CS 591: Formal Methods in Security and Privacy
RHL and probabilistic computations

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu
From the previous classes
NonInterference

In symbols, c is noninterferent if and only if for every $m_1 \sim_{\text{low}} m_2$:

1) $\{c\}_{m_1} = \bot$ iff $\{c\}_{m_2} = \bot$

2) $\{c\}_{m_1} = m_1'$ and $\{c\}_{m_2} = m_2'$ implies $m_1' \sim_{\text{low}} m_2'$

\begin{align*}
U_1 & \quad \text{private} & C & \quad \text{private} & O_1 \\
V & \quad \text{public} & & & \\
U_2 & \quad \text{private} & C & \quad \text{private} & O_2 \\
V & \quad \text{public} & & & W
\end{align*}
Relational Hoare Logic - RHL

Program₁ \sim \text{ Postcondition}

\begin{align*}
\quad & c₁ \sim c₂ : P \Rightarrow Q \\
\quad & \text{Precondition} \\
\text{Postcondition} & \quad \text{(a logical formula)}
\end{align*}
Some Rules of Relational Hoare Logic

\[ \vdash \text{skip} \sim \text{skip} : P \rightarrow P \]

\[ \vdash \text{abort} \sim \text{abort} : \text{true} \rightarrow \text{false} \]

\[ \vdash x_1 := e_1 \sim x_2 := e_2 : P[e_1<1>/x_1<1>, e_2<2>/x_2<2>] \rightarrow P \]

\[ \vdash c_1 \sim c_2 : P \rightarrow R \quad \vdash c_1' \sim c_2' : R \rightarrow S \]

\[ \vdash c_1 ; c_1' \sim c_2 ; c_2' : P \rightarrow S \]

\[ P \rightarrow S \quad \vdash c_1 \sim c_2 : S \rightarrow R \quad R \rightarrow Q \]

\[ \vdash c_1 \sim c_2 : P \rightarrow Q \]
Some Rules of Relational Hoare Logic

\[ \vdash c_1 \sim c_2 : e_1 < 1 > \land P \Rightarrow Q \quad P \Rightarrow e_1 < 1 > = e_2 < 2 > \]

\[ \vdash c_1' \sim c_2' : \neg e_1 < 1 > \land P \Rightarrow Q \]

\[ \vdash \text{if } e_1 \text{ then } c_1 \text{ else } c_1' : P \Rightarrow Q \]
\[ \vdash \text{if } e_2 \text{ then } c_2 \text{ else } c_2' \]

\[ \vdash c_1 \sim c_2 : e_1 < 1 > \land P \Rightarrow P \quad P \Rightarrow e_1 < 1 > = e_2 < 2 > \]

\[ \vdash \text{while } e_1 \text{ do } c_1 \]
\[ \vdash \text{while } e_2 \text{ do } c_2 : P \Rightarrow P \land \neg e_1 < 1 > \]
One-sided Rules

\[\vdash c_1 \sim c_2 : e<1> \land P \Rightarrow Q \quad \vdash c_1' \sim c_2 : \neg e<1> \land P \Rightarrow Q\]

\[\vdash \text{if } e \text{ then } ^\sim c_1 \text{ else } c_1' : P \Rightarrow Q \quad \vdash \text{if } e \text{ then } c_2 \text{ else } c_2' : P \Rightarrow Q\]

\[\vdash c_1 \sim c_2 : e<2> \land P \Rightarrow Q \quad \vdash c_1 \sim c_2' : \neg e<2> \land P \Rightarrow Q\]

\[\vdash \text{if } e \text{ then } ^\sim c_1 \text{ else } c_2' : P \Rightarrow Q\]
How can we prove this?

s1:public
s2:private
r:private
i:public

proc Compare (s1:list[n] bool,s2:list[n] bool)
  i:=0;
  r:=0;
  while i<n do
    if not(s1[i]=s2[i]) then
      r:=1
    i:=i+1
  : n>0 /\ =low ⇒ =low
Today: more on RHL and probabilistic computations
What do we do if our two programs have different forms? There are three pairs of one-sided rules.
Assignment — left

\[ \vdash x := e \sim \text{skip}: \quad P[e<1>/x<1>] \implies P \]
Assignment — right

\[ \vdash \text{skip} \sim x := e : \]
\[ P[e<2>/x<2>] \implies P \]

