Secure stepwise refinement: Beating the refinement paradox

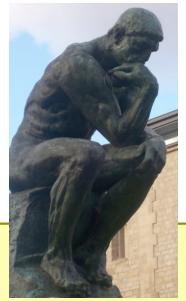
Annabelle McIver
Macquarie University

In brief ...

1. *Traditional refinement;*
2. Security and the “refinement paradox”;
3. Motivation: why refinement and security?
4. The Shadow semantics: beating the paradox;
5. Encryption with refinement;
6. Deriving the Oblivious Transfer;
7. Deriving the Dining Cryptographers.

The *refinement* tradition

What's *"refinement"* ?



The dream...

1. *Adds rigour to program development;*
2. Is a framework for what it means to be correct;
3. *Sanctions the safe...*
4. *...and prevents the unsafe implementations.*

Program development by stepwise refinement
(Wirth, CACM 1971)

What's *"refinement"* ?

WYWIWYG

specification ⊑ implementation

```
public class StatsPackage {
   private realSeq data= [ ];

   void reset() { data= [ ]; }

   void addDatum(real datum) {
         data= data++[datum];
   }

   real mean() throws NoData {
         if (data != [ ])
               return ∑data / #data;
         else throw new NoData("No data points.");
   }
}
```

What's *"refinement"* ?

```
public class StatsPackage' implements StatsPackage {
   private real sum= 0, count= 0;

   void reset() { sum= count= 0; }

   void addDatum(real datum) {
         sum+= datum; count+= 1;
   }

   real mean() throws NoData {
         if (count != 0)
               return sum / count;
         else throw new NoData("No data points.");
   }
}
```

WYWIWYG

specification ⊑ implementation

What's *"refinement"* ?

```
public class StatsPackage {
   private realSeq data= [ ];

   void reset() { data= [ ]; }

   void addDatum(real datum) {
         data= data++[datum];
   }

   real mean() throws NoData {
         if (data != [ ])
               return ∑data / #data;
         else throw new NoData("No data points.");
   }
}
```

```
realSeq getData() { return data; }
```

What's *"refinement"* ?

```
public class StatsPackage' implements StatsPackage {
   private real sum= 0, count= 0;

   void reset() { sum= count= 0; }

   void addDatum(real datum) {
         sum+= datum; count+= 1;
   }

   real mean() throws NoData {
         if (count != 0)
               return sum / count;
         else throw new NoData("No data points.");
   }
}
```

```
realSeq getData() { ??????????? }
```

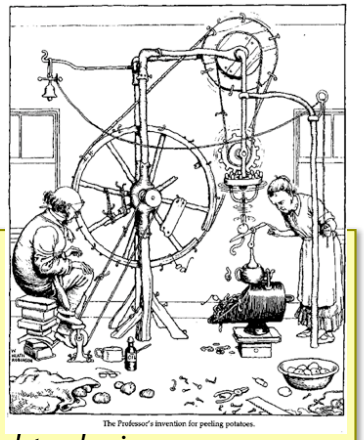
What's *"refinement"* ?

P is refined by *Q* means:

1. *Q* is at least as good as *P* ; or
2. Every possible behaviour of *Q* is a possible behaviour of *P* ; or
3. Everything you can prove about *P* you can also prove about *Q*.

1. What's ***good***?
2. What ***behaviours*** can I observe?
3. What ***logic*** do I use for proof?

What's *"refinement"* ?


The Professor's invention for peeling potatoes.

The reality...

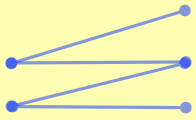
1. *VDM* and *Z*
2. *B* and *event-B*
3. *CSP*
4. The *Refinement Calculus*
5. *A large body of mathematical techniques*

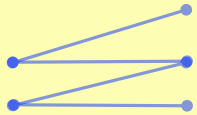
Program development by stepwise refinement
(Wirth, CACM 1971)

But the *Refinement Paradox*
inhibits use of *traditional* refinement in security.

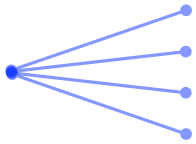
The refinement paradox

- Standard refinement is based on a relational-style semantics, with refinement described by reverse-subset inclusion.

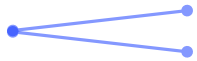




$h := 0 \sqcap h := 1 \;;\; v := 0 \sqcap v := 1 \quad \sqsubseteq \quad h := 0 \sqcap h := 1 \;;\; v := h$



h	v
0	1
1	1
0	0
1	0



h	v
1	1
0	0

The refinement paradox

- Standard formal security analysis is based on classifying variables into two kinds: “high” and “low” (or “hidden” and “visible”);
- Programs are said to be secure (or not) depending on whether their execution causes the hidden values to be revealed;
- A refinement-oriented framework should ensure that even security properties are preserved...

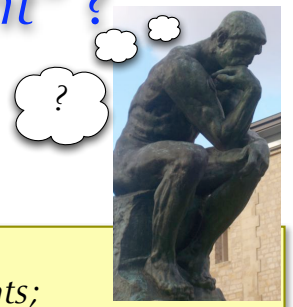
$$v := 0 \sqcap v := 1 \;;\; h := 0 \sqcap h := 1$$
$$= h := 0 \sqcap h := 1 \;;\; v := 0 \sqcap v := 1 \quad \sqsubseteq \quad h := 0 \sqcap h := 1 \;;\; v := h$$

This would be considered secure...

This refinement doesn’t preserve secrecy!

... but clearly not this.

What's *"secure refinement"* ?



The dream...

