A Calculus of Revelations

being an application of
the Shadow theory,
illustrated with source-level proofs
of security protocols

Annabelle McIver       Carroll Morgan
Security and refinement using the Shadow
Security and refinement using the Shadow

The principles (a sketch)

This is an *informal* talk on a topic whose basis is rigorously formalised: thus some (indeed, many) claims will be made without proof; and some (indeed, many) manipulations will be carried out without justification.

But they can be (and have been) proved and justified elsewhere.
Security and refinement using the Shadow

The principles (a sketch)

This is an *informal* talk on a topic whose basis is rigorously formalised: thus some (indeed, many) claims will be made without proof; and some (indeed, many) manipulations will be carried out without justification.

But they can be (and have been) proved and justified elsewhere.
Security and refinement using the Shadow:

The principles (a sketch): the approach...

... is related to non-interference
... but allows refinement
... encourages source-level reasoning
... is very suitable for mechanisation

Rather than describe the theory in detail, I will describe how we want to use it, and why.
Security and refinement: based on non-interference

Variables are partitioned into *visible* (low-security, labelled `vis`) and *hidden* (high-security, labelled `hid`).

Visible variables are directly observable; hidden variables are not.

Hidden variables’ values can be deduced by observing the visible variables’ (intermediate) values, knowing the source code, and following the execution’s path through it.

Refinement — from a security point of view — is viewed as an adversary.
Refinement — from a security point of view — is viewed as an adversary.
Refinement as an adversary?

Refinement — from a security point of view — is viewed as an adversary.
Refinement as an adversary?

In brief: allowing someone to refine your carefully crafted code is just as bad as allowing your run-time adversary to have perfect recall and to observe the program flow.

Refinement — from a security point of view — is viewed as an adversary.
Refinement as an *adversary*?

In brief: allowing someone to refine your carefully crafted code is just as bad as allowing your run-time adversary to have perfect recall and to observe the program flow.

These are however the same assumptions you might make to model a distributed algorithm with a single sequential program, a technique which allows considerable simplification.

Refinement — from a security point of view — is viewed as an *adversary*. 
Refinement as an adversary?

In brief: allowing someone to refine your carefully crafted code is just as bad as allowing your run-time adversary to have perfect recall and to observe the program flow.

These are however the same assumptions you might make to model a distributed algorithm with a single sequential program, a technique which allows considerable simplification.

In spite of this, refinement is such an important development tool that it is worth taking the security hit. Indeed, sometimes the adversary has that power anyway (as in today’s examples).

Refinement — from a security point of view — is viewed as an adversary.
Shadow-style program logic, based on assertions of knowledge and ignorance
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary Knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the Possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\psi$ means “the adversary Knows $\psi$,” i.e. that $\psi$ can be deduced by observation.

$P\psi$ means “the adversary admits the Possibility of $\psi$,” i.e. that $\psi$ cannot be ruled out by observation.

They are dual: $P\psi$ is the same as $\neg K(\neg \psi)$. 
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary Knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the Possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h:=1$ does establish $K(h=1)$

They are dual: $P\Psi$ is the same as $\neg K(\neg \Psi)$.
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary Knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the Possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h:= 1$ does establish $K(h=1)$

$h:\epsilon \{0,1\}$ does not establish $K(h=1)$

They are dual: $P\Psi$ is the same as $\neg K(\neg \Psi)$. 

Hoare-triples and weakest preconditions for knowledge and ignorance

\( K\Psi \) means “the adversary \text{Knows} \ \Psi,” i.e. that \( \Psi \) can be deduced by observation.

\( P\Psi \) means “the adversary admits the \text{Possibility of} \ \Psi,” i.e. that \( \Psi \) cannot be ruled out by observation.

\( h:= 1 \) does establish \( K(h=1) \)

\( h: \in \{0,1\} \) does \textbf{not} establish \( K(h=1) \)

\( h: \in \{0,1\} \) does establish \( P(h=1) \)

They are dual: \( P\Psi \) is the same as \( \neg K(\neg \Psi) \).
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h:= 1$ does establish $K(h=1)$
$h:\in \{0,1\}$ does not establish $K(h=1)$
$h:\in \{0,1\}$ does establish $P(h=1)$
$h:\in \{0,1\}$ does not establish $P(h=2)$

They are dual: $P\Psi$ is the same as $\neg K(\neg \Psi)$. 
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary Knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the Possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h:= 1$ does establish $K(h=1)$
$h:\in \{0,1\}$ does not establish $K(h=1)$
$h:\in \{0,1\}$ does establish $P(h=1)$
$h:\in \{0,1\}$ does not establish $P(h=2)$
$h:\in \{0,1\}; \nu:\in \{0,1\}$ does establish $P(h=1)$
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary \textbf{K}nows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the \textbf{P}ossibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h := 1$ does establish $K(h=1)$

$h \in \{0,1\}$ does \textbf{not} establish $K(h=1)$

$h \in \{0,1\}$ does establish $P(h=1)$

$h \in \{0,1\}$ does \textbf{not} establish $P(h=2)$

$h \in \{0,1\}; \; v \in \{0,1\}$ does establish $P(h=1)$

$h \in \{0,1\}; \; v := h$ does \textbf{not} establish $P(h=1)$
Hoare-triples and weakest preconditions for knowledge and ignorance

\( K\psi \) means “the adversary Knows \( \psi \),” i.e. that \( \psi \) can be deduced by observation.

\( P\psi \) means “the adversary admits the Possibility of \( \psi \),” i.e. that \( \psi \) cannot be ruled out by observation.

\( h := 1 \) does establish \( K(h=1) \)
\( h :\in \{0,1\} \) does not establish \( K(h=1) \)
\( h :\in \{0,1\} \) does establish \( P(h=1) \)
\( h :\in \{0,1\} \) does not establish \( P(h=2) \)
\( h :\in \{0,1\}; v :\in \{0,1\} \) does establish \( P(h=1) \)
\( h :\in \{0,1\}; v := h \) does not establish \( P(h=1) \)

You cannot be sure that, once this program fragment is complete, you will consider \( h=1 \) to be possible based on what you have observed.
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\psi$ means “the adversary Knows $\psi$,” i.e. that $\psi$ can be deduced by observation.

$P\psi$ means “the adversary admits the Possibility of $\psi$,” i.e. that $\psi$ cannot be ruled out by observation.

$h:=1$ does establish $K(h=1)$
$h\in\{0,1\}$ does not establish $K(h=1)$
$h\in\{0,1\}$ does establish $P(h=1)$
$h\in\{0,1\}$ does not establish $P(h=2)$
$h\in\{0,1\}; \; v\in\{0,1\}$ does establish $P(h=1)$
$h\in\{0,1\}; \; v:=h$ does not establish $P(h=1)$

You cannot be sure that, once this program fragment is complete, you will consider $h=1$ to be possible based on what you have observed.
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary Knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the Possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h:= 1$ does establish $K(h=1)$
$h:\in \{0,1\}$ does not establish $K(h=1)$
$h:\in \{0,1\}$ does establish $P(h=1)$
$h:\in \{0,1\}$ does not establish $P(h=2)$
$h:\in \{0,1\}; \ v:\in \{0,1\}$ does establish $P(h=1)$
$h:\in \{0,1\}; \ v:= h$ does not establish $P(h=1)$

You cannot be sure that, on completing this program fragment, you will consider $h=1$ to be possible based on what you have observed.
Hoare-triples and weakest preconditions for knowledge and ignorance

$K\Psi$ means “the adversary Knows $\Psi$,” i.e. that $\Psi$ can be deduced by observation.

$P\Psi$ means “the adversary admits the Possibility of $\Psi$,” i.e. that $\Psi$ cannot be ruled out by observation.

$h:=1$ does establish $K(h=1)$

$h:\in\{0,1\}$ does not establish $K(h=1)$

$h:\in\{0,1\}$ does establish $P(h=1)$

$h:\in\{0,1\}$ does not establish $P(h=2)$

$h:\in\{0,1\}; \, v:\in\{0,1\}$ does establish $P(h=1)$

$h:\in\{0,1\}; \, v:=h$ does not establish $P(h=1)$
This is the Refinement Paradox

$K \psi$ means “the adversary Knows $\psi$,” i.e. that $\psi$ can be deduced by observation.

$P \psi$ means “the adversary admits the Possibility of $\psi$,” i.e. that $\psi$ cannot be ruled out by observation.

$h := 1$ does establish $K(h=1)$

$h \in \{0,1\}$ does not establish $K(h=1)$

$h \in \{0,1\}$ does establish $P(h=1)$

$h \in \{0,1\}$ does not establish $P(h=2)$

$h \in \{0,1\}; \ v \in \{0,1\}$ does establish $P(h=1)$

$h \in \{0,1\}; \ v := h$ does not establish $P(h=1)$

You cannot be sure that, once this program fragment is complete, you will consider $h=1$ to be possible based on what you have observed.
Essential principles for scaling up: some refinements must be banned

\[
\text{skip} \notin \quad \text{if } h=0 \text{ then skip else skip fi}
\]

specification

implementation
Essential principles for scaling up: some refinements must be banned

\[
\text{skip} \quad \begin{cases} \text{if } h=0 \text{ then skip else skip fi} \\
\text{if } h=0 \text{ then } \\
\quad \mathbf{vis} \; v \cdot \; v := 0 \\
\text{else} \\
\quad \mathbf{vis} \; v \cdot \; v := 1 \\
\text{fi}
\end{cases}
\]

\begin{itemize}
\item \begin{itemize}
\item \text{skip} \quad \text{if } h=0 \text{ then skip else skip fi}
\item \text{if } h=0 \text{ then } \\
\quad \mathbf{vis} \; v \cdot \; v := 0 \\
\text{else} \\
\quad \mathbf{vis} \; v \cdot \; v := 1 \\
\text{fi}
\end{itemize}
\end{itemize}