Also pair of one-sided rules for while — we’ll ignore for now
Rules of Relational Hoare Logic

Program Equivalence Rule

\[
\models P : c_1' \equiv c_1 \\
\models P : c_2' \equiv c_2 \quad \vdash c_1' \sim c_2' : P \Rightarrow Q \\
\vdash c_1 \sim c_2 : P \Rightarrow Q
\]

\[
\models P : c_1 \equiv c_2 \text{ means } \{c_1\}_m = \{c_2\}_m \\
\text{for all } m \text{ such that } P(m)
\]
Rules of Relational Hoare Logic
Program Equivalences

\[ \vdash P : \text{skip}; c \equiv c \]

\[ \vdash P : c; \text{skip} \equiv c \]

\[ \vdash P : (c_1; c_2); c_3 \equiv c_1; (c_2; c_3) \]

...
We can combine the Composition and Program Equivalence Rules to split commands where we like:

\[ \vdash C_1 ; C_2 \sim C_1' : P \Rightarrow R \]
\[ \vdash C_3 \sim C_2' ; C_3' : R \Rightarrow Q \]
\[ \vdash C_1 ; C_2 ; C_3 \sim C_1' ; C_2' ; C_3' : P \Rightarrow Q \]
Rules of Relational Hoare Logic
Combining Composition and Equivalence

\[ \vdash c_1 \sim \text{skip}: P \Rightarrow R \]

\[ \vdash c_2 \sim c_1': R \Rightarrow Q \]

\[ \vdash c_1; c_2 \sim \text{skip}; c_1': P \Rightarrow Q \]

\[ \vdash c_1; c_2 \sim c_1': P \Rightarrow Q \]
Rules of Relational Hoare Logic
Combining Composition and Equivalence

\[ \vdash c_1 \sim c_1': \quad P \Rightarrow R \]

\[ \vdash c_2 \sim \text{skip}: \quad R \Rightarrow Q \]

\[ \vdash c_1; c_2 \sim c_1'; \text{skip}: \quad P \Rightarrow Q \]

\[ \vdash c_1; c_2 \sim c_1': \quad P \Rightarrow Q \]
Soundness

If we can derive $\vdash c_1 \sim c_2 : P \Rightarrow Q$ through the rules of the logic, then the quadruple $c_1 \sim c_2 : P \Rightarrow Q$ is valid.
Validity of Hoare quadruple

We say that the quadruple $\mathbf{c}_1 \sim \mathbf{c}_2 : \mathbf{P} \Rightarrow \mathbf{Q}$ is valid if and only if for every pair of memories $m_1, m_2$ such that $\mathbf{P}(m_1, m_2)$ we have:

1) $\{\mathbf{c}_1\}_{m_1} = \bot$ iff $\{\mathbf{c}_2\}_{m_2} = \bot$

2) $\{\mathbf{c}_1\}_{m_1} = m_1'$ and $\{\mathbf{c}_2\}_{m_2} = m_2'$ implies $\mathbf{Q}(m_1', m_2')$. 
Validity of Hoare quadruple

We say that the quadruple $c_1 \sim c_2 : P \Rightarrow Q$ is valid if and only if for every pair of memories $m_1, m_2$ such that $P(m_1, m_2)$ we have:

1) $\{c_1\}_{m_1} = \perp$ iff $\{c_2\}_{m_2} = \perp$

2) $\{c_1\}_{m_1} = m_1'$ and $\{c_2\}_{m_2} = m_2'$ implies $Q(m_1', m_2')$.