1. *Adds rigour to secure developments;*
2. Is a framework for what it means to be secret;
3. *Sanctions security-preserving programs...*
4. *...and prevents insecure implementations.*

"Secure program development by stepwise refinement"
(Morgan, MPC 2006)

Motivation

Oblivious transfer

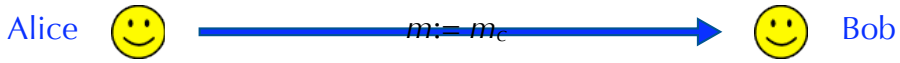
Oblivious transfer: the problem

Alice has two messages;
Bob knows their "names",
but not their contents.

Bob asks for one,
by name.

m_0, m_1: M

c: {0,1}



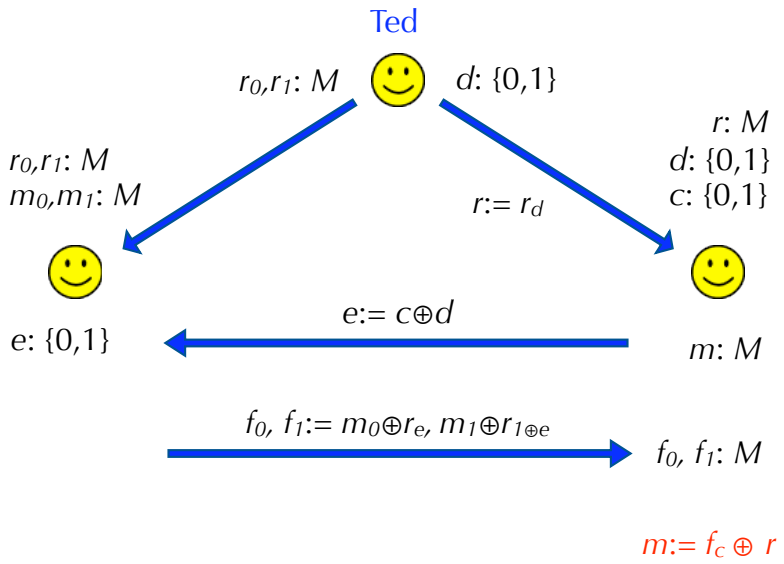
m: M

Alice sends it...

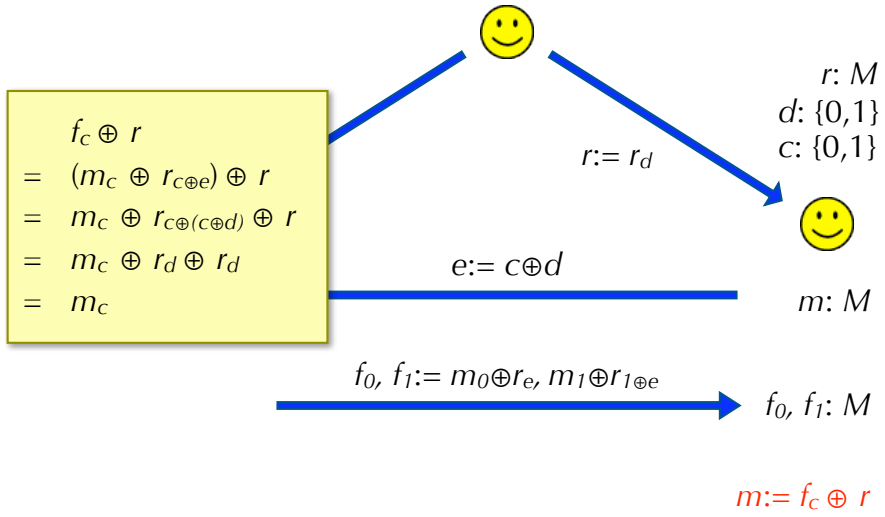
...Bob receives it.

But Alice is not to know which message Bob asked for,
and Bob is not to find out the contents of the other one.

Oblivious transfer: the solution



Oblivious transfer: the "proof"



So what?

So what?

What “we” do...

$m := m_c$ — Alice, Bob

✗ Not localised.

$\sqsubseteq$ { **var** c': {0,1} •
$c' := c$; ←
$m := m_{c'}$
}

✗ Reveals c to Alice.

$\sqsubseteq$ { **var** m_0', m_1': M •
$m_0', m_1' := m_0, m_1$; ←
$m := m_{c'}$
}

✗ Reveals $m_{1 \oplus c}$ to Bob.

Refinement

So what?

What Rivest does

$m := m_c$

$\sqsubseteq$ { **var** $r, r_0, r_1, f_0, f_1 : M$
$d,e : \{0,1\}$ •

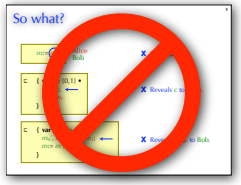
$r_0, r_1 :\in M,M;$
$d :\in \{0,1\};$
$r := r_d;$
$e := c \oplus d;$
$f_0, f_1 := m_0 \oplus r_c, m_1 \oplus r_{1 \oplus c};$
$m := f_c \oplus r$
}

✗ Not localised.

✓ Reveals *only* m_c to Bob.

? But how to prove this?

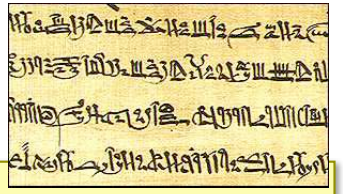
? And prevent these?



That's what.

A new approach: Shadow semantics

What's *"secure refinement"* ?