\text{[[ ... ]] declares local variables}
Essential principles for scaling up: some refinements must be banned

\[ P() \not\subseteq \text{if } h=0 \text{ then } P() \text{ else } P() \text{ fi } \]
Essential principles for scaling up:
some refinements must be banned

\[ P() \not\subseteq \text{ if } h=0 \text{ then } P() \text{ else } P() \text{ fi } \]

\[ P() \subseteq P_0() \]
Essential principles for scaling up: some refinements must be banned

\[ P() \not\subseteq \begin{cases} \text{if } h=0 \text{ then } P() \text{ else } P() \end{cases} \]

\[ P() \subseteq P_0() \]

\[ P() \subseteq P_1() \]
Essential principles for scaling up: some refinements must be banned

\[ P() \not\subseteq \text{ if } h=0 \text{ then } P() \text{ else } P() \text{ fi } \]

\[ P() \subseteq P_0() \]
\[ P() \subseteq P_1() \]

These procedure bodies could be in a separate module, their source code hundreds of pages away.
Essential principles for scaling up: some refinements must be banned.

\[ P() \not\subseteq \text{if } h=0 \text{ then } P_0() \text{ else } P_1() \text{ fi } \]

\[ P() \subseteq P_0() \]

\[ P() \subseteq P_1() \]

These procedure bodies could be in a separate module, their source code hundreds of pages away.
Essential principles for scaling up: some refinements must be banned

\[ P() \subseteq \text{if } h=0 \text{ then } P_0() \text{ else } P_1() \text{ fi} \]

\[ P() \subseteq P_0() \]
\[ P() \subseteq P_1() \]

These procedure bodies could be in a separate module, their source code hundreds of pages away.
Essential principles for scaling up: *most refinements must be retained*

- All refinements in involving only visible variables.
- All equalities in which no hiddens are assigned to visibles.
- Substitution of equals for equals, in any context.
- All structural refinements based on operators’ general properties.

\[
\begin{align*}
v &: \in \{0,1\} & \subseteq & v := 0 & \text{Allowed: visible only.} \\
h &: \in \{0,1\} & \not\subseteq & h := 0 & \text{Not allowed: not equality.} \\
v &: \in \{0,1\} & = & v &: \in \{h, 1-h\} & \text{Allowed: equals for equals.} \\
(v := h) \cap (v := 1-h) & \subseteq & v := h & \text{Allowed: structural.} \\
v &: \in \{h, 1-h\} & \not\subseteq & (v := h) \cap (v := 1-h) & \text{Not allowed: hidden assigned to visible.}
\end{align*}
\]
Outcome of logical analysis of refinement

... most refinement are retained
... characterise (most of) these in a clear way
... exclude some refinements (as few as possible)
Flying high:

algebra does it without logic
Flying high:

algebra does it without logic

“Here’s a little refinement I prepared earlier.”
An “assertion statement” checks an (important) predicate, and halts program execution if it does not hold:

```
assert pred
```
Algebraic source-level reasoning
An old example

\texttt{assert \ pred}

is just

\texttt{if \ \neg pred \ then \ fi}
Algebraic source-level reasoning
An old example

assert pred
is just \textbf{if} \neg\textit{pred} \textbf{then} fi

assert pre; prog \sqsubseteq prog; assert post

If \textit{pre} holds beforehand then executing \textit{prog} will establish \textit{post}, if termination occurs: in logic \{pre\} prog \{post\}.
Algebraic source-level reasoning
A new example
Algebraic source-level reasoning
A new example: visible and hidden variables

reveal expression
Algebraic source-level reasoning
A new example: visible and hidden variables

reveal expression

Shorthand for assert pre

\{pre\} prog ⊑ reveal expr; prog

If pre holds beforehand, then executing prog might reveal the initial value of expr.
Algebraic source-level reasoning
A new example: *visible* and *hidden* variables

\[
\{ \text{pre} \} \ prog \ \subseteq \ \text{reveal} \ expr; \ prog \\
\{ \text{pre} \} \ prog \ \subseteq \ prog; \ \text{reveal} \ expr
\]

If \( \text{pre} \) holds beforehand, then executing \( \text{prog} \) might reveal the initial value of \( \text{expr} \).
Algebraic source-level reasoning over visible and hidden variables

reveal expression

\(\{v \neq 0\} \quad v := v \times h \quad \sqsubseteq \quad \text{reveal } h; \quad v := v \times h\)

If \(pre\) holds beforehand, then executing \(prog\) might reveal the initial value of \(expr\).

All variables here are natural numbers.
Algebraic source-level reasoning over visible and hidden variables

\[ \{ v \neq 0 \} \; v := v \times h \; \subseteq \; \text{reveal} \; h; \; v := v \times h \]

If \( \text{pre} \) holds beforehand, then executing \( \text{prog} \) might reveal the initial value of \( \text{expr} \).
A calculus of revelations

(1) Replace $E$ with $F$.

\[
\{\text{pre}\} \text{ reveal } E \subseteq \text{ reveal } F
\]

... provided truth of \textit{pre} implies that $F = \mathbb{F}(E)$, for some context $\mathbb{F}$ with only visible variables.
A calculus of revelations

(1) Replace $E$ with $F$.

\[
\{pre\} \text{ reveal } E \sqsubseteq \text{ reveal } F
\]

... provided truth of $pre$ implies that $F = \mathcal{F}(E)$, for some context $\mathcal{F}$ with only visible variables.
A calculus of revelations

(1) Replace $E$ with $F$: examples.

\[
\{pre\} \text{ reveal } E \subseteq \text{ reveal } F
\]

... provided truth of $pre$ implies that $F = F(E)$, for some context $F$ with only visible variables.
A calculus of revelations

(1) Replace $E$ with $F$: examples.

$$\{ pre \} \; \text{reveal} \; E \; \subseteq \; \text{reveal} \; F$$

... provided truth of $pre$ implies that $F = \mathbb{F}(E)$, for some context $\mathbb{F}$ with only visible variables.

Note that $\text{reveal (empty)}$ is just $\text{skip}$. 

18
A calculus of revelations

(1) Replace $E$ with $F$: examples.

$$\{pre\} \text{ reveal } E \quad \sqsubseteq \quad \text{reveal } F$$
$$\{h=0\} \text{ skip} \quad \sqsubseteq \quad \text{reveal } h$$

... provided truth of $pre$ implies that $F = \mathbb{F}(E)$, for some context $\mathbb{F}$ with only visible variables.

Note that $\text{reveal } (\text{empty})$ is just $\text{skip}$.
A calculus of revelations

(1) Replace $E$ with $F$: examples.

\[
\begin{align*}
\{pre\} & \operatorname{reveal} E & \sqsubseteq & \operatorname{reveal} F \\
\{h=0\} & \sqsubseteq & \operatorname{reveal} h
\end{align*}
\]

... provided truth of $pre$ implies that $F = F(E)$, for some context $F$ with only visible variables.

Note that $\operatorname{reveal}(\text{empty})$ is just $\text{skip}$. 
A calculus of revelations

(1) Replace $E$ with $F$: examples.

\[
\begin{align*}
\{ \text{pre} \} \ & \text{reveal } E & \square \ & \text{reveal } F \\
\{ h=0 \} & \text{reveal } h & \square \ & \text{reveal } h \ominus 1
\end{align*}
\]

... provided truth of $\text{pre}$ implies that $F = F(E)$, for some context $\mathbb{F}$ with only visible variables.

Note that $\text{reveal} \ (\text{empty})$ is just skip.
A calculus of revelations

(1) Replace $E$ with $F$: examples.

$\{pre\} \ reveals \ E \ \sqsubseteq \ reveals \ F$

$\{h=0\} \ reveals \ h \ \sqsubseteq \ reveals \ h \oplus 1$

$\ reveals \ h \oplus 1 \ \not\sqsubseteq \ reveals \ h$

... provided truth of $pre$ implies that $F = F(E)$,
for some context $F$ with only visible variables.

Note that $reveal\ (empty)$ is just $skip$. 
A calculus of revelations

(1) Replace $E$ with $F$: examples.

\[
\{pre\} \text{ reveal } E \sqsubseteq \text{ reveal } F \\
\{h=0\} \sqsubseteq \text{ reveal } h \\
\text{reveal } h \sqsubseteq \text{ reveal } h \ominus 1 \\
\text{reveal } h \ominus 1 \nsubseteq \text{ reveal } h \\
\{h>0\} \text{ reveal } h \ominus 1 \sqsubseteq \text{ reveal } h
\]

... provided truth of $pre$ implies that $F = F(E)$, for some context $F$ with only visible variables. Note that $\text{reveal } (\text{empty})$ is just $\text{skip}$. 
A calculus of revelations

(2) Combine $E$ with $F$.

\[
\text{reveal } E; \text{ reveal } F = \text{reveal } (E, F)
\]
A calculus of revelations

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

reveal $E$; reveal $F$

= reveal $(E, F)$

reveal $x \oplus y$; reveal $y \oplus z$
A calculus of revelations

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

\[
\text{reveal } E; \text{ reveal } F \quad = \quad \text{reveal } (E, F)
\]

\[
\text{reveal } x \oplus y; \text{ reveal } y \oplus z
\]

addition mod 2 or, equivalently, exclusive-or
A calculus of revelations

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

\[
\text{reveal } E; \text{ reveal } F = \text{ reveal } (E, F)
\]

\[
= \text{ reveal } x \oplus y; \text{ reveal } y \oplus z
\]

\[
= \text{ reveal } (x \oplus y, y \oplus z)
\]
A calculus of revelations

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

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

$\text{reveal } x \oplus y; \text{ reveal } y \oplus z$
$= \text{ reveal } (x \oplus y, y \oplus z)$
$= \text{ reveal } (x \oplus y, x \oplus z)$
A calculus of revelations

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

\[\text{reveal } E; \text{ reveal } F\]
\[= \text{ reveal } (E, F)\]
\[
\text{reveal } x \oplus y; \text{ reveal } y \oplus z
\]
\[= \text{ reveal } (x \oplus y, y \oplus z)\]
\[= \text{ reveal } (x \oplus y, x \oplus z)\]
\[= \text{ reveal } x \oplus y; \text{ reveal } x \oplus z\]
A calculus of revelations

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

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

\[
\text{reveal } E \ = \ \left[ \texttt{vis } v \cdot v := E \right] \]