How do we check this?
Relative Completeness

If a quadruple $c_1 \sim c_2 : P \Rightarrow Q$ is valid, and we have an oracle to derive all the true statements of the form $P \Rightarrow S$ and of the form $R \Rightarrow Q$, then we can derive $\vdash c_1 \sim c_2 : P \Rightarrow Q$ through the rules of the logic.
Soundness and completeness with respect to Hoare Logic

\[ \vdash_{\text{RHL}} c_1 \sim c_2 : P \Rightarrow Q \]

iff

\[ \vdash_{\text{HL}} c_1; c_2 : P \Rightarrow Q \]
Soundness and completeness with respect to Hoare Logic

\[ \vdash_{\text{RHL}} C_1 \sim C_2 : P \Rightarrow Q \]

iff

\[ \vdash_{\text{HL}} C_1 ; C_2 : P \Rightarrow Q \]

Under the assumption that we can partition the memory adequately, and that we have termination.
Possible projects

In Easycrypt
• Look at how to guarantee trace-based noninterference.
• Look at how to guarantee side-channel free noninterference.
• Look at the relations between self-composition and relational logic.

Not related to Easycrypt
• Look at type systems for non-interference.
• Look at other methods for relational reasoning
• Look at declassification
Probabilistic Language
An example

OneTimePad(m : private msg) : public msg
key :=$ Uniform({0,1}^n);
cipher := msg xor key;
return cipher

Learning a ciphertext does not change any a priori knowledge about the likelihood of messages.
Probabilistic While (PWhile)

c ::= abort
  | skip
  | x := e
  | x := $ d
  | c ; c
  | if e then c else c
  | while e do c

$d_1, d_2, \ldots$  probabilistic expressions
Probabilistic Expressions

We extend the language with expression describing probability distributions.

\[ d ::= f(e_1, ..., e_n, d_1, ..., d_k) \]

Where \( f \) is a distribution declaration.

Some expression examples

- \( \text{uniform}({0,1}^n) \)
- \( \text{gaussian}(k, \sigma) \)
- \( \text{laplace}(k, b) \)
Semantics of Probabilistic Expressions

We would like to define it on the structure:

$$\{f(e_1, \ldots, e_n, d_1, \ldots, d_k)\}_m = \{f\}(\{e_1\}_m, \ldots, \{e_n\}_m, \{d_1\}_m, \ldots, \{d_k\}_m)$$

but is the result just a value?
A discrete subdistribution over a set $A$ is a function $\mu : A \rightarrow [0, 1]$ such that the mass of $\mu$, $|\mu| = \sum_{a \in A} \mu(a)$ verifies $|\mu| \leq 1$.

The support of a discrete subdistribution $\mu$, $\text{supp}(\mu) = \{a \in A | \mu(a) > 0\}$ is necessarily countable, i.e. finite or countably infinite.

We will denote the set of sub-distributions over $A$ by $D(A)$, and say that $\mu$ is of type $D(A)$ denoted $\mu:D(A)$ if $\mu \in D(A)$. 
Probabilistic Subdistributions

We call a subdistribution with mass exactly 1, a distribution.

We define the probability of an event $E \subseteq A$ with respect to the subdistribution $\mu: D(A)$ as

$$
\mathbb{P}_\mu[E] = \sum_{a \in E} \mu(a)
$$
Probabilistic Subdistributions

Let’s consider \( \mu \in \mathcal{D}(A) \), and \( E \subseteq A \), we have the following properties:

\[
\mathbb{P}_\mu[\emptyset] = 0
\]
\[
\mathbb{P}_\mu[A] \leq 1
\]
\[
0 \leq \mathbb{P}_\mu[E] \leq 1
\]

\( E \subseteq F \subseteq A \) implies \( \mathbb{P}_\mu[E] \leq \mathbb{P}_\mu[F] \)

\( E \subseteq A \) and \( F \subseteq A \) implies \( \mathbb{P}_\mu[E \cup F] \leq \mathbb{P}_\mu[E] + \mathbb{P}_\mu[F] - \mathbb{P}_\mu[E \cap F] \)

We will denote by \( \mathcal{O} \) the subdistribution \( \mu \) defined as constant 0.
Operations over Probabilistic Subdistributions

Let’s consider an arbitrary \( a \in A \), we will often use the distribution \( \text{unit}(a) \) defined as:

\[
\mathbb{P}_{\text{unit}(a)}[\{b\}] = \begin{cases} 
1 & \text{if } a = b \\
0 & \text{otherwise}
\end{cases}
\]