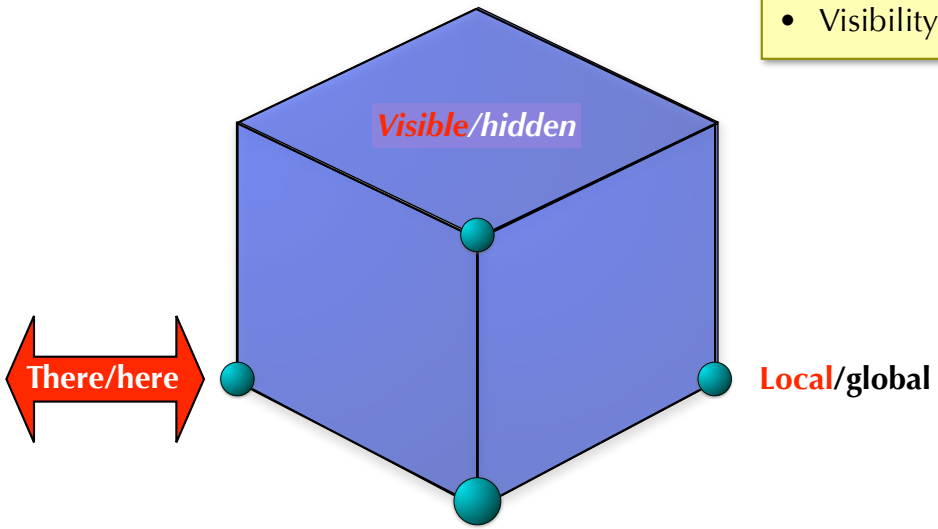


The reality...

The way to resolve the paradox is to regard insecure refinements at "attacks" performed during development. Beating the refinement paradox is possible within a framework which can distinguish between secure and insecure refinements.

A new approach

- Scope
- Locale
- Visibility



What kind of variables are there?

A new approach

- Scope: Newly-declared, local variables can have arbitrary values assigned to them:

  skip ⊑ **{ var** *x*: *T* • *x*:= “anything” ...

  Global variables are constrained by the specification:

  skip ⋤ *x*:= “anything”

- Locale: Mixed locales cannot occur inside expressions:

  x:= *here* + *there* ✗ not allowed in code

  Mixed locales across “:=” imply communication:

  there:= *here* ✓ sends/receives a message

A new approach

- Visibility: Hidden variables cannot be revealed, even partially, by refinement:

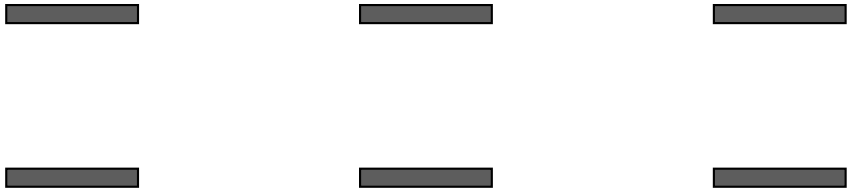
  $m_0 :\in M \quad \not\sqsubseteq \quad m_0 := r$

  Visible variables and constants can:

  $m_0 :\in M \quad \sqsubseteq \quad m_0 := m_1$

  $m_0 :\in M \quad \sqsubseteq \quad m_0 :=$ *"This is written in the code."*

Gedanken experiments in program algebra



What are our goals?

Security, privacy, zero-knowledge protocols...
done *algebraically*, *logically*, *"refinably"*.

- Treat sequential programs in the "usual" way, as far as possible, but...
- Include *ignorance* (somehow defined) among the properties that the refinement is to preserve...
- Thus permitting fewer (but not too few) refinements than before, while...
- Keeping certain practical principles in mind, for scalability.

- Which principles?

What are our principles?

Pr1 *All traditional "visible-only" refinements are retained* — It would be impractical to search an entire program for hidden variables in order to validate local *visible-only* reasoning in which the hiddens are not mentioned.

Pr2 *All traditional "structural" refinements are retained* — Associativity of sequential composition, distribution of code into branches of a conditional *etc.* are refinements (actually equalities) that do not depend on the actual code fragments affected: they are *structurally* valid, acting *en bloc*. It would be impractical to have to check through the fragments' interiors (including *e.g.* procedure calls) to validate such familiar rearrangements.

Pr3 *Some traditional "explicit-hidden" refinements are excluded* — Those that preserve ignorance will be retained; the others (*e.g.* the Paradox) will be excluded. For this principle we need a model and a logic.

More briefly...

- Equality- and refinement laws of the program algebra should be preserved "as much as possible". But which are the most important?
  - *Laws in which no hidden variable appears.*
  - *"Schematic" laws referring to general program fragments.*
- Which can we afford to lose?
  - *Laws in which hidden variables appear explicitly.*

More briefly...

- Which laws do we keep?

$v := 0 \sqcap v := 1$ ✓ Doesn't involve hiddens

$\sqsubseteq$ **if** $(v \leq 10)$ **then** $v := 1$ **else** $v := 0$

$P ; (Q \sqcap R) = P;Q \sqcap P;R$ ✓ Schematic law

- Which laws can we lose?

