs}(l, h) \; \{ ct^{\langle 1 \rangle} \leq ct^{\langle 2 \rangle} \}$$

pRHL

⋆ ~380loc HL/pRHL + ~60loc eHL

Reasoning with $\mathtt{pRHL}$ within $\mathtt{eHL}$

$$\models \{\, \mathtt{P} \,\}\, \mathtt{C}_1 \sim \mathtt{C}_2 \,\{\, \mathtt{Q} \,\}$$

for all pairs of initial states m_1, m_2 related by $\mathtt{P}$, output distributions of $\mathtt{C}_1, \mathtt{C}_2$ coupled by $\mathtt{Q}$

Reasoning with $\mathtt{pRHL}$ within $\mathtt{eHL}$

$$\models \{\, \mathtt{P} \,\}\, \mathtt{C}_1 \sim \mathtt{C}_2 \,\{\, \mathtt{Q} \,\}$$

for all pairs of initial states m_1, m_2 related by $\mathtt{P}$, output distributions of $\mathtt{C}_1, \mathtt{C}_2$ coupled by $\mathtt{Q}$

$$\frac{\vdash \{\, f' \,\}\, \mathtt{C}' \,\{\, g' \,\} \qquad \models \{\, \mathtt{P} \,\}\, \mathtt{C}' \sim \mathtt{C} \,\{\, \mathtt{Q} \,\} \qquad \begin{array}{c} \forall m \exists m'.\, \mathtt{P}\, m'\, m \wedge f'\, m' \leq f\, m \\ \forall m' m.\, \mathtt{Q}\, m'\, m \Rightarrow g\, m \leq g'\, m' \end{array}}{\vdash \{\, f \,\}\, \mathtt{C} \,\{\, g \,\}}\ [\text{prhl}]$$

Verification of Dilithium PQC signature scheme [Barbosa et al., 23]

```
// logged rejection sampling
proc rsample()
  var t, r;
  t ← false;
  while ¬t do
    r ←$ sample();
    log ← r :: log;
    t ← test(r);

// oracle
proc orcl()
  var r;
  c ← c + 1;
  rsample();
  if c = N then
    r* ←$ sample();
    bad ← r* ∈ log;

// adversarial code
proc game()
  bad ← false;
  c ← 0;
  log ← [];
  _ ← A_orcl();
  return bad
```

Stage of *Dilithium security proof in EasyCrypt* (simplified)

Manuel Barbosa et al. *"Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium"*. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V*, pp. 358–389, 2023.

Verification of Dilithium PQC signature scheme [Barbosa et al., 23]

```
// logged rejection sampling
proc rsample()
  var t, r;
  t ← false;
  while ¬t do
    r ←$ sample();
    log ← r :: log;
    t ← test(r);

// oracle
proc orcl()
  var r;
  c ← c + 1;
  rsample();
  if c = N then
    r* ←$ sample();
    bad ← r* ∈ log;

// adversarial code
proc game()
  bad ← false;
  c ← 0;
  log ← [];
  _ ← A_orcl();
  return bad
```

Stage of *Dilithium security proof in EasyCrypt* (simplified)

$$\textbf{lemma}\ \texttt{mu\_mem} : \forall \texttt{d} : a\ \textbf{distr}, \texttt{lst} : a\ \textbf{list}, \epsilon : \textbf{real}, (\forall \texttt{x} : a,\ \texttt{d}\,\texttt{x} \leq \epsilon) \Rightarrow \Pr[\texttt{d} : \lambda \texttt{x}.\ \texttt{x} \in \texttt{lst}] \leq \epsilon \cdot |\texttt{lst}|$$

Manuel Barbosa et al. *"Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium"*. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V*, pp. 358–389, 2023.