We can think about \( \text{unit} \) as a function of type \( \text{unit}:A \rightarrow D(A) \)
Operations over Probabilistic Subdistributions

Let’s consider a distribution $\mu \in D(A)$, and a function $M : A \rightarrow D(B)$ then we can define their composition by means of an expression $\text{let } a = \mu \text{ in } M \ a$ defined as:

$$\mathbb{P}\left[\text{let } a = \mu \text{ in } M \ a\right][E] = \sum_{a \in \text{supp}(\mu)} \mathbb{P}_\mu[\{a\}] \cdot \mathbb{P}_{(Ma)}[E]$$
Semantics of Probabilistic Expressions - revisited

We would like to define it on the structure:

\[ \{ f(e_1, ..., e_n, d_1, ..., d_k) \}_m = \{ f \}(\{ e_1 \}_m, ..., \{ e_n \}_m, \{ d_1 \}_m, ..., \{ d_k \}_m) \]

With input a memory \( m \) and output a subdistribution \( \mu \in D(A) \) over the corresponding type \( A \). E.g.

\[ \{ \text{uniform} (\{ 0, 1 \}^n) \}_m \in D(\{ 0, 1 \}^n) \]

\[ \{ \text{gaussian} (k, \sigma) \}_m \in D(\text{Real}) \]
Semantics of PWhile Commands

What is the meaning of the following command?

\[
k := \$ \text{uniform}({0, 1}^n); \quad z := x \mod k;
\]
Semantics of PWhile Commands

What is the meaning of the following command?

\[ k := \$ \text{ uniform}(\{0,1\}^n); \ z := x \mod k; \]

We can give the semantics as a function between command, memories and subdistributions over memories.

\[ \text{Cmd} \times \text{Mem} \to D(\text{Mem}) \]

We will denote this relation as:

\[ \{c\}_{m=\mu} \]
Semantics of Commands

This is defined on the structure of commands:
Semantics of Commands

This is defined on the structure of commands:

\[ \{ \text{abort} \}_m = 0 \]
Semantics of Commands

This is defined on the structure of commands:

\[ \{\text{abort}\}_m = 0 \]

\[ \{\text{skip}\}_m = \text{unit}(m) \]
Semantics of Commands

This is defined on the structure of commands:

\[
\begin{align*}
\{\text{abort}\}_m &= \emptyset \\
\{\text{skip}\}_m &= \text{unit}(m) \\
\{x:=e\}_m &= \text{unit}(m[x\leftarrow\{e\}_m])
\end{align*}
\]
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_m = 0
\]

\[
\{\text{skip}\}_m = \text{unit}(m)
\]

\[
\{x := e\}_m = \text{unit}(m[x \leftarrow \{e\}_m])
\]

\[
\{c; c'\}_m = \text{let } m' = \{c\}_m \text{ in } \{c'\}_m
\]
Semantics of Commands

This is defined on the structure of commands:

\[
\begin{align*}
\{\text{abort}\}_m &= \emptyset \\
\{\text{skip}\}_m &= \text{unit}(m) \\
\{x := e\}_m &= \text{unit}(m[x \leftarrow \{e\}_m]) \\
\{c; c'\}_m &= \text{let } m' = \{c\}_m \text{ in } \{c'\}_m \\
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_m &= \{c_t\}_m \text{ if } \{e\}_m = \text{true}
\end{align*}
\]
Semantics of Commands