$v := 0 \sqcap v := 1$ ✗ Does involve hiddens

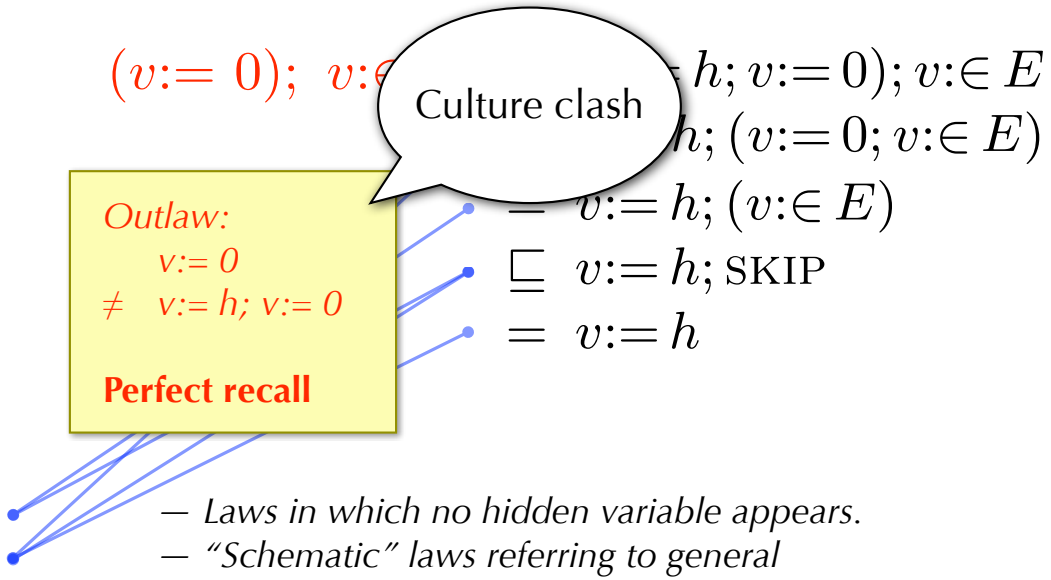
$\not\sqsubseteq$ **if** $(h \leq 10)$ **then** $v := 0$ **else** $v := 1$

$v := 0 \quad \neq \quad v := h; v := 0$ ✗ But this is also banned

Gedanken argument... for perfect recall

$$
\begin{aligned}
& (v{:=}\, h; v{:=}\, 0); v{:\in} E \\
= \;& v{:=}\, h; (v{:=}\, 0; v{:\in} E) \\
= \;& v{:=}\, h; (v{:\in} E) \\
\sqsubseteq \;& v{:=}\, h; \mathrm{SKIP} \\
= \;& v{:=}\, h
\end{aligned}
$$

Gedanken argument...



— *Laws in which no hidden variable appears.*
— *"Schematic" laws referring to general program fragments.*

Gedanken argument... for history of program counter

Another outlaw:
if *E(h)* ***then skip else skip fi*** $\neq$ ***skip***

$$\begin{array}{ll} & \textbf{if } h = 0 \textbf{ then } \textsc{skip} \textbf{ else } \textsc{skip} \textbf{ fi}; \quad v{:\in} E \\ = & \textbf{if } h = 0 \textbf{ then } \textsc{skip}; v{:\in} E \textbf{ else } \textsc{skip}; v{:\in} E \textbf{ fi} \\ = & \textbf{if } h = 0 \textbf{ then } v{:\in} E \textbf{ else } v{:\in} E \textbf{ fi} \\ \sqsubseteq & \textbf{if } h = 0 \textbf{ then } v{:=} \textsc{true} \textbf{ else } v{:=} \textsc{false} \textbf{ fi} \\ = & v{:=}(h = 0) \end{array}$$

How do we outlaw the outlaws?

only

Programming language syntax

Identity	**skip**
Assigment	$x := E$
Choose	$x :\in E$
Demonic choice	$\mathrm{S1} \sqcap \mathrm{S2}$
Composition	$\mathrm{S1}; \mathrm{S2}$
Conditional	**if** E **then** S1 **else** S2 **fi**
Declare visible	$[\![\textsc{vis}\ v \cdot \mathrm{S}]\!]$
Declare hidden	$[\![\textsc{hid}\ h \cdot \mathrm{S}]\!]$

Informal examples of observations

	Program	Informal commentary
1.1	both $v{:\in}\,\{0,1\}$ and $v{:=}\,0 \sqcap v{:=}\,1$	We can see the value of v, either 0 or 1.We know h is h_0, though we cannot see it.
1.2	$h{:\in}\,\{0,1\}$	We know that h is either 0 or 1, but we don't know which; we see that v is v_0.
1.3	(two atomic statements) $h{:=}\,0 \sqcap h{:=}\,1$	We know the value of h, because from the program-counter history we know which of the atomic $h{:=}\,0$ or $h{:=}\,1$ was executed.
1.4	$h{:\in}\,\{0,1\}$; $v{:=}\,0 \sqcap v{:=}\,1$	We don't know whether h is 0 or it is 1: even the $\sqcap$-demon cannot see the hidden variable.
1.5	$h{:\in}\,\{0,1\}$; $v{:\in}\,\{h, 1{-}h\}$	Though the choice of v refers to h it reveals no information, since the statement is atomic.
1.6	$h{:\in}\,\{0,1\}$; $v{:=}\,h \sqcap v{:=}\,1{-}h$	Here h is revealed, because we know which of the two atomic assignments to v was executed.
1.7	$h{:\in}\,\{0,1,2,3\}$; $v{:=}\,h$	We see v; we deduce h since we can see $v{:=}\,h$ in the program text.
1.8	$h{:\in}\,\{0,1,2,3\}$; $v{:=}\,h \bmod 2$	We can see v; either we deduce h is 0 or 2, or that h is 1 or 3.
1.9	$h{:\in}\,\{0,1,2,3\}$; $v{:=}\,h \bmod 2$; $v{:=}\,0$	We see v is 0; but our deductions about h are as for 1.8, because we saw v's earlier value.

Add more structure...

The *Shadow Semantics* adds an extra component to the normal relational semantics of sequential programs, one that keeps "in the shadows" the *set* of possible hidden values the observer must consider possible, based on what he has seen of the execution so far.

Ignorance Refinement takes the shadow component into account, not allowing it to shrink; and the reduce-nondeterminism/increase-termination rules of ordinary refinement continue to apply.

A *weakest-precondition* logic can be given. Ultimately the logic and model borrows from the *Kripke* structu of Knowledge.