Verification of Dilithium PQC signature scheme [Barbosa et al., 23]

```
// logged rejection sampling
proc rsample()
  var t, r;
  t ← false;
  while ¬t do
    r <$ sample();
    log ← r :: log;
    t ← test(r);

// oracle
proc orcl()
  var r;
  c ← c + 1;
  rsample();
  if c = N then
    r* <$ sample();
    bad ← r* ∈ log;

// adversarial code
proc game()
  bad ← false;
  c ← 0;
  log ← [];
  _ ← A_orcl();
  return bad
```

Stage of *Dilithium security proof in EasyCrypt* (simplified)

$$\textbf{lemma}\ \texttt{mu\_mem} : \forall \texttt{d} : a\ \textbf{distr}, \texttt{lst} : a\ \textbf{list}, \epsilon : \textbf{real}, (\forall \texttt{x} : a,\ \texttt{d}\,\texttt{x} \leq \epsilon) \Rightarrow \Pr[\texttt{d} : \lambda \texttt{x}.\ \texttt{x} \in \texttt{lst}] \leq \epsilon \cdot |\texttt{lst}|$$

difficult to formalize in EasyCrypt prior to eHL:

- ⋆ size of log potentially infinite, but expected size bounded linearly
- ⋆ requires approximation of rsample so that (worst-case) size becomes finite
- ⋆ leads to over-approximated bound

Manuel Barbosa et al. *"Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium"*. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V*, pp. 358–389, 2023.

Adversary rule in eHL

$$\frac{\forall \mathtt{orcl} \in \mathtt{Orcls},\ \vdash \{f\}\ \mathtt{orcl}\ \{f\} \qquad \mathsf{Vars}\, f \subseteq \mathsf{Vars} \setminus \mathsf{Writeable}_{\mathcal{A}}}{\vdash \{f\}\ \mathcal{A}_{\mathtt{Orcls}}\ \{f\}}\ [\mathrm{adv}]$$

Adversary rule in eHL

$$\frac{\forall \mathtt{orcl} \in \mathtt{Orcls},\ \vdash \{f\}\ \mathtt{orcl}\ \{f\} \qquad \mathsf{Vars}\, f \subseteq \mathsf{Vars} \setminus \mathsf{Writeable}_{\mathcal{A}}}{\vdash \{f\}\ \mathcal{A}_{\mathtt{Orcls}}\ \{f\}}\ [\mathsf{adv}]$$

```
// 1/δ + |log|
proc rsample()
  var t, r;
  t ← false;
  while ¬t do
    r ←$ sample();
    log ← r :: log;
    t ← test(r);
// |log|
```

```
// φ | if N ≤ c then bad else ε(|log| + (N−c)/δ)
proc orcl()
  var r;
  c ← c + 1;
  rsample();
  if c = N then
    r* ←$ sample();
    bad ← r* ∈ log;
// φ | if N ≤ c then bad else ε(|log| + (N−c)/δ)
```

```
// ε · N/δ
proc game()
  bad ← false;
  c ← 0;
  log ← [];
  // φ | if N ≤ c then bad else ε(|log| + (N−c)/δ)
  _ ← A_orcl();
  // φ | if N ≤ c then bad else ε(|log| + (N−c)/δ)
  return bad
// res
```

$\phi \triangleq \mathtt{bad} \Rightarrow \mathrm{N} \leq \mathtt{c}, \quad \delta \triangleq \Pr[\mathtt{sample} : \mathtt{test}] > 0, \quad \epsilon \triangleq \sup_{\mathsf{v} \in \mathsf{Val}} \Pr[\mathtt{sample} : \mathbf{1}_{\mathsf{v}}]$

Some words on modularity & integration

1. logical variables + bookkeeping rules

$$\{\, \mathtt{l} \le \mathtt{h} \mid \frac{1}{\mathtt{h}-\mathtt{l}+1} \sum_{\mathtt{i}=\mathtt{l}}^{\mathtt{h}} \mathtt{g}(\mathtt{i}) \,\}\ \mathtt{rpartition\_abs}(\mathtt{l},\mathtt{h})\ \{\, \mathtt{g}(\mathtt{res}) \,\}$$

   - effectively turn triples into schemas
   - necessary to retain completeness in the presence of procedures