This is defined on the structure of commands:

\[ \{\text{abort}\}_m = \emptyset \]

\[ \{\text{skip}\}_m = \text{unit}(m) \]

\[ \{x := e\}_m = \text{unit}(m[x \leftarrow \{e\}_m]) \]

\[ \{c; c'\}_m = \text{let } m' = \{c\}_m \text{ in } \{c'\}_m \]

\[ \{\text{if } e \text{ then } c_t \text{ else } c_f\}_m = \{c_t\}_m \text{ if } \{e\}_m = \text{true} \]

\[ \{\text{if } e \text{ then } c_t \text{ else } c_f\}_m = \{c_f\}_m \text{ if } \{e\}_m = \text{false} \]
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_{m} = \text{O}
\]

\[
\{\text{skip}\}_{m} = \text{unit}(m)
\]

\[
\{x:=e\}_{m} = \text{unit}(m[x\leftarrow\{e\}_{m}])
\]

\[
\{x:=$ d\}_{m} = \text{let } a=\{d\}_{m} \text{ in } \text{unit}(m[x\leftarrow a])
\]

\[
\{c;c'\}_{m} = \text{let } m'=\{c\}_{m} \text{ in } \{c'\}_{m'}
\]

\[
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_{m} = \{c_t\}_{m} \text{ if } \{e\}_{m}=\text{true}
\]

\[
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_{m} = \{c_f\}_{m} \text{ if } \{e\}_{m}=\text{false}
\]
Semantics of While

What about while

How did we handle the deterministic case?
Semantics of While

What about while

\{ \text{while } e \text{ do } c \}_m = ???

How did we handle the deterministic case?
Semantics of While

We defined it as

\[ \{\text{while } e \text{ do } c\}_m = \sup_{n \in \text{Nat}} \mu_n \]

Where

\[ \mu_n = \]

let \( m' = \{\text{while } n \text{ e do } c\}_m \text{ in } \{\text{if } e \text{ then abort}\}_m' \]
Semantics of While

We defined it as

\[ \{ \text{while } e \text{ do } c \}_m = \sup_{n \in \text{Nat}} \mu_n \]

Where

\[ \mu_n = \text{let } m' = \{ \text{while}_n e \text{ do } c \}_m \text{ in } \{ \text{if } e \text{ then abort} \}_m' \]

Is this well defined?
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{while } e \text{ do } c\}_m = \sup_{n \in \text{Nat}} \mu_n \\
\mu_n = \text{let } m' = \{(\text{while } e \text{ do } c)\}_m \text{ in } \{\text{if } e \text{ then abort}\}_m,
\]
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_m = \emptyset
\]

\[
\{\text{while } e \text{ do } c\}_m = \sup_{n \in \text{Nat}} \mu_n
\]

\[
\mu_n = \text{let } m' = \{(\text{while } e \text{ do } c)\}_m \text{ in } \{\text{if } e \text{ then abort}\}_m
\]
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_m = 0
\]

\[
\{\text{skip}\}_m = \text{unit}(m)
\]

\[
\{\text{while } e \text{ do } c\}_m = \sup_{n \in \text{Nat}} \mu_n
\]

\[
\mu_n = \text{let } m' = \{\text{while } n \text{ e do } c\}_m \text{ in } \{\text{if e then abort}\}_{m'}
\]
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_m = O
\]

\[
\{\text{skip}\}_m = \text{unit}(m)
\]

\[
\{x:=e\}_m = \text{unit}(m[x\leftarrow\{e\}_m])
\]

\[
\{\text{while } e \text{ do } c\}_m = \sup_{n\in\text{Nat}} \mu_n
\]

\[
\mu_n=\text{let } m'='\{\text{while } n \text{ e do } c\}_m \text{ in } \{\text{if } e \text{ then } \text{abort}\}_m',
\]
Semantics of Commands

This is defined on the structure of commands:

\[ \{\text{abort}\}_m = \emptyset \]

\[ \{\text{skip}\}_m = \text{unit}(m) \]

\[ \{x:=e\}_m = \text{unit}(m[x\leftarrow\{e\}_m]) \]

\[ \{c;c'\}_m = \text{let } m' = \{c\}_m \text{ in } \{c'\}_m', \]

\[ \{\text{while } e \text{ do } c\}_m = \sup_{n\in\text{Nat}} \mu_n \]

\[ \mu_n = \text{let } m' = \{(\text{while}_n e \text{ do } c)\}_m \text{ in } \{\text{if } e \text{ then abort}\}_m', \]
Semantics of Commands

This is defined on the structure of commands:

\{\texttt{abort}\}_m = \emptyset

\{\texttt{skip}\}_m = \text{unit}(m)