Derived by quotienting a Kripke model

Shadow Semantics

- Structure the state space using a triple.
- The H component (the shadow) captures what an observer can infer is possible (given the observed behaviour of the program).
- The underlying order encodes what we mean by preservation of ignorance.

$$(v_0, h_0, H_0) \sqsubseteq' (v_1, h_1, H_1)$$

iff $v_0 = v_1 \wedge h_0 = h_1 \wedge H_0 \subseteq H_1$

Ignorance refinement

The operational view

- Program P refines Q just when P's result triples are contained in Q's result set (of triples);
- "Healthiness conditions" ensure that h's value is revealed just when the H set collapses.

If (v, h, H) is contained in a result set of program P, then

H1 : $h \in H$;
H2 : so is (v, h′, H), for any $h' \in H$;
H3 : if (v, h, H′) is also in the result set, then so is (v, h, $H' \cup H$).

Ignorance refinement

The operational view

Identity	$[\![\textbf{skip}]\!]$	$\widehat{=}$	**skip**
Assign to visible	$[\![v{:=}\,E]\!]$	$\widehat{=}$	$e{:=}\,E;\ H{:=}\,\{h{:}\,H \mid e{=}E\};\ v{:=}\,e$
Choose visible	$[\![v{:\in}\,E]\!]$	$\widehat{=}$	$e{:\in}\,E;\ H{:=}\,\{h{:}\,H \mid e{\in}E\};\ v{:=}\,e$
Assign to hidden	$[\![h{:=}\,E]\!]$	$\widehat{=}$	$h{:=}\,E;\ H{:=}\,\{E \mid h{:}\,H\}$
Choose hidden	$[\![h{:\in}\,E]\!]$	$\widehat{=}$	$h{:\in}\,E;\ H{:=}\,\cup\{E \mid h{:}\,H\}$
Demonic choice	$[\![\mathrm{S1} \sqcap \mathrm{S2}]\!]$	$\widehat{=}$	$[\![\mathrm{S1}]\!] \sqcap [\![\mathrm{S2}]\!]$
Composition	$[\![\mathrm{S1};\mathrm{S2}]\!]$	$\widehat{=}$	$[\![\mathrm{S1}]\!];\ [\![\mathrm{S2}]\!]$

Conditional $[\![\textbf{if}\ E\ \textbf{then}\ \mathrm{S1}\ \textbf{else}\ \mathrm{S2}\ \textbf{fi}]\!]$

$\widehat{=}$ $\textbf{if}\ E\ \textbf{then}\ H{:=}\,\{E \mid h{:}\,H\};[\![\mathrm{S1}]\!]\ \textbf{else}\ H{:=}\,\{\neg E \mid h{:}\,H\};[\![\mathrm{S2}]\!]\ \textbf{fi}$

Examples of operational semantics

The initial state is $(\mathsf{v}_0, \mathsf{h}_0, \{\mathsf{h}_0\})$.

	Program	Final states in the "reduced" $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ model	
3.1	both $v{:}\in \{0,1\}$ and $v{:=}0 \sqcap v{:=}1$	$(0, h_0, \{h_0\})$, $(1, h_0, \{h_0\})$	
3.2	$h{:}\in \{0,1\}$	$(v_0, 0, \{0,1\})$, $(v_0, 1, \{0,1\})$	
3.3	$h{:=}0 \sqcap h{:=}1$	$(v_0, 0, \{0\})$, $(v_0, 1, \{1\})$	
3.4	$h{:}\in \{0,1\}$; $v{:=}0 \sqcap v{:=}1$	$(0,0,\{0,1\})$, $(0,1,\{0,1\})$, $(1,0,\{0,1\})$, $(1,1,\{0,1\})$	
3.5	$h{:}\in \{0,1\}$; $v{:}\in \{h, 1{-}h\}$	$(0,0,\{0,1\})$, $(1,0,\{0,1\})$, $(0,1,\{0,1\})$, $(1,1,\{0,1\})$	Thus this and 3.4 are equal.
3.6	$h{:}\in \{0,1\}$; $v{:=}h \sqcap v{:=}1{-}h$	$(0,0,\{0\})$, $(1,0,\{0\})$, $(0,1,\{1\})$, $(1,1,\{1\})$	But this one differs.
3.7	$h{:}\in \{0,1,2,3\}$; $v{:=}h$	$(0,0,\{0\})$, $(1,1,\{1\})$, $(2,2,\{2\})$, $(3,3,\{3\})$	
3.8	$h{:}\in \{0,1,2,3\}$; $v{:=}h \bmod 2$	$(0,0,\{0,2\})$, $(1,1,\{1,3\})$, $(0,2,\{0,2\})$, $(1,3,\{1,3\})$	
3.9	$h{:}\in \{0,1,2,3\}$; $v{:=}h \bmod 2$; $v{:=}0$	$(0,0,\{0,2\})$, $(0,1,\{1,3\})$, $(0,2,\{0,2\})$, $(0,3,\{1,3\})$	The final $v{:=}0$ does not affect H.

Ignorance refinement

The operational view

$v{:\in}\ V \quad \sqsubseteq \quad v{:=}\ v'$

$v{:\in}\ V \quad \not\sqsubseteq \quad v{:=}\ h$

We can now outlaw refinements which reveal hidden state.

(*initial state-triple*) **[** *program* **]** (*final state-triple*)

(v,h,H) **[** $v{:\in}\ V$ **]** (x,h,H) any $x \in V$

(v,h,H) **[** $v{:=}\ v'$ **]** (v',h,H) $v' \in V$

(v,h,H) **[** $v{:=}\ h$ **]** $(h,h,\{h\})$ $h \in V$, but $H \not\subseteq \{h\}$ in general

Weakest precondition

The logical view

- We introduce knowledge/ignorance modalities **K** / **P** (they are dual) so that for example
  - **K**(h>0) means we know that h is positive even though we can't see it.
  - **P**(h>0) means we don't know that h isn't positive.
- We focus on ignorance formulae, those in which **K** occurs only negatively, equivalently in which **P** occurs only positively.
- We interpret these formulae à la $v,h,H \models \Phi$ with **P** acting existentially in H.