A calculus of revelations

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

\[
\text{reveal } E = \left\lvert \left[ \text{vis } v \cdot v := E \right] \right\rvert
\]

Previous approach: harder to manipulate.
A calculus of revelations

\[ wp.(\text{reveal } E).P\phi = P(E=E_0 \wedge \phi) \]
A calculus of revelations

\[ wp.(\text{reveal } (h \mod 2)).P(h=3) \]

\[ wp.(\text{reveal } E).P\phi = P(E=E_0 \land \phi) \]
A calculus of revelations

\[
\begin{align*}
wp. (\text{reveal } (h \mod 2)). P(h=3) &= P((h \mod 2) = (h_0 \mod 2) \land h=3) \\
&= P((h_0 \mod 2) = 1 \land h=3) \\
&= (h \mod 2) = 1 \land P(h=3) \\
&= \text{odd } h \land P(h=3) .
\end{align*}
\]

\[
\begin{align*}
wp. (\text{reveal } E). P\phi &= P(E=E_0 \land \phi)
\end{align*}
\]
A calculus of revelations

\[ wp.(\text{reveal } (h \mod 2)).P(h=3) \]
\[ = P((h \mod 2)=(h_0 \mod 2) \land h=3) \]
\[ = P((h_0 \mod 2)=1 \land h=3) \]
\[ = (h \mod 2)=1 \land P(h=3) \]
\[ = \text{odd } h \land P(h=3). \]

Does \( h : \in \{1,3\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? Yes

\[ wp.(\text{reveal } E).P\phi = P(E=E_0 \land \phi) \]
A calculus of revelations

\[ wp.(\text{reveal } (h \mod 2)).P(h=3) \]
\[ = P((h \mod 2)=(h_0 \mod 2) \land h=3) \]
\[ = P((h_0 \mod 2)=1 \land h=3) \]
\[ = (h \mod 2)=1 \land P(h=3) \]
\[ = \text{odd } h \land P(h=3) . \]

Does \( h: \in \{1,3\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? Yes
Does \( h: \in \{1,5\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? No

\[ wp.(\text{reveal } E).P\phi \quad = \quad P(E=E_0 \land \phi) \]
A calculus of revelations

\[ wp.(\text{reveal } (h \mod 2)).P(h=3) \]
\[ = P((h \mod 2)=(h_0 \mod 2) \land h=3) \]
\[ = P((h_0 \mod 2)=1 \land h=3) \]
\[ = (h \mod 2)=1 \land P(h=3) \]
\[ = \text{odd } h \land P(h=3). \]

Does \( h \in \{1,3\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? Yes

Does \( h \in \{1,5\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? No

Does \( h \in \{2,3\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? No

\[ wp.(\text{reveal } E).P\phi = P(E=E_0 \land \phi) \]
A calculus of revelations

\[ wp. (\text{reveal } (h \mod 2)).P(h=3) \]
\[ = P((h \mod 2) = (h_0 \mod 2) \land h=3) \]
\[ = P((h_0 \mod 2) = 1 \land h=3) \]
\[ = (h \mod 2) = 1 \land P(h=3) \]
\[ = \text{odd } h \land P(h=3). \]

Does \( h : \in \{1,3\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? \hspace{1cm} Yes
Does \( h : \in \{1,5\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? \hspace{1cm} No
Does \( h : \in \{2,3\}; \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? \hspace{1cm} No

Does \( (h:= 1 \sqcap h:= 3); \text{reveal } (h \mod 2) \) establish \( P(h=3) \)? \hspace{1cm} No

\[ wp. (\text{reveal } E).P\phi = P(E=E_0 \land \phi) \]
Flying low:

the logic *justifies* the algebra, creating *beforehand* a library of very small but reusable identities
Flying low:

the logic justifies the algebra, creating beforehand a library of very small but reusable identities.

“Here’s how I prepared that little refinement earlier.”
The Encryption Lemma: a piece of algebraic Lego

\[ hid \ h \cdot \]

\[
\left\lfloor \begin{array}{l}
hid \ h' ; \ h' : \in \{0, 1\} ; \ \text{reveal} \ h \oplus h' \\
\end{array} \right\rfloor
\]

Does this program fragment reveal anything about \( h \)?
The Encryption Lemma: a piece of algebraic Lego

\[ \text{hid } h \cdot \]

\[ \left[ \text{hid } h'; \ h' \in \{0, 1\}; \ \text{reveal } h \oplus h' \right] \]

Does this program fragment reveal anything about \( h \)?

The context of declared variables
The Encryption Lemma: a piece of algebraic Lego

\[
\begin{align*}
\text{hid } h \cdot \\
\left[ \text{hid } h'; \ h' : \in \{0, 1\}; \ \text{reveal } h \oplus h' \right]
\end{align*}
\]

Does this program fragment reveal anything about \( h \)?
The $wp$ semantics validates the 
**construction** of the Lego

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identity</strong></td>
<td>$wp.\text{skip}.\Psi \equiv \Psi$</td>
<td></td>
</tr>
<tr>
<td><strong>Revelation</strong></td>
<td>$wp.(\text{reveal } E).\Psi \equiv [\downarrow E_0=E]\Psi$</td>
<td></td>
</tr>
<tr>
<td><strong>Assign to visible</strong></td>
<td>$wp.(v:= E).\Psi \equiv [e\backslash E] [\downarrow e=E] [v\backslash e] \Psi$</td>
<td></td>
</tr>
<tr>
<td><strong>Choose visible</strong></td>
<td>$wp.(v:\in E).\Psi \equiv (\forall e: E \cdot [\downarrow e\in E] [v\backslash e] \Psi)$</td>
<td></td>
</tr>
<tr>
<td><strong>Assign to hidden</strong></td>
<td>$wp.(h:= E).\Psi \equiv [h\leftarrow E] \Psi$</td>
<td></td>
</tr>
<tr>
<td><strong>Choose hidden</strong></td>
<td>$wp.(h:\in E).\Psi \equiv (\forall e: E \cdot [h\backslash e] [h\leftarrow E] \Psi)$</td>
<td></td>
</tr>
<tr>
<td><strong>Demonic choice</strong></td>
<td>$wp.(S_1 \sqcap S_2).\Psi \equiv wp.S_1.\Psi \land wp.S_2.\Psi$</td>
<td></td>
</tr>
<tr>
<td><strong>Composition</strong></td>
<td>$wp.(S_1; S_2).\Psi \equiv wp.S_1.(wp.S_2.\Psi)$</td>
<td></td>
</tr>
</tbody>
</table>
| **Conditional** | $wp.(\text{if } E \text{ then } S_1 \text{ else } S_2 \text{ fi}).\Psi$  
| & $\equiv E \Rightarrow [\downarrow E] wp.S_1.\Psi \land \neg E \Rightarrow [\downarrow \neg E] wp.S_2.\Psi$ | |
| **Declare visible** | $wp.(\text{VIS } v).\Psi \equiv (\forall e \cdot [v\backslash e] \Psi)$ | |
| **Declare hidden** | $wp.(\text{HID } h).\Psi \equiv (\forall e \cdot [h\leftarrow e] \Psi)$ | |
In this case...

\[
wp.\left[ \text{hid } h'; \ h' : \in \{0, 1\}; \ \text{reveal } h \oplus h' \right].(P \Psi)
\]
In this case...

\[ wp.[\text{hid } h'; \; h' \in \{0, 1\}; \: \text{reveal } h \oplus h'] \cdot (P\Psi) \]

\[ = (\forall e \in \{0, 1\} \cdot [h\setminus e][h \leftarrow \{0, 1\}] \]

\[ P(h \oplus h' = h_0 \oplus h'_0 \land \Psi)) \]

\[ = (\forall e \in \{0, 1\} \cdot [h\setminus e] \]

\[ P(\exists h \in \{0, 1\} \cdot h \oplus h' = h_0 \oplus h'_0 \land \Psi)) \]

\[ = (\forall e \in \{0, 1\} \cdot [h\setminus e]P\Psi) \]
In this case...

\[ wp.|[ \text{hid } h'; \ h':\in \{0, 1\}; \ \text{reveal } h\oplus h' ]|. (P\Psi) \]

\[= (\forall e \in \{0, 1\} \cdot [h\setminus e][h\leftarrow\{0, 1\}] P(h\oplus h' = h_0\oplus h_0' \land \Psi)) \]

\[= (\forall e \in \{0, 1\} \cdot [h\setminus e] P(\exists h \in \{0, 1\} \cdot h\oplus h' = h_0\oplus h_0' \land \Psi)) \]

\[= (\forall e \in \{0, 1\} \cdot [h\setminus e] P\Psi) \]

\[= P\Psi \]

\[= wp.\text{skip.}(P\Psi) \]
The Encryption Lemma:
a piece of algebraic Lego

\[ \text{hid } h \cdot \]

\[ \left[ \text{hid } h' ; \ h' : \in \{0, 1\} ; \ \text{reveal } h \oplus h' \right] \]

= skip

Does this program fragment reveal anything about \( h \)?
The Encryption Lemma:
a piece of algebraic Lego

\[
\begin{align*}
\text{hid} \ h \cdot \\
\left[ \text{hid} \ h' ; \ h' : \in \{0, 1\}; \ \text{reveal} \ h \oplus h' \right]
\end{align*}
\]

= \text{skip}

Does this program fragment reveal anything about \( h \)?
No — it reveals nothing at all.
The Encryption Lemma: a piece of algebraic Lego

hid \( h \cdot \)

\[
\begin{vmatrix}
\text{hid } h'; \ h' : \in \{0, 1\}; \ \text{reveal } h \oplus h'
\end{vmatrix}
\]

= \skip

Does this program fragment reveal anything about \( h \)?
No — it reveals nothing at all.
The Encryption Lemma: a piece of algebraic Lego

\[
\text{hid } h \cdot
\]

\[
[[ \text{hid } h'; h':\in \{0, 1\}; \text{reveal } h \oplus h' ]] = \text{skip}
\]

Does this program fragment reveal anything about \( h \)?
No — it reveals nothing at all.
The Dining Cryptographers

algebraically
The Dining Cryptographers algebraically
The Dining Cryptographers
The Dining Cryptographers

(No)

(No)

(No)

(No)
The Dining Cryptographers
The Dining Cryptographers

Yes

H

(No)

H

(No)

(No)

T

(No)
The Dining Cryptographers

Yes

(No)

H

(No)

H

(No)

T

Yes
The Dining Cryptographers

H
(No)

T
(No)

H
(No)

Yes

(No)

No

Yes
The Dining Cryptographers

Yes

(No)

H

(No)

No

Yes

H

(No)

(Yes)

T
The Dining Cryptographers

(No) (Yes) (No)

H

H

Yes

(No)

T

H

(No)

H

Yes

(Yes)
— Assume $l$ is already set.