\{x:=e\}_m = \text{unit}(m[x\leftarrow\{e\}_m])

\{c;c'\}_m = \text{let } m' = \{c\}_m \text{ in } \{c'\}_m'

\{\texttt{if } e \texttt{ then } c_t \texttt{ else } c_f\}_m = \{c_t\}_m \text{ if } \{e\}_m = \text{true}

\{\texttt{while } e \texttt{ do } c\}_m = \sup_{n \in \mathbb{Nat}} \mu_n

\mu_n = \text{let } m' = \{(\texttt{while }^n e \texttt{ do } c)\}_m \text{ in } \{\texttt{if } e \texttt{ then } \texttt{abort}\}_m'

Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_m = \emptyset
\]

\[
\{\text{skip}\}_m = \text{unit}(m)
\]

\[
\{x:=e\}_m = \text{unit}(m[x\leftarrow\{e\}_m])
\]

\[
\{c;c'\}_m = \text{let } m' = \{c\}_m \text{ in } \{c'\}_m
\]

\[
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_m = \{c_t\}_m \text{ if } \{e\}_m = \text{true}
\]

\[
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_m = \{c_f\}_m \text{ if } \{e\}_m = \text{false}
\]

\[
\{\text{while } e \text{ do } c\}_m = \text{sup } \mu_n \in \text{Nat} \mu_n
\]

\[
\mu_n = \text{let } m' = \{(\text{while }_n e \text{ do } c)\}_m \text{ in } \{\text{if } e \text{ then abort}\}_m
\]
Semantics of Commands

This is defined on the structure of commands:

\[
\{\text{abort}\}_m = O
\]

\[
\{\text{skip}\}_m = \text{unit}(m)
\]

\[
\{x := e\}_m = \text{unit}(m[x\leftarrow\{e\}_m])
\]

\[
\{x := \$ d\}_m = \text{let } a = \{d\}_m \text{ in } \text{unit}(m[x\leftarrow a])
\]

\[
\{c; c'\}_m = \text{let } m' = \{c\}_m \text{ in } \{c'\}_m
\]

\[
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_m = \{c_t\}_m \text{ if } \{e\}_m = \text{true}
\]

\[
\{\text{if } e \text{ then } c_t \text{ else } c_f\}_m = \{c_f\}_m \text{ if } \{e\}_m = \text{false}
\]

\[
\{\text{while } e \text{ do } c\}_m = \sup_{n \in \text{Nat}} \mu_n
\]

\[
\mu_n = \text{let } m' = \{(\text{while}_n e \text{ do } c)\}_m \text{ in } \{\text{if } e \text{ then abort}\}_m
\]
Revisiting the example

\begin{verbatim}
OneTimePad(m : private msg) : public msg
key := $ Uniform({0,1}^n);
cipher := msg xor key;
return cipher
\end{verbatim}

Learning a ciphertext does not change any a priori knowledge about the likelihood of messages.
Revisiting the example

```
OneTimePad (m : private msg) : public msg
    key ::= $ Uniform({0,1}^n);
    cipher := msg xor key;
    return cipher
```

Learning a ciphertext does not change any a priori knowledge about the likelihood of messages.

How do we formalize this?
A program $\text{prog}$ is probabilistically noninterferent if and only if, whenever we run it on two low equivalent memories $m_1$ and $m_2$ we have that the probabilistic distributions we get as outputs are the same on public outputs.
Noninterference as a Relational Property

In symbols, c is noninterferent if and only if for every $m_1 \sim_{\text{low}} m_2$:

$$\{c\}_{m_1} = \mu_1 \text{ and } \{c\}_{m_2} = \mu_2 \implies \mu_1 \sim_{\text{low}} \mu_2$$
Revisiting the example

\textbf{OneTimePad}(m:\text{private msg}) : public msg
key := $\text{Uniform}({0,1}^n)$;
cipher := msg xor key;
return cipher
Revisiting the example

```
OneTimePad(m : private msg) : public msg
    key := $ Uniform({0,1}^n);
    cipher := msg xor key;
    return cipher
```

How can we prove that this is noninterferent?