Logical vs operational semantics

The operational-logical connection:

We define truth of Φ at $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ under valuation w by induction, writing $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi$. Let t be the term-valuation built inductively from the valuation $\mathsf{v} \lhd \mathsf{h} \lhd \mathsf{w}$. Then we have the following [9, pp. 79,81]:

- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \mathrm{R}.\mathrm{T}_1.\cdots.\mathrm{T}_k$ for relation symbol R and terms $\mathrm{T}_1 \cdots \mathrm{T}_k$ iff the tuple $(\mathsf{t}.\mathrm{T}_1, \cdots, \mathsf{t}.\mathrm{T}_k)$ is an element of the interpretation of R.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \mathrm{T}_1 = \mathrm{T}_2$ iff $\mathsf{t}.\mathrm{T}_1 = \mathsf{t}.\mathrm{T}_2$.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \neg\Phi$ iff $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \not\models \Phi$.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi_1 \wedge \Phi_2$ iff $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi_1$ and $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi_2$.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models (\forall \mathrm{L} \cdot \Phi)$ iff $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \lhd \langle \mathrm{L} \mapsto \mathsf{d} \rangle \models \Phi$ for all d in D.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \mathrm{K}\Phi$ iff $(\mathsf{v}, \mathsf{h}_1, \mathsf{H}), \mathsf{w} \models \Phi$ for all h_1 in H.

Logical vs operational semantics

The operational-logical connection:

- Based on our operational understanding, we define $(\mathsf{v}, \mathsf{h}, \mathsf{H})\ [\![P]\!]\ (\mathsf{v}', \mathsf{h}', \mathsf{H}')$ inductively.

- We (re-)define the *Hoare-triple* $\{\Phi\}\ P\ \{\Psi\}$ as

$$\begin{array}{ll} & (\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \Phi \\ \wedge & (\mathsf{v}, \mathsf{h}, \mathsf{H})\ [\![P]\!]\ (\mathsf{v}', \mathsf{h}', \mathsf{H}') \\ & \\ \Rightarrow & (\mathsf{v}', \mathsf{h}', \mathsf{H}') \models \Psi \end{array}$$

Logical vs operational semantics

The operational-logical connection (continued):

- We define Dijkstra-style *weakest preconditions* so that

$$wp.P.\Psi \quad \text{is the } \textit{weakest } \Phi \text{ such that} \quad \{\Phi\}\ P\ \{\Psi\}$$

- We give syntactic *predicate-transformer style* rules for calculating weakest-preconditions in general.

Logical vs operational semantics

The operational-logical connection (continued):

- We define *refinement* based on our logic of ignorance:

$$P \sqsubseteq Q \quad \text{means} \quad \models wp.P.\Psi \Rightarrow wp.Q.\Psi \quad \text{for all } \textit{ignorant } \Psi$$

An outlaw outlawed

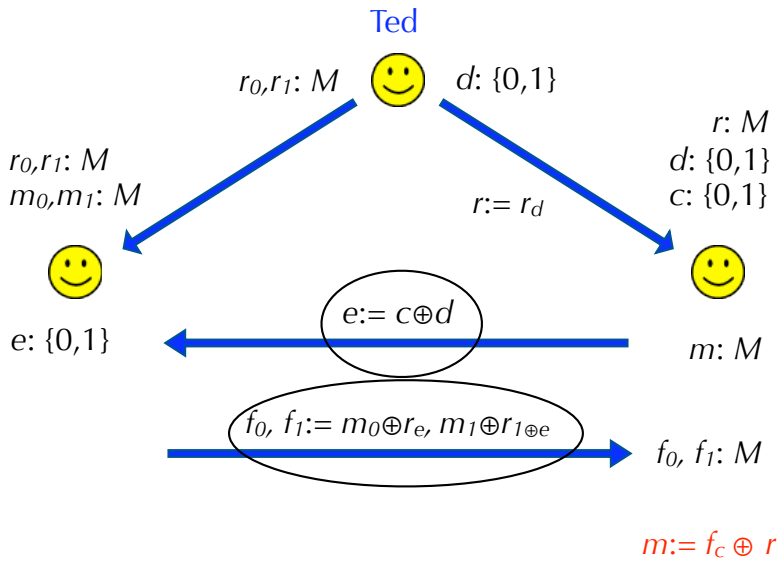
The logical view

{**P**(h=C)}	$v :\in V$	{**P**(h=C)}	holds for all C
{**P**(h=C)}	$v := h$	{**P**(h=C)}	does not hold in general

Examples of modal postconditions

	Program	Valid Ψ (in these examples, for any Φ)	Invalid Ψ
6.1	both $v{:\in}\{0,1\}$ and $v{:=}0 \sqcap v{:=}1$	$v \in \{0,1\}$	$v = 0$
6.2	$h{:\in}\{0,1\}$	$\mathrm{P}(h{=}0)$	$\mathrm{K}(h{=}0)$
6.3	$h{:=}0 \sqcap h{:=}1$	$h \in \{0,1\}$	$\mathrm{P}(h{=}0)$
6.4	$h{:\in}\{0,1\}$; $v{:=}0 \sqcap v{:=}1$	$\mathrm{P}(v{=}h)$	$\mathrm{K}(v{\neq}h)$
6.5	$h{:\in}\{0,1\}$; $v{:\in}\{h, 1{-}h\}$	$\mathrm{P}(h{=}0)$ In fact Program 6.5 equals Program 6.4.	$\mathrm{P}(v{=}0)$
6.6	$h{:\in}\{0,1\}$; $v{:=}h \sqcap v{:=}1{-}h$	$v \in \{0,1\}$ But Program 6.6 differs from Program 6.5.	$\mathrm{P}(h{=}0)$
6.7	$h{:\in}\{0,1,2,3\}$; $v{:=}h$	$\mathrm{K}(v{=}h)$	$\mathrm{P}(v{\neq}h)$
6.8	$h{:\in}\{0,1,2,3\}$; $v{:=}h \bmod 2$	$v{=}0$ $\Rightarrow$ $\mathrm{P}(h{\in}\{2,4\})$	$\mathrm{P}(h{=}1)$ $\wedge$ $\mathrm{P}(h{=}2)$
6.9	$h{:\in}\{0,1,2,3\}$; $v{:=}h \bmod 2$; $v{:=}0$	$\mathrm{P}(h{\in}\{1,2\})$	$v{=}0$ $\Rightarrow$ $\mathrm{P}(h{\in}\{2,4\})$