$r \in \{0, 1\};$

**reveal** $l \oplus x \oplus r$

All variables are “Boolean,” with $l$ and $r$ being visible. Variable $x$ is a hidden Boolean, and $\oplus$ is “exclusive or.”
Loop Lego
for the Dining Cryptographers

Rename the $r$ variable to $m$ for “middle,”
and add a second component.

— Assume $l$ is already set.

$m \in \{0, 1\};$

reveal $l \oplus x \oplus m$
Loop Lego for the Dining Cryptographers

Rename the $r$ variable to $m$ for “middle,” and add a second component.

— Assume $l$ is already set. $m: \in \{0, 1\}$; reveal $l \oplus x \oplus m$

— Assume $m$ is already set. $r: \in \{0, 1\}$; reveal $m \oplus y \oplus r$
Loop Lego
for the Dining Cryptographers

Hide the middle variable.

— Assume $l$ is already set.
$m: \in \{0, 1\};$
reveal $l \oplus x \oplus m$

— Assume $m$ is already set.
$r: \in \{0, 1\};$
reveal $m \oplus y \oplus r$
Loop Lego for the Dining Cryptographers

Hide the middle variable.

\[
\begin{align*}
\textbf{hid} & \ m \ . \\
\text{— Assume } l & \text{ is already set.} \\
&m : \in \{0, 1\}; \\
\text{reveal} & \ l \oplus x \oplus m \\
\text{— Assume } m & \text{ is already set.} \\
r : \in \{0, 1\}; \\
\text{reveal} & \ m \oplus y \oplus r
\end{align*}
\]
Loop Lego
for the Dining Cryptographers

Squash up.

\[
\begin{array}{l}
\text{hid } m \\
\quad \text{— Assume } l \text{ is already set.}
\end{array}
\]
\[
\begin{array}{l}
m : \in \{0, 1\}; \\
\text{reveal } l \oplus x \oplus m
\end{array}
\]

\[
\begin{array}{l}
\text{— Assume } m \text{ is already set.}
\end{array}
\]
\[
\begin{array}{l}
r : \in \{0, 1\}; \\
\text{reveal } m \oplus y \oplus r
\end{array}
\]
Loop Lego
for the Dining Cryptographers

Squash up.

\[
\begin{align*}
&\left[ \text{hid } m \cdot \\
&m \in \{0, 1\}; \\
&\text{reveal } l \oplus x \oplus m \\
&r \in \{0, 1\}; \\
&\text{reveal } m \oplus y \oplus r \\
\right]
\end{align*}
\]
Bring the \texttt{reveal} commands next to each other.

\[
\begin{array}{c}
\mid[ \texttt{hid} \ m \cdot \\
m : \in \{0, 1\}; \\
\texttt{reveal} \ l \oplus x \oplus m \\
r : \in \{0, 1\}; \\
\texttt{reveal} \ m \oplus y \oplus r \\
]\mid
\end{array}
\]
Loop Lego

Bring the \texttt{reveal} commands next to each other.

\[
\begin{array}{ll}
\text{hiding m} & \cdot \\
m : \in \{0, 1\} & ; \\
\text{reveal } l \oplus x \oplus m & \\
r : \in \{0, 1\} & ; \\
\text{reveal } m \oplus y \oplus r & \\
\end{array}
\]
Loop Lego

Bring the \texttt{reveal} commands next to each other.

\[
\ddagger \left[ \begin{array}{l}
\text{hid } m \\
r \in \{0, 1\}; \\
m \in \{0, 1\}; \\
\text{reveal } l \oplus x \oplus m \\
\text{reveal } m \oplus y \oplus r
\end{array} \right]
\]
Loop Lego

Use *revelation algebra* to alter the expression in the second one.

\[
\begin{align*}
  &| [ \text{hid} \ m \cdot \\
  & \quad r : \in \{0, 1\}; \\
  & \quad m : \in \{0, 1\}; \\
  & \quad \text{reveal} \ l \oplus x \oplus m \\
  & \quad \text{reveal} \ m \oplus y \oplus r \]
\end{align*}
\]
Use revelation algebra to alter the expression in the second one.
Loop Lego

Use revelation algebra to alter the expression in the second one.

\[
\begin{align*}
| & \text{hid } m \\ r \in \{0, 1\}; \\
& m \in \{0, 1\}; \\
& \text{reveal } l \oplus x \oplus m \\
& \text{reveal } l \oplus x \oplus y \oplus r
\end{align*}
\]
Loop Lego

Move the non-$m$-using commands out of the local scope.

\[
\mid \begin{align*}
\text{hid} & \; m \cdot \\
r &: \in \{0, 1\}; \\
m &: \in \{0, 1\}; \\
\text{reveal} & \; l \oplus x \oplus m \\
\text{reveal} & \; l \oplus x \oplus y \oplus r
\end{align*}
\mid
\]
Loop Lego

Move the non-\(m\)-using commands out of the local scope.

\[
\begin{align*}
| & [ \text{hid } m \cdot \\
& r : \in \{0, 1\}; \\
& m : \in \{0, 1\}; \\
& \text{reveal } l \oplus x \oplus m \\
& \text{reveal } l \oplus x \oplus y \oplus r \\
|]
\end{align*}
\]
Loop Lego

Move the non-$m$-using commands out of the local scope.

\[
\begin{align*}
  r \in & \{0, 1\}; \\
  |[ & \text{hid } m \cdot \\
  & m \in \{0, 1\}; \\
  & \text{reveal } l \oplus x \oplus m \\
  ]| \\
  & \text{reveal } l \oplus x \oplus y \oplus r
\end{align*}
\]
Loop Lego

Appeal to the *Encryption Lemma*.

\[
\begin{align*}
 r & \in \{0, 1\}; \\
 \| \begin{array}{c}
 \text{hid} \ m \\
 m & \in \{0, 1\}; \\
 \text{reveal} \ l \oplus x \oplus m \\
 \end{array} \\
\| \\
 \text{reveal} \ l \oplus x \oplus y \oplus r
\end{align*}
\]
Loop Lego

Appeal to the *Encryption Lemma.*

\[ r \in \{0, 1\}; \]
\[ \mid [ \text{hid } m \cdot \]
\[ m \in \{0, 1\}; \]
\[ \text{reveal } l \oplus x \oplus m \]
\[ ]\]
\[ \text{reveal } l \oplus x \oplus y \oplus r \]
Loop Lego

Appeal to the *Encryption Lemma*.

\[ r \in \{0, 1\}; \]

\[ \text{reveal } l \oplus x \oplus y \oplus r \]

\[ \text{skip}; \]

\[ \text{reveal } l \oplus x \oplus y \oplus m \]

\[ \text{reveal } l \oplus x \oplus y \oplus r \]
Loop Lego

Appeal to the *Encryption Lemma*.

\[
\begin{align*}
 r \in \{0, 1\}; \\
\text{skip; } \\
\text{reveal } l \oplus x \oplus y \oplus r
\end{align*}
\]
Loop Lego

Squash up, and
recall where we started.

\[ r \in \{0, 1\}; \]

\textbf{reveal} \( l \oplus x \oplus y \oplus r \)
Loop Lego

Squash up, and recall where we started.

— Assume $l$ is already set.
$r \in \{0, 1\};$
reveal $l \oplus x \oplus m$

— Assume $m$ is already set.
$r \in \{0, 1\};$
reveal $m \oplus y \oplus r$

— Assume $l$ is already set.
$r \in \{0, 1\};$
reveal $l \oplus x \oplus y \oplus r$
Loop Lego

Squash up, and recall where we started.

— Assume \( l \) is already set.
\[ r \in \{0, 1\}; \]
reveal \( l \oplus x \oplus m \)

— Assume \( m \) is already set.
\[ r \in \{0, 1\}; \]
reveal \( m \oplus y \oplus r \)
Loop Lego

Squash up, and recall where we started.

--- Assume \( l \) is already set.
\( r \in \{0, 1\}; \)

|\[ \begin{array}{c}
\text{hid} \ m \cdot \\
\text{— Assume} \ m \text{ is already set.} \\
\end{array} \]

\( m \in \{0, 1\}; \)

\text{reveal} \ l \oplus x \oplus m

--- Assume \( m \) is already set.
\( r \in \{0, 1\}; \)

\text{reveal} \ m \oplus y \oplus r

--- Assume \( l \) is already set.
\( r \in \{0, 1\}; \)

\text{reveal} \ l \oplus x \oplus y \oplus r
Loop Lego

Squash up, and recall where we started.

— Assume $l$ is already set.
$r \in \{0, 1\}$;
reveal $l \oplus x \oplus m$

— Assume $m$ is already set.
$r \in \{0, 1\}$;
reveal $m \oplus y \oplus r$
Loop Lego

Squash up, and recall where we started.

| hid m ·
| — Assume l is already set.
| m:∈ {0, 1};
| reveal l ⊕ x ⊕ m
| — Assume m is already set.
| r:∈ {0, 1};
| reveal m ⊕ y ⊕ r

— Assume l is already set.
l:∈ {0, 1};
reveal l ⊕ x ⊕ y ⊕ r
This is how we will make the loop that treats an arbitrary number of cryptographers.

— Assume $l$ is already set.
$r: \in \{0, 1\}$;
reveal $l \oplus x \oplus m$

— Assume $m$ is already set.
$r: \in \{0, 1\}$;
reveal $m \oplus y \oplus r$

Use this to abbreviate exclusive-or of that range.
This is how we will make the loop that treats an arbitrary number of cryptographers.