Some words on modularity & integration

1. logical variables + bookkeeping rules

$$\{\, \mathtt{l} \leq \mathtt{h} \mid \frac{1}{\mathtt{h}-\mathtt{l}+1} \sum_{\mathtt{i}=\mathtt{l}}^{\mathtt{h}} \mathtt{g}(\mathtt{i}) \,\}\ \mathtt{rpartition\_abs}(\mathtt{l},\mathtt{h})\ \{\, \mathtt{g}(\mathtt{res}) \,\}$$

   - effectively turn triples into schemas
   - necessary to retain completeness in the presence of procedures

2. frame rule

$$\frac{\{\, \frac{1}{\delta} + |\mathtt{log}| \,\}\ \mathtt{rsample}()\ \{\, |\mathtt{log}| \,\}}{\{\, \mathtt{if}\ \mathrm{N} < \mathtt{c}\ \mathtt{then}\ \mathtt{bad}\ \mathtt{else}\ \epsilon(\frac{1}{\delta} + |\mathtt{log}| + \frac{\mathrm{N}-\mathtt{c}}{\delta}) \,\}\ \mathtt{rsample}()\ \{\, \mathtt{if}\ \mathrm{N} < \mathtt{c}\ \mathtt{then}\ \mathtt{bad}\ \mathtt{else}\ \epsilon(|\mathtt{log}| + \frac{\mathrm{N}-\mathtt{c}}{\delta}) \,\}}$$

   - context arbitrary concave + non-decreasing function (often, automatically discharged)
   - should not depend memory modified by analysed code (checked syntactically)

Some words on modularity & integration

1. logical variables + bookkeeping rules

$$\{\, \mathtt{l} \leq \mathtt{h} \mid \frac{1}{\mathtt{h}-\mathtt{l}+1} \sum_{\mathtt{i}=\mathtt{l}}^{\mathtt{h}} \mathtt{g}(\mathtt{i}) \,\}\ \mathtt{rpartition\_abs}(\mathtt{l}, \mathtt{h})\ \{\, \mathtt{g}(\mathtt{res}) \,\}$$

   - effectively turn triples into schemas
   - necessary to retain completeness in the presence of procedures

2. frame rule

$$\frac{\{\, \frac{1}{\delta} + |\mathtt{log}| \,\}\ \mathtt{rsample}()\ \{\, |\mathtt{log}| \,\}}{\{\, \mathbf{if}\ \mathrm{N} < \mathtt{c}\ \mathbf{then}\ \mathtt{bad}\ \mathbf{else}\ \epsilon(\frac{1}{\delta} + |\mathtt{log}| + \frac{\mathrm{N}-\mathtt{c}}{\delta}) \,\}\ \mathtt{rsample}()\ \{\, \mathbf{if}\ \mathrm{N} < \mathtt{c}\ \mathbf{then}\ \mathtt{bad}\ \mathbf{else}\ \epsilon(|\mathtt{log}| + \frac{\mathrm{N}-\mathtt{c}}{\delta}) \,\}}$$

   - context arbitrary concave + non-decreasing function (often, automatically discharged)
   - should not depend memory modified by analysed code (checked syntactically)

3. automation in `EasyCrypt` through derived rules through WP-style reasoning, except:
   - `while` loops: specify invariant
   - procedure `calls`: specify lemma (+ optional frame context)
   - final weakening steps

Conclusion

Aim

usable and expressive logic with applications to avg. case complexity and security proofs

Our solution

- ⋆ combination of expectation based logic (eHL) with relational reasoning (pRHL)
- ⋆ implementation within EasyCrypt; derived rules facilitate bottom-up reasoning
- ⋆ development driven by highly non-trivial example

Conclusion

Aim

usable and expressive logic with applications to avg. case complexity and security proofs

Our solution

- ⋆ combination of expectation based logic (eHL) with relational reasoning (pRHL)
- ⋆ implementation within EasyCrypt; derived rules facilitate bottom-up reasoning
- ⋆ development driven by highly non-trivial example

Future Work

- ⋆ lower-bounds $\equiv$ mixed-sign expectations
- ⋆ more applications
- ⋆ expectation based relational logic (eRHL)
- ⋆ higher-order extensions
- ⋆ ...

Thanks!