Oblivious transfer: the solution

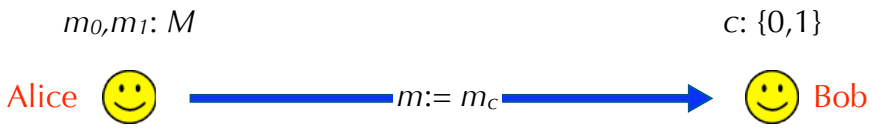


The encryption lemma

This formalises the basic idea of how to hide information by publishing the result of an exclusive OR with the secret. The Encryption Lemma gives the conditions under which no information is revealed. And it can be expressed very easily using a secure refinement law.

$$\textbf{skip} \sqsubseteq \{\textbf{hid}\ x;\ \textbf{vis}\ y \bullet x :\in X;\ y := x \oplus h\}$$

Keeping secrets is meaningless without multiple viewpoints



$\sqsubseteq$ "standard reasoning"

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e : \{0,1\};\ f_0, f_1, r : M; \\
\quad \textbf{hid}\ r_0, r_1 : M \ \cdot \\
\quad r_0, r_1 :\in M, M; \\
\quad d :\in \{0,1\}; \\
\quad r := r_d; \\
\quad e := c \oplus d \\
\quad f_0 := m_0 \oplus r_e\ ; \\
\quad f_1 := m_1 \oplus r_{1 \oplus e}\ ; \\
\quad m := f_c \oplus r \\
\}
\end{array}
$$

Bob's view

Derivation of *OTP*

$m := m_c$

$\sqsubseteq$ **skip**;
$m := m_c$

$\sqsubseteq$ "local visible variables"
{ vis $d, e : \{0,1\}$ **·**
$d :\in \{0,1\}$;
$e := c \oplus d$;
$m := m_c$
}

$\sqsubseteq$ "encryption lemma"
{ vis $d, e : \{0,1\}$; $f_0, f_1 : M$;
hid $r_0, r_1 : M$ **·**
$d :\in \{0,1\}$;
$e := c \oplus d$
$r_0, r_1 :\in M, M$;
$f_0 := m_0 \oplus r_e$;
$f_1 := m_1 \oplus r_{1 \oplus e}$;
$m := m_c$
}

skip $\sqsubseteq$ {**hid** *x*; **vis** *y* • *x*:∈ *X*; *y*:= *x*⊕*h*}

Derivation of *OTP*

$\sqsubseteq$ "encryption lemma"

$\{$ **vis** $d, e\colon\{0,1\};\ f_0, f_1\colon M;$
hid $r_0, r_1\colon M$ $\cdot$

$d{:}\in \{0,1\};$
$e{:=}\, c \oplus d$
$r_0, r_1 :\in M, M;$
$f_0{:=}\, m_0 \oplus r_e\ ;$
$f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ;$
$m{:=}\, m_c$
$\}$

$\sqsubseteq$ "local visible variable"

$\{$ **vis** $d, e\colon\{0,1\};\ f_0, f_1, r\colon M;$
hid $r_0, r_1\colon M$ $\cdot$

$d{:}\in \{0,1\};$
$e{:=}\, c \oplus d$
$r_0, r_1 :\in M, M;$
$f_0{:=}\, m_0 \oplus r_e\ ;$
$f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ;$
$m{:=}\, m_c;$
$r{:=}\, f_c \oplus m$
$\}$

Step 4 of 7

Derivation of *OTP*

$\sqsubseteq$ "local visible variable"

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e{:}\{0,1\};\ f_0, f_1, r{:}M; \\
\quad \textbf{hid}\ r_0, r_1{:}M\ \cdot \\
\quad\quad d{:}\in \{0,1\}; \\
\quad\quad e{:=}\, c \oplus d \\
\quad\quad r_0, r_1\ {:}\in\ M, M; \\
\quad\quad f_0{:=}\, m_0 \oplus r_e\ ; \\
\quad\quad f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ; \\
\quad\quad m{:=}\, m_c; \\
\quad\quad r{:=}\, f_c \oplus m \\
\}
\end{array}
$$

$\sqsubseteq$ "standard reasoning"

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e{:}\{0,1\};\ f_0, f_1, r{:}M; \\
\quad \textbf{hid}\ r_0, r_1{:}M\ \cdot \\
\quad\quad d{:}\in \{0,1\}; \\
\quad\quad e{:=}\, c \oplus d \\
\quad\quad r_0, r_1\ {:}\in\ M, M; \\
\quad\quad f_0{:=}\, m_0 \oplus r_e\ ; \\
\quad\quad f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ; \\
\quad\quad m{:=}\, m_c; \\
\quad\quad \boxed{r{:=}\, r_d} \\
\}
\end{array}
$$

Step 5 of 7

Derivation of *OTP*

$\sqsubseteq$ "standard reasoning"