— Assume $l$ is already set.
$r \in \{0, 1\};$
reveal $l \oplus x \oplus m$

— Assume $m$ is already set.
$r \in \{0, 1\};$
reveal $m \oplus y \oplus r$

Use this to abbreviate exclusive-or of that range.
This is how we will make the loop that treats an arbitrary number of cryptographers.

— Assume \( l \) is already set.
\( r \in \{0, 1\}; \)
reveal \( l \oplus x \oplus m \)

— Assume \( m \) is already set.
\( r \in \{0, 1\}; \)
reveal \( m \oplus y \oplus r \)

Use this to abbreviate exclusive-or of that range.
Closing the circle

reveal $x[0, N]$
Closing the circle

$x[0, N)$

reveal $x[0, N]$
Closing the circle

Parity of $x[0,N]$ revealed — but nothing else.
Closing the circle

Parity of $x[0, N]$ revealed — but nothing else.
Closing the circle

\[ \text{hide } l, r \cdot \\
\quad l \in \{0, 1\}; \\
\quad r \in \{0, 1\}; \\
\quad \text{reveal } l \oplus x[0, N) \oplus r; \\
\quad \text{reveal } r \oplus x[N] \oplus l \\
\]

Parity of $x[0,N]$ revealed — but nothing else.
Closing the circle

Parity of \( x[0, N] \) revealed — but nothing else.

\[
\begin{array}{c}
\text{hidden } l, r \cdot \\
\begin{array}{l}
l \in \{0, 1\}; \\
r \in \{0, 1\};
\end{array} \\
\text{reveal } l \oplus x[0, N] \oplus r; \\
\text{reveal } r \oplus x[N] \oplus l
\end{array}
\]
How do we treat loops?

First example
How do we treat loops?

First example

...and only, in this talk
Iterated revelation

If

\[ \text{if } h > 0 \text{ then } h := h - 1 \text{ fi} \]

reveals whether \( h \) was non-zero initially,
Iterated revelation

If

\[
\text{if } h > 0 \text{ then } h := h - 1 \text{ fi}
\]

reveals whether \( h \) was non-zero initially,
Iterated revelation

then does the loop

\begin{verbatim}
while h > 1 do h := h - 2 od
\end{verbatim}

reveal the initial value of \( h \div 2 \)?
Does it reveal more than that?
then does the loop

\[ \textbf{while } h > 1 \textbf{ do } h := h - 2 \textbf{ od} \]

reveal the initial value of \( h \div 2 \)? Does it reveal more than that?
Iterated revelation

then does the loop

\[
\text{while } h > 1 \text{ do } h := h - 2 \text{ od}
\]

reveal the initial value of \( h \div 2 \)?
Does it reveal more than that?
Loops are fixed-points, and unique if terminating

\[
\text{while } h > 1 \text{ do } \begin{align*}
    h &:= h - 2 \\
    \text{reveal } h &:= h \div 2; \\
    h &:= h \mod 2
\end{align*}
\text{od}
\]
Loops are fixed-points, and unique if terminating

\[
\text{while } h > 1 \text{ do } \quad \text{?} \quad = \quad \text{reveal } h \div 2; \quad \text{h := h mod 2} \\
\text{od}
\]
Loops are fixed-points, and unique if terminating

\[
\textbf{while } h > 1 \textbf{ do } \begin{align*}
& h := h - 2 \\
& \text{reveal } h \div 2; \quad h := h \mod 2
\end{align*} \textbf{ od}
\]
Loops are fixed-points, and unique if terminating

\[
\text{while } h > 1 \text{ do } = \text{reveal } h \div 2; \\
\quad h := h - 2; \quad h := h \mod 2 \text{ od}
\]

just when

\[
\text{if } h > 1 \text{ then } = \text{reveal } h \div 2; \\
\quad h := h - 2; \quad h := h \mod 2 \text{ while } h > 1 \text{ do} \\
\quad h := h - 2; \quad \text{ od fi}
\]
Loops are fixed-points, and unique if terminating

\begin{align*}
\textbf{while } h > 1 & \textbf{ do } \quad \textbf{reveal } h \div 2; \\
& h := h - 2 \quad \textbf{reveal } h \mod 2 \\
\textbf{od }
\end{align*}

\begin{align*}
\textbf{just when }
\textbf{if } h > 1 & \textbf{ then } \quad \textbf{reveal } h \div 2; \\
& h := h - 2; \quad \textbf{reveal } h \mod 2 \\
& \textbf{reveal } h \div 2; \quad h := h \mod 2 \\
\textbf{fi }
\end{align*}
Calculate for equality

```plaintext
if \( h > 1 \) then
  \( h := h - 2; \)
  reveal \( h \div 2; \)
  reveal \( h \div 2; \)
  \( h := h \mod 2 \)
fi
```
Calculate for equality

\[
\text{if } h > 1 \text{ then }
\]
\[
h := h - 2;
\]
\[
\text{reveal } h \div 2;
\]
\[
h := \text{mod } h \text{ mod } 2
\]
\[
\text{fi}
\]
Calculate for equality

if $h > 1$ then
  $h := h - 2$;
  reveal $h \div 2$;
  $h := h \mod 2$
fi

reveal $h \div 2$;
$h := h \mod 2$
Calculate for equality

if $h > 1$ then
    reveal $h \div 2$;
    $h := h - 2$;
    $h := h \mod 2$
fi
Calculate for equality

\[
\text{reveal } h/2; \\
h := h \mod 2
\]

\[
\text{if } h > 1 \text{ then} \\
\quad \text{reveal } (h-2)/2; \\
\quad h := h-2; \\
\quad h := h \mod 2
\]
\fi
Calculate for equality

\[
\text{reveal } h \div 2; \\
\text{if } h > 1 \text{ then} \\
\text{reveal } (h - 2) \div 2; \\
\text{fi} \\
h := h \text{ mod } 2
\]
Calculate for equality

\[
\begin{align*}
\text{if } h > 1 & \text{ then} \\
& \text{reveal } (h - 2) \div 2 \\
\text{fi;}
\end{align*}
\]

\[
h := h \mod 2
\]
Calculate for equality

\[
\text{reveal } h \div 2; \\
h := h \mod 2
\]

if \( h > 1 \) then

\[
\text{reveal } (h - 2) \div 2
\]

fi;

\[
h := h \mod 2
\]
Calculate for equality

\[
\begin{align*}
\text{if } h > 1 \text{ then} & \quad \{ h > 1 \} \\
& \quad \text{reveal } (h - 2) \div 2 \\
\text{else} & \quad \{ h \leq 1 \}
\end{align*}
\]

fi;

\[
\begin{align*}
h & := h \mod 2
\end{align*}
\]
Calculate for equality

\[
\begin{align*}
\text{if } h &> 1 \text{ then } \\
\{ h > 1 \} & \\
\text{reveal } (h - 2) \div 2 & \\
\text{else } & \\
\{ h \leq 1 \} & \\
\text{reveal } h \div 2 & \\
\text{fi;} & \\
h &:= h \mod 2
\end{align*}
\]

\[\text{reveal } h \div 2; \quad h := h \mod 2\]
Calculate for equality

\[
\text{reveal } h \div 2; \\
h := h \mod 2
\]

if \( h > 1 \) then
   \{ h > 1 \}
   \text{reveal } h \div 2

else
   \{ h \leq 1 \}
   \text{reveal } h \div 2

fi;

\( h := h \mod 2 \)
Calculate for equality

```plaintext
reveal h ÷ 2;
h := h mod 2

if h > 1 then
    reveal h ÷ 2
else
    reveal h ÷ 2
fi;
h := h mod 2
```
Calculate for equality

\[
\begin{align*}
\text{if } h > 1 \text{ then} & \quad \text{reveal } h \div 2 \\
\text{else} & \quad \text{reveal } h \div 2 \\
\text{fi; } & \quad h := h \mod 2
\end{align*}
\]
Calculate for equality

\[
\text{reveal } h \div 2;
\]
\[
h := h \mod 2
\]
Calculate for equality

\[ h := h \mod 2 \]

reveal \( h \div 2 \);
\[ h := h \mod 2 \]

reveal \( h > 1 \);
reveal \( h \div 2 \);
\[ h := h \mod 2 \]
Calculate for equality

\[ h := h \mod 2 \]

reveal \( h \div 2 \);
\[ h := h \mod 2 \]
We have the equality

\[
\text{while } h > 1 \text{ do }
\begin{align*}
    h &:= h - 2 \\
\end{align*}
\text{od}
\]

\[
\text{reveal } h \div 2; \\
    h := h \mod 2
\]
We have the equality
	hen does the loop

\[
\text{while } h > 1 \text{ do } \begin{array}{c}
\text{reveal } h \div 2; \\
\text{h} := h \mod 2
\end{array}
\]

reveal the initial value of \( h \div 2 \)?
Does it reveal more than that?
We have the equality

\[ \text{while } h > 1 \text{ do } \]
\[ h := h - 2 \]
\[ \text{od} \]

Does the loop reveal the initial value of \( h \div 2 \)? Yes.

Does it reveal more than that?
We have the equality

then does the loop

\[ \text{while } h > 1 \text{ do } \]
\[ h := h - 2 \]
\[ \text{od} \]

reveal the initial value of \( h \div 2 \)? Yes.
Does it reveal more than that? No.
We have the equality

\[ \text{while } h > 1 \text{ do } \]
\[ h := h - 2 \]
\[ \text{od} \]

then does the loop

reveal the initial value of \( h \div 2 \)? Yes.
Does it reveal more than that? No.

Actually this is quite a delicate question: what’s “more”?
Quite a delicate question...

All those delicate questions are answered for the loop, automatically, if they are answerable for its straight-line simplification...

...and in any context.

This is why the refinement-based approach has been so popular elsewhere, for >35 years.
Quite a delicate question...

All those delicate questions are answered for the loop, automatically, if they are answerable for its straight-line simplification...

...and in any context.

This is why the refinement-based approach has been so popular elsewhere, for >35 years.

Program development by stepwise refinement. Wirth, CACM 1971.
Quite a delicate question...

All those delicate questions are answered for the loop, automatically, if they are answerable for its straight-line simplification...

...and in any context.

This is why the refinement-based approach has been so popular elsewhere, for >35 years.

