

# The thousand-and-one cryptographers

AK McIver<sup>1</sup> and CC Morgan<sup>2</sup>

<sup>1</sup> Dept. of Computing, Macquarie University, NSW 2109 Australia;  
anabel@ics.mq.edu.au

<sup>2</sup> School of Comp. Sci. and Eng., Univ. NSW, NSW 2052 Australia;  
carrollm@cse.unsw.edu.au

**Abstract.** Chaum’s *Dining Cryptographers* protocol crystallises the essentials of security just as other famous diners once captured deadlock and livelock: it is a benchmark for security models and their associated verification methods.

Here we give a correctness proof of the Cryptographers in a new style, one in which stepwise refinement plays a prominent role. Furthermore, our proof applies to arbitrarily many Diners: to our knowledge we are only the second group to have done that.

The proof is based on the *Shadow Security Model* which integrates non-interference and program refinement: with it, we try to make a case that stepwise development of security protocols is not only possible but actually is quite a good idea. It benefits from more than three decades’ of experience of how layers of abstraction can both simplify the design process and make its outcomes more likely to be correct.

**Keywords:** Security, refinement, automated proof, correctness, unbounded dining cryptographers.

## 1 Introduction: refinement of security properties

Program development by stepwise refinement [32] is widely accepted as a good idea in theory, but it is often a late arrival in practice. Indeed, with some notable exceptions [1, 5] most current approaches and tools for correctness concentrate on proving<sup>3</sup> that a single system has certain desirable properties, whereas a refinement-based approach would rather prove that one (real, i.e. implementation) system had all the desirable properties of another (ideal, i.e. specification) system.

As an example, we note the frequent claims that downgrading is a challenging issue in the non-interference model of security [12]. For example, in that model a program is secure if observation of its “low-security” visible outputs does not reveal anything about its “high-security” hidden inputs; thus in the context of visible integer variables  $v$  and hidden  $h$  the program  $v:=0\times h$  is secure but  $v:=1\times h$  is not. As an intermediate option there is the program  $v:=h\div 2$  that

---

<sup>3</sup> We include model checking as a form of proving, at this informal level.

“downgrades” the security, revealing in this case most bits of  $v$  but not all, yet it is not considered to be “intermediately” secure: like  $v:=1\times h$ , it is considered (simply) insecure.

Now in a real system we might find the code

$$v:=0; \text{ while } v+2 \leq h \text{ do } \langle \text{send two bytes} \rangle; v:=v+2 \text{ end } , \quad (1)$$

in which  $v$  counts the number of bytes sent, ensuring no more than  $h$  can be sent overall. If the sent messages are observed and counted, then this program also reveals –to anyone aware of the source code– all but the low-order bit of  $h$ . Like  $v:=h \div 2$  above, it is considered insecure in the non-interference model.

Downgrading is inescapable in practice, and it is to reason about it effectively –in spite of the black-or-white judgement of the original non-interference model– that downgrading extensions to that model are introduced [7, 19] in which one can express, by annotations of the code, the information leaks that are to be considered acceptable. The proof of correctness is then relative to those annotations.

But there is an alternative to concentrating on downgrading exclusively: instead we concentrate on refinement, with downgrading then a special case. With this approach we would describe a downgrading policy for the loop of (1) as a refinement, saying that

$$v:=2(h\div 2) \quad \sqsubseteq \quad v:=0; \text{ while } v+2 \leq h \text{ do } \langle \text{send} \rangle; v:=v+2 \text{ end } , \quad (2)$$

i.e. that the *lhs* “is refined by” the *rhs* and meaning that the desirable properties of the specification on the left –including its not revealing  $h$ ’s low-order bit– are shared by its implementation on the right. The downgrading aspect is that the *rhs* can indeed reveal the higher-order bits of  $h$  — because the *lhs* does just that. We feel that a refinement-based approach has many advantages, demonstrated over the years by its success generally (where it has after all been adopted). In this particular case, for example, using it would mean that we require neither annotations nor an explicit notion of downgrading.

Integrating refinement and security is exactly what we do here: we make (2) precise by giving an appropriately extended definition of refinement, one which is “security aware.” It is explained below (and earlier [23, 25]). In doing so we join a small number of other researchers –from the large security community– who have similar aims [20, 2, 9]; we compare our work with theirs in