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e{:}\{0,1\};\ f_0, f_1, r{:}M; \\
\quad \textbf{hid}\ r_0, r_1{:}M\ \bullet \\
\quad\quad d{:}\in \{0,1\}; \\
\quad\quad e{:=}\, c \oplus d \\
\quad\quad r_0, r_1 :\in\ M, M; \\
\quad\quad f_0{:=}\, m_0 \oplus r_e\ ; \\
\quad\quad f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ; \\
\quad\quad m{:=}\, m_c; \\
\quad\quad r{:=}\, r_d \\
\}
\end{array}
$$

$\sqsubseteq$ "reordering"

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e{:}\{0,1\};\ f_0, f_1, r{:}M; \\
\quad \textbf{hid}\ r_0, r_1{:}M\ \bullet \\
\quad\quad r_0, r_1 :\in\ M, M; \\
\quad\quad d{:}\in \{0,1\}; \\
\quad\quad r{:=}\, r_d; \\
\quad\quad e{:=}\, c \oplus d \\
\quad\quad f_0{:=}\, m_0 \oplus r_e\ ; \\
\quad\quad f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ; \\
\quad\quad m{:=}\, m_c \\
\}
\end{array}
$$

Step 6 of 7

Derivation of *OTP*

$\sqsubseteq$ "reordering"

$\{$ **vis** $d, e\colon \{0,1\};\ f_0, f_1, r\colon M;$
hid $r_0, r_1\colon M$ $\cdot$

$r_0, r_1 :\in M, M;$
$d :\in \{0,1\};$
$r := r_d;$
$e := c \oplus d$
$f_0 := m_0 \oplus r_e\ ;$
$f_1 := m_1 \oplus r_{1\oplus e}\ ;$
$m := m_c$
$\}$

$\sqsubseteq$ "standard reasoning"

$\{$ **vis** $d, e\colon \{0,1\};\ f_0, f_1, r\colon M;$
hid $r_0, r_1\colon M$ $\cdot$

$r_0, r_1 :\in M, M;$
$d :\in \{0,1\};$
$r := r_d;$
$e := c \oplus d$
$f_0 := m_0 \oplus r_e\ ;$
$f_1 := m_1 \oplus r_{1\oplus e}\ ;$
$m := f_c \oplus r$
$\}$

Final step

Overall, in 7 steps,

$m := m_c$

Bob's point of view.

Ted sends r_0, r_1 to Alice.
Ted sends d to Bob.
Ted sends r_d to Bob.
Bob sends e to Alice.
Alice sends f_0, f_1 to Bob.
Bob computes m_c.

$\sqsubseteq$ **Ignorance-preserving refinement**

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e\colon \{0,1\};\ f_0, f_1, r\colon M; \\
\quad \textbf{hid}\ r_0, r_1\colon M\ \bullet \\
\quad\quad r_0, r_1 :\in M, M; \\
\quad\quad d :\in \{0,1\}; \\
\quad\quad r := r_d; \\
\quad\quad e := c \oplus d \\
\quad\quad f_0 := m_0 \oplus r_e\ ; \\
\quad\quad f_1 := m_1 \oplus r_{1\oplus e}\ ; \\
\quad\quad m := f_c \oplus r \\
\}
\end{array}
$$

Rivest's *Oblivious Transfer Protocol*.

Conclusion

$m{:=}\, m_c$ $\sqsubseteq$ **Ignorance-preserving refinement**

$$
\begin{array}{l}
\{\ \textbf{vis}\ d, e{:}\{0,1\};\ f_0, f_1, r{:}M; \\
\quad \textbf{hid}\ r_0, r_1{:}M \cdot \\
\quad\ r_0, r_1 :\in M, M; \\
\quad\ d{:}\in \{0,1\}; \\
\quad\ r{:=}\, r_d; \\
\quad\ e{:=}\, c \oplus d \\
\quad\ f_0{:=}\, m_0 \oplus r_e\ ; \\
\quad\ f_1{:=}\, m_1 \oplus r_{1\oplus e}\ ; \\
\quad\ m{:=}\, f_c \oplus r \\
\}
\end{array}
$$

The desired security properties do not have to be listed initially.

This is the power of refinement and program algebra.

A calculus of revelations

Having an explicit **reveal** command simplifies the algebra considerably, and focuses attention —where desired— on the pure security properties.

$$\mathbf{reveal}\ E \quad = \quad |[\ \mathbf{vis}\ v \cdot v{:=}E\ ]|$$

Previous approach: harder to manipulate.

The Encryption Lemma: a piece of algebraic Lego

$|[\ \mathbf{hid}\ h'\ ;\ h' :\in \{0,1\}\ ;\ \mathbf{reveal}\ h \oplus h'\ ]|$

"Here's a little refinement I prepared earlier." = **skip**

hid $h\ \cdot$

$$|[\ \mathbf{hid}\ h'\ ;\ h' :\in \{0,1\}\ ;\ \mathbf{reveal}\ h \oplus h'\ ]|$$

$$= \quad \mathbf{skip}$$

Does this program fragment reveal anything about *h*?
No — it reveals nothing at all.

A calculus of revelations

(1+2=3) Combine E with F; replace F with F'; separate E and F'.

$$\textbf{reveal}\ E;\ \textbf{reveal}\ F \quad = \quad \textbf{reveal}\ (E, F)$$

$$\begin{aligned} & \textbf{reveal}\ x{\oplus}y;\ \textbf{reveal}\ y{\oplus}z \\ = \quad & \textbf{reveal}\ (x{\oplus}y, y{\oplus}z) \\ = \quad & \textbf{reveal}\ (x{\oplus}y, x{\oplus}z) \\ = \quad & \textbf{reveal}\ x{\oplus}y;\ \textbf{reveal}\ x{\oplus}z \end{aligned}$$

addition mod 2 or, equivalently, exclusive-or