This is why we need it here too.

*Program development by stepwise refinement.* Wirth, CACM 1971.
How do we treat loops?

Second example:

unboundedly many cryptographers
How do we treat loops?

Second example:

unboundedly many cryptographers
Conclusions

How do we treat loops?

Second example:

unboundedly many cryptographers
Conclusions

We have

... a firm semantics
... a supple algebra
... easy automation possible in a simple-minded style
... integration with traditional reasoning
... examples (almost) never done before
Conclusions

... a firm semantics
   (not given in the talk)

... a supple algebra
... easy automation possible in a simple-minded style
... integration with traditional reasoning
... examples (almost) never done before
Conclusions

... a firm semantics
   (not given in the talk)
... a supple algebra
   (illustrated above)

... easy automation possible in a simple-minded style
... integration with traditional reasoning
... examples (almost) never done before
Conclusions

... a firm semantics  
  (not given in the talk)
... a supple algebra  
  (illustrated above)
... easy automation possible in a simple-minded style  
  (suits the hacker mentality)

... integration with traditional reasoning  
... examples (almost) never done before
Conclusions

... a firm semantics
  (not given in the talk)
... a supple algebra
  (illustrated above)
... easy automation possible in a simple-minded style
  (suits the hacker mentality)
... integration with traditional reasoning
  (scale-up principles)
... examples (almost) never done before
Conclusions

... a firm semantics
  (not given in the talk)
... a supple algebra
  (illustrated above)
... easy automation possible in a simple-minded style
  (suits the hacker mentality)
... integration with traditional reasoning
  (scale-up principles)
... examples (almost) never done before
  (unbounded Dining Cryptographers)
The Dining Cryptographers, for arbitrary $N$

$$
\begin{array}{l}
[ \textbf{hid} \ l, m, r : \{0, 1\} \cdot \\\n\quad l : \in \{0, 1\}; \ m := l; \\\n\quad n := N; \\quad \text{repeat} \\\n\quad \quad n := n - 1; \\\n\quad \quad r : \in \{0, 1\}; \\\n\quad \quad \textbf{reveal} \ m \oplus x[ n ] \oplus r; \\\n\quad \quad m := r \\quad \text{until } n = 0; \\\n\quad \textbf{reveal} \ r \oplus x[ N ] \oplus l
\end{array}
$$
The Dining Cryptographers, for arbitrary $N$

$\begin{align*}
\lbrack & \text{hid } l, m, r : \{0, 1\}. \\
& l : \in \{0, 1\}; \quad m := l; \\
& n := N; \\
& \text{repeat} \\
& \quad n := n - 1; \\
& \quad r : \in \{0, 1\}; \\
& \quad \text{reveal } m \oplus x[n] \oplus r; \\
& \quad m := r \\
& \text{until } n = 0; \\
& \text{reveal } r \oplus x[N] \oplus l
\end{align*}$

By using model checking techniques one can verify DC up to 8 and more cryptographers with resulting state spaces for the model of about $10^{36}$ states, and considerably more cryptographers if the representation of the model is optimised.

Pucella, SIGACT News, 2007
The Dining Cryptographers, for arbitrary $N$

$\text{repeat} \hspace{1cm} n := n - 1; \hspace{1cm} n := N;$

$\text{[hid l, m, r : \{0, 1\}.} \hspace{1cm} l \in \{0, 1\}; \hspace{1cm} m := l; \hspace{1cm} N]$

$\text{until } n = 0; \hspace{1cm} \text{reveal } x[0, N]$\hspace{1cm} \text{reveal } m \oplus x[n] \oplus r; \hspace{1cm} m := r \oplus x[N] \oplus l$

By using model checking techniques one can verify DC up to 8 and more cryptographers with resulting state spaces for the model of about $10^{36}$ states, and considerably more cryptographers if the model is optimised.

$Pucella, \hspace{1cm} \text{SIGACT News, 2007}$

To the best of the author's knowledge, this is the first machine-verified proof of privacy of the Dining Cryptographers protocol for an unbounded number of participants and a quantitative metric for privacy.

$Coble, \hspace{1cm} 2008$
Making a **repeat** loop
via a fixed-point argument

— Assume $l$ is already set.
$r \in \{0, 1\}$;
**reveal** $l \oplus x[0, N) \oplus r$
“Reasonable” guess for a suitable loop

\[
\begin{align*}
n &:= N; \\
l &\in \{0, 1\}; \\
\text{repeat} & \\
\quad & n := n - 1; \\
\quad & r := r; \\
\quad & \text{reveal } l \oplus x[n] \oplus r; \\
\quad & l := r \\
\text{until } & n = 0
\end{align*}
\]

\[
\begin{align*}
l &\in \{0, 1\}; \\
r &\in \{0, 1\}; \\
\text{reveal } & l \oplus x[0, N) \oplus r
\end{align*}
\]
But not quite right, because it overwrites \( l \)

\[
\begin{align*}
n &= N; \\
l &\in \{0, 1\}; \\
\text{repeat} \\
& \quad n := n - 1; \\
& \quad r \in \{0, 1\}; \\
& \quad \text{reveal } l \oplus x[n] \oplus r; \\
& \quad l := r \\
\text{until } n = 0
\end{align*}
\]
So introduce a temporary variable $m$

\[
\begin{align*}
n &:= N; \\
l &\in \{0, 1\}; \\
m &:= l; \\
\text{repeat} & \\
\quad n &:= n - 1; \\
r &\in \{0, 1\}; \\
\text{reveal } m &\oplus x[n] \oplus r; \\
m &:= r \\
\text{until } n = 0
\end{align*}
\]
So introduce a temporary variable $m$

\[
\begin{align*}
n &:= N; \\
l &\in \{0, 1\}; \\
m &:= l;
\end{align*}
\]

\textbf{repeat}

\[
\begin{align*}
n &:= n - 1; \\
r &\in \{0, 1\}; \\
\text{reveal } m &\oplus x[n] \oplus r; \\
m &:= r
\end{align*}
\]

\textbf{until } $n=0$

\[
\begin{align*}
l &\in \{0, 1\}; \\
r &\in \{0, 1\}; \\
\text{reveal } l \oplus x[0, N) \oplus r
\end{align*}
\]
Concentrate on the repeat on its own

\( n := N; \)
\( l \in \{0, 1\}; \)
\( m := l; \)

repeat
\( n := n - 1; \)
\( r \in \{0, 1\}; \)
reveal \( m \oplus x[n] \oplus r; \)
\( m := r \)
until \( n = 0 \)
Concentrate on the \texttt{repeat} on its own

\begin{verbatim}
repeat
  \texttt{n:=n-1;}
  \texttt{r:\in\{0,1\};}
  \texttt{reveal \quad m \oplus x[n] \oplus r;}
  \texttt{m:=r}
until \texttt{n=0}
\end{verbatim}

\begin{itemize}
  \item Assume \texttt{n}>0
  \texttt{r:\in\{0,1\};}
  \texttt{reveal \quad m \oplus x[0,n] \oplus r;}
  \texttt{m:=r;}
  \texttt{n:=0}
\end{itemize}
Apply the fixed-point functional

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]
\[ m := r; \]
\[ \text{if } n > 0 \text{ then} \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
\[ \text{fi} \]
Apply the fixed-point functional

\[
\begin{align*}
n &:= n - 1; \quad \text{loop body} \\
r &\in \{0, 1\}; \\
\text{reveal } m \oplus x[n] \oplus r;
\end{align*}
\]

\[
\begin{align*}
\text{if } n > 0 \text{ then} \\
\text{reveal } m \oplus x[0, n) \oplus r; \\
m &:= r; \\
n &:= 0 \quad \text{loop specification}
\end{align*}
\]
It must be a fixed-point

```
\textbf{loop specification}
\begin{align*}
n &:= 0; \\
\text{if } n > 0 \text{ then} & \\
\begin{align*}
r &\in \{0, 1\}; \\
\text{reveal } m &\oplus x[0, n] \oplus r; \\
\end{align*} \\
\end{align*}
\textbf{loop body}
\begin{align*}
n &:= n - 1; \\
r &\in \{0, 1\}; \\
\text{reveal } m &\oplus x[n] \oplus r;
\end{align*}
\textbf{loop conditional}
```
It must be a fixed-point

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]
\[ m := r; \]
\[ \text{if } n > 0 \text{ then} \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
\[ \text{fi} \]

\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]  

loop specification
We calculate as follows...
Expand the if

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]
\[ m := r; \]
\textbf{if } n > 0 \textbf{ then}
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
\textbf{fi}
We calculate as follows...
Expand the if

\[
\text{if } n > 0 \text{ then } \\
n := n - 1; \\
r \in \{0, 1\}; \\
\text{reveal } m \oplus x[n] \oplus r; \\
m := r; \\
r \in \{0, 1\}; \\
\text{reveal } m \oplus x[0, n) \oplus r; \\
m := r; \\
n := 0
\]
fi
Use \( n=1 \) in the \texttt{else}

\begin{align*}
\textbf{if} \ n>1 \ \textbf{then} & \\
& n := n-1; \\
& r \in \{0, 1\}; \\
& \texttt{reveal} \ m \oplus x[n] \oplus r; \\
& m := r; \\
& r \in \{0, 1\}; \\
& \texttt{reveal} \ m \oplus x[0, n) \oplus r; \\
& m := r; \\
& n := 0 \\
\textbf{else \ldots} & \\
\end{align*}
Use $n=1$ in the else

if $n>1$ then
  $n := n - 1$;
  $r \in \{0, 1\}$;
  reveal $m \oplus x[n] \oplus r$;
  $m := r$;
  $r \in \{0, 1\}$;
  reveal $m \oplus x[0, n) \oplus r$;
  $m := r$;
  $n := 0$
else ...

... else

$r \in \{0, 1\}$;
reveal $m \oplus x[n] \oplus r$;
$m := r$

$n := n - 1$;
fi
Introduce local hidden $r'$

\[
\begin{aligned}
&\text{if } n > 1 \text{ then} \\
&\quad n := n - 1; \\
&\quad r \in \{0, 1\}; \\
&\quad \text{reveal } m \oplus x[n] \oplus r; \\
&\quad m := r; \\
&\quad r \in \{0, 1\}; \\
&\quad \text{reveal } m \oplus x[0, n) \oplus r; \\
&\quad m := r; \\
&\quad n := 0 \\
&\text{else . . .}
\end{aligned}
\]
Introduce local hidden $r'$

if $n > 1$ then
  $n := n - 1$;
else
  $r \in \{0, 1\}$;
  reveal $m \oplus x[0] \oplus r$;
  $m := r$;
  $n := 0$
fi
Reorder; eliminate superfluous assignment

\[
\text{if } n > 1 \text{ then} \\
\quad n := n - 1; \\
\quad r := \{0, 1\}; \\
\quad \text{reveal } m \oplus x[0] \oplus r; \\
\quad m := r; \\
\quad n := 0 \\
\text{fi}
\]

\[
\text{else} \\
\quad r := \{0, 1\}; \\
\quad \text{reveal } m \oplus x[n] \oplus r'; \\
\quad r' := \{0, 1\}; \\
\quad \text{reveal } m \oplus x[0, n] \oplus r; \\
\quad m := r; \\
\quad n := 0 \\
\text{else} \ldots
\]
Reorder; eliminate superfluous assignment

\[ r \in \{0, 1\}; \]

if \( n > 1 \) then

\[ n := n - 1; \]

\[ [\text{hid} \ r'] \cdot \]

\[ r' \in \{0, 1\}; \]

reveal \( m \oplus x[n] \oplus r' \);

reveal \( m \oplus x[0, n) \oplus r; \)

else...

... else

reveal \( m \oplus x[0] \oplus r; \)

fi

\[ m := r; \]

\[ n := 0 \]
Reveal calculus; use $n=1$ in else

$r \in \{0, 1\}$;

if $n > 1$ then

$n := n - 1$;

$[ [ \text{hid } r' \cdot$

$r' \in \{0, 1\}$;

reveal $m \oplus x[n] \oplus r'$;

reveal $r' \oplus x[0, n) \oplus r$

$] [$

else . . .

\ldots$ else

reveal $m \oplus x[0] \oplus r$

fi

$m := r$;

$n := 0$
Reveal calculus; use $n=1$ in else

$r \in \{0, 1\}$;  
$m := r$;

if $n > 1$ then

$n := n - 1$;

[r']  

[r'] \in \{0, 1\};

reveal $m \oplus x[n] \oplus r'$;

reveal $r' \oplus x[0, n) \oplus r$

else . . .
Reorder; remove assignment to \( n \)

\[
\begin{align*}
r &\in \{0, 1\}; \\
\text{if } n &> 1 \text{ then} \\
    n &:= n - 1; \\
    n &:= 0 \\
\end{align*}
\]

\[
\begin{align*}
    r' &\in \{0, 1\}; \\
    \text{reveal } m &\oplus x[n] \oplus r'; \\
    \text{reveal } m &\oplus x[0, n] \oplus r \\
\end{align*}
\]

\[
\begin{align*}
    \text{else . . . }
\end{align*}
\]
Reorder; remove assignment to $n$

$r\in \{0, 1\};$

if $n > 1$ then

\[
\text{reveal } m \oplus x[0, n] \oplus r
\]

fi

\[
m := r;
\]

\[
n := 0
\]

else

\[
\text{reveal } m \oplus x[0, n) \oplus r
\]

else . . .
... else

reveal \( m \oplus x[0, n) \oplus r \)

fi

\( m := r; \)

\( n := 0 \)

\( r \in \{0, 1\}; \)

if \( n > 1 \) then

\[\begin{align*}
\text{hid } r' .
\end{align*}\]

\( r' \in \{0, 1\}; \)

reveal \( m \oplus x[n-1] \oplus r'; \)

reveal \( m \oplus x[0, n) \oplus r \)

else . . .
\[ r \in \{0, 1\}; \]
\[ \text{if } n > 1 \text{ then} \]
\[ \begin{array}{l}
| [ \text{hid } r'. \]
\[ r' \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n-1] \oplus r'; \]
\[ ]| \]
\[ \text{reveal } m \oplus x[0, n) \oplus r \]
\[ m := r; \]
\[ n := 0 \]
Encryption lemma

\[ r \in \{0, 1\}; \]
\[ \text{if } n > 1 \text{ then} \]
\[ \text{hid } r'. \]
\[ r' \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n-1] \oplus r' \]
\[ \text{fi}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
Encryption lemma

\[ r \in \{0, 1\}; \]

\textbf{reveal} \hspace{1em} m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
Variable $n$ is visible

\[
\begin{align*}
\text{if } n > 1 & \text{ then skip fi; } \\
\text{reveal } m \oplus x[0, n) \oplus r; \\
m & := r; \\
n & := 0
\end{align*}
\]
Variable $n$ is visible

$r \in \{0, 1\};$
\textbf{reveal } $m \oplus x[0, n) \oplus r;$
$m := r;$
$n := 0$
And we started with...

\[
\begin{align*}
  r &\in \{0, 1\}; \\
  \text{reveal} & \quad m \oplus x[0, n) \oplus r; \\
  m &:= r; \\
  n &:= 0
\end{align*}
\]
And we started with...

\[
\begin{align*}
\text{Assume } n &> 0 \\
\begin{array}{l}
r \in \{0, 1\}; \\
\text{reveal } m \oplus x[0, n) \oplus r; \\
m := r; \\
n := 0
\end{array}
\end{align*}
\]
The complete program, with the circle closed

\[
\begin{array}{l}
\text{repeat} \\
\quad n := n - 1; \\
\quad r := \{0, 1\}; \\
\quad \text{reveal } m \oplus x[n] \oplus r; \\
\quad m := r \\
\text{until } n = 0; \\
\text{reveal } r \oplus x[N] \oplus l
\end{array}
\]
Conclusions

We have

... a firm semantics
... a supple algebra
... easy automation in a simple-minded style
... integration with traditional reasoning
... examples (almost) never done before
Cryptographers’ derivation: reprise
Making a **repeat** loop
via a fixed-point argument

— Assume \( l \) is already set.
\( r \in \{0, 1\} \);
reveal \( l \oplus x[0, N) \oplus r \)
“Reasonable” guess for a suitable loop

\[
\begin{align*}
n &:= N; \\
l &\in \{0, 1\}; \\
\text{repeat} & \\
& \quad n := n - 1; \\
& \quad r \in \{0, 1\}; \\
& \quad \text{reveal } l \oplus x[n] \oplus r; \\
& \quad l := r \\
\text{until } n = 0
\end{align*}
\]

\[
\begin{align*}
l &\in \{0, 1\}; \\
r &\in \{0, 1\}; \\
\text{reveal } l \oplus x[0, N) \oplus r
\end{align*}
\]
But not *quite* right, because it overwrites $l$

\[
\begin{align*}
n &:= N; \\
l &\in \{0, 1\}; \\
\text{repeat} \\
\ &\quad n := n - 1; \\
\ &\quad r \in \{0, 1\}; \\
\ &\quad \text{reveal } l \oplus x[n] \oplus r; \\
\ &\quad l := r \\
\text{until } n = 0
\end{align*}
\]
So introduce a temporary variable $m$

\[
\begin{align*}
  n & := N; \\
  l & \in \{0, 1\}; \\
  m & := l; \\
  \text{repeat} \\
  & \quad n := n - 1; \\
  & \quad r \in \{0, 1\}; \\
  & \quad \text{reveal } m \oplus x[n] \oplus r; \\
  & \quad m := r \\
  \text{until } n=0
\end{align*}
\]

\[l \in \{0, 1\}; \quad r \in \{0, 1\}; \quad \text{reveal } l \oplus x[0, N) \oplus r\]
So introduce a temporary variable $m$

$n := N;$
\[ l \in \{0, 1\}; \]
\[ m := l; \]
\[ \text{repeat} \]
\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal} \ m \oplus x[n] \oplus r; \]
\[ m := r \]
\[ \text{until } n = 0 \]

\[ l \in \{0, 1\}; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal} \ l \oplus x[0, N] \oplus r \]
Concentrate on the repeat on its own

\[
n := N; \\
l : \in \{0, 1\}; \\
m := l;
\]

**repeat**

\[
n := n - 1; \\
r : \in \{0, 1\};
\]

**reveal** \( m \oplus x[n] \oplus r; \)

\[
m := r
\]

**until** \( n=0 \)
Concentrate on the \textbf{repeat} on its own

\begin{verbatim}
repeat
  \textcolor{blue}{n:= n-1;}
  \textcolor{red}{r\in \{0,1\};}
  \textcolor{green}{reveal m \oplus x[n] \oplus r;}
  \textcolor{magenta}{m:= r}
\textcolor{green}{until n=0}
\end{verbatim}
Concentrate on the \textbf{repeat} on its own

\textbf{repeat}
\begin{align*}
n &:= n-1; \\
r &\in \{0, 1\}; \\
\text{reveal} \quad m \oplus x[n] \oplus r; \\
m &:= r
\end{align*}
\textbf{until} \quad n=0

\begin{itemize}
\item Assume \( n>0 \)
\item \( r\in \{0, 1\} \);
\item \textbf{reveal} \( m \oplus x[0, n) \oplus r; \)
\item \( m := r; \)
\item \( n := 0 \)
\end{itemize}
Apply the fixed-point functional

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]
\[ m := r; \]
\[ \text{if } n > 0 \text{ then} \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
\[ \text{fi} \]
Apply the fixed-point functional

\[
\begin{align*}
  n &:= n - 1; \quad \text{loop body} \\
  r &\in \{0, 1\}; \\
  \text{reveal } m \oplus x[n] \oplus r; \\
  m &:= r; \\
  \text{if } n > 0 \text{ then} \\
  \quad r &\in \{0, 1\}; \\
  \quad \text{reveal } m \oplus x[0, n) \oplus r; \\
  \quad m &:= r; \\
  \quad n &:= 0 \\
  \text{fi}
\end{align*}
\]
Apply the fixed-point functional

\[
\begin{align*}
n &:= n - 1; \\
r &\in \{0, 1\}; \\
\text{reveal } m \oplus x[n] \oplus r; \\
m &:= r;
\end{align*}
\]

\text{loop body}

\text{if } n > 0 \text{ then}

\[
\begin{align*}
r &\in \{0, 1\}; \\
\text{reveal } m \oplus x[0, n) \oplus r; \\
m &:= r; \\
m &:= r; \\
n &:= 0 \\
\end{align*}
\]

\text{loop specification}

\text{fi}
Apply the fixed-point functional

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]

\textbf{loop body}

\begin{align*}
\text{if } n > 0 & \text{ then}
\end{align*}

\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]

\textbf{loop conditional}

\textbf{loop specification}
It must be a fixed-point

\[
\begin{align*}
n &:= n - 1; & \text{loop body} \\
r &\in \{0, 1\}; \\
\text{reveal } m \oplus x[n] \oplus r;
\end{align*}
\]

\begin{align*}
\text{if } n > 0 \text{ then} \\
r &\in \{0, 1\}; \\
\text{reveal } m \oplus x[0, n) \oplus r; \\
m &:= r; \\
n &:= 0 & \text{loop specification}
\end{align*}

\textbf{fi}
It must be a fixed-point

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]
\[ m := r; \]
\[ \text{if } n > 0 \text{ then} \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
\[ \text{fi} \]
We calculate as follows...
Expand the if

\begin{align*}
n &:= n - 1; \\
r &\in \{0, 1\}; \\
\text{reveal} &\quad m \oplus x[n] \oplus r; \\
m &:= r; \\
\text{if } n>0 \text{ then} \\
r &\in \{0, 1\}; \\
\text{reveal} &\quad m \oplus x[0, n) \oplus r; \\
m &:= r; \\
n &:= 0 \\
\text{fi}
\end{align*}
We calculate as follows...
Expand the \textbf{if}

\begin{verbatim}
if \ n > 0 \ then
\ n := n - 1; \n\ r := \{0, 1\}; \n\ reveal \ m \oplus x[n] \oplus r; \n\ m := r; \n\ r := \{0, 1\}; \n\ reveal \ m \oplus x[0, n) \oplus r; \n\ m := r; \n\ n := 0
fi
\end{verbatim}

We calculate as follows...
Expand the if

\[ n := n - 1; \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r; \]
\[ m := r; \]
\[ \text{if } n > 0 \text{ then} \]
\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
\[ \text{fi} \]
We calculate as follows...
Expand the if

\[
\textbf{if } n > 0 \text{ then }
\]
\[
\begin{align*}
n &:= n - 1; \\
r &\in \{0, 1\}; \\
\text{reveal } m &\oplus x[n] \oplus r; \\
m &:= r; \\
r &\in \{0, 1\}; \\
\text{reveal } m &\oplus x[0, n) \oplus r; \\
m &:= r; \\
n &:= 0
\end{align*}
\]
\textbf{fi}
if $n > 1$ then

$n := n - 1$;

$r \in \{0, 1\}$;

reveal $m \oplus x[n] \oplus r$;

$m := r$;

$r \in \{0, 1\}$;

reveal $m \oplus x[0, n) \oplus r$;

$m := r$;

$n := 0$

else

\[ \ldots \]

Use $n=1$ in the else
Use $n=1$ in the else

if $n>1$ then
  \[ n := n - 1; \]
  \[ r \in \{0, 1\}; \]
  \[ \text{reveal } m \oplus x[n] \oplus r; \]
  \[ m := r; \]
  \[ r \in \{0, 1\}; \]
  \[ \text{reveal } m \oplus x[0, n) \oplus r; \]
  \[ m := r; \]
  \[ n := 0 \]
else . . .
Introduce local hidden $r'$

if $n > 1$ then
    $n := n - 1$;
    $r :\in \{0, 1\}$;
    reveal $m \oplus x[n] \oplus r$;
    $m := r$;
    $r :\in \{0, 1\}$;
    reveal $m \oplus x[0, n) \oplus r$;
    $m := r$;
    $n := 0$
fi

else
    ...

else
    ...

else
    reveal $m \oplus x[0] \oplus r$;
    $m := r$;
    $n := 0$
fi
Introduce local hidden $r'$

\[
\text{if } n > 1 \text{ then} \\
n := n - 1;
\]

\[
\text{else} \\
r := \{0, 1\}; \\
\text{reveal } m \oplus x[0] \oplus r; \\
m := r; \\
n := 0 \\
\text{fi}
\]

\[
r := \{0, 1\}; \\
\text{reveal } m \oplus x[0, n] \oplus r; \\
m := r; \\
n := 0 \\
\text{else . . .}
\]
if $n > 1$ then
  $n := n - 1;$

...else
  $r \in \{0, 1\};$
  \textbf{reveal} $m \oplus x[0] \oplus r;$
  $m := r;$
  $n := 0$
fi

\[\begin{align*}
  | & \textbf{hid } r'. \\
  r' & \in \{0, 1\}; \textbf{reveal} m \oplus x[n] \oplus r'; \\
  r & \in \{0, 1\}; \quad \textbf{reveal} m \oplus x[0, n) \oplus r; \\
  m & := r; \\
  n & := 0
\end{align*}\]
$$n := n - 1;$$

\[ r \in \{0, 1\}; \]

\[
\text{if } n > 1 \text{ then } \\
\quad n := n - 1; \\
\quad \parallel [ \text{hid } r' . \\
\quad \quad r' \in \{0, 1\}; \\
\quad \quad \text{reveal } m \oplus x[n] \oplus r'; \\
\quad \quad \text{reveal } m \oplus x[0, n) \oplus r; \\
\quad ]; \\
\text{else . . . }
\]

\[
\ldots \text{else } \\
\quad \text{reveal } m \oplus x[0] \oplus r; \\
\quad \text{fi} \\
\quad m := r; \\
\quad n := 0
\]
Reveal calculus;
use $n=1$ in else

$r \in \{0, 1\}$;
if $n > 1$ then
    $n := n - 1$;
    $\| \begin{array}{ll}
        \text{hid } r' .
        r' \in \{0, 1\};
        \text{reveal } m \oplus x[n] \oplus r';
        \text{reveal } r' \oplus x[0, n) \oplus r
    \end{array} \|
$ else ...
Reveal calculus;
use $n=1$ in else

$$r \in \{0, 1\}; \quad m := r;$$

if $n > 1$ then
  $$n := n - 1;$$
  $$r' \in \{0, 1\};$$
  $$m \oplus x[n] \oplus r';$$
  $$r' \oplus x[0, n] \oplus r$$
  $$n := 0$$
fi

else . . .
\[ n \equiv n - 1; \]
\[ r \in \{0, 1\}; \] if \( n > 1 \) then
\[ n \equiv n - 1; \]
\[ r' \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n] \oplus r'; \]
\[ \text{reveal } m \oplus x[0,n] \oplus r' \]
\[ r' \in \{0, 1\}; \]
else
\[ m \equiv r; \]
\[ n \equiv 0 \]
\[ \text{reveal } m \oplus x[0,n] \oplus r \]
else ...
Reorder; remove assignment to $n$

$r \in \{0, 1\}$;

if $n > 1$ then

| [ hid $r'$ .
  $r' \in \{0, 1\}$ ;
  reveal $m \oplus x[n] \oplus r'$ ;
  reveal $m \oplus x[0, n] \oplus r$
  ] |

else . . .

\[ \text{\ldots else} \]

reveal $m \oplus x[0, n] \oplus r$

fi

$m := r$;

$n := 0$
Reorder

... else

reveal $m \oplus x[0, n) \oplus r$

fi

$m := r$;

$n := 0$

$r \in \{0, 1\}$;

if $n > 1$ then

$|[

\text{hid} \ r' \cdot \\

r' \in \{0, 1\}; \\

\text{reveal} \ m \oplus x[n - 1] \oplus r'; \\

\text{reveal} \ m \oplus x[0, n) \oplus r$

]|$

else ...


Reorder

\[ r : \in \{0, 1\}; \]
\[ \text{if } n > 1 \text{ then} \]
\[ [\text{hid } r' . \]
\[ r' : \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[n-1] \oplus r'; \]
\[ ]\]
\[ \text{reveal } m \oplus x[0, n) \oplus r \]
\[ m := r; \]
\[ n := 0 \]
Encryption lemma

\( r : \in \{0, 1\}; \)
\( \textbf{if} \ n > 1 \ \textbf{then} \)

\[
\| \text{hid } r' \cdot \\
  r' : \in \{0, 1\}; \\
  \textbf{reveal } m \oplus x[n-1] \oplus r' 
\]

\( \| \)
\( \textbf{fi}; \)
\( \textbf{reveal } m \oplus x[0, n) \oplus r; \)
\( m := r; \)
\( n := 0 \)
Encryption lemma

\[ r \in \{0, 1\}; \]

**reveal** \( m \oplus x[0, n) \oplus r; \)

\( m := r; \)

\( n := 0 \)
Variable $n$ is visible

$r \in \{0, 1\}$;
if $n > 1$ then skip fi;
reveal $m \oplus x[0, n) \oplus r$;
$m := r$;
$n := 0$
Variable $n$ is visible

$r \in \{0, 1\};$
reveal $m \oplus x[0, n) \oplus r;$
$m := r;$
$n := 0$
And we started with...

\[ r \in \{0, 1\}; \]
\[ \text{reveal } m \oplus x[0, n) \oplus r; \]
\[ m := r; \]
\[ n := 0 \]
And we started with...

\[
\begin{align*}
r &: \in \{0, 1\}; \\
\text{reveal} & \quad m \oplus x[0, n) \oplus r; \\
m &: = r; \\
n &: = 0
\end{align*}
\]
The complete program, with the circle closed

\[
\begin{align*}
\begin{array}{l}
\text{hid } l, m, r : \{0, 1\}. \\
l : \in \{0, 1\}; \quad m := l; \\
n := N; \\
\text{repeat} \\
\quad n := n - 1; \\
r : \in \{0, 1\}; \\
\text{reveal } m \oplus x[n] \oplus r; \\
m := r \\
\text{until } n = 0; \\
\text{reveal } r \oplus x[N] \oplus l \\
\end{array}
\end{align*}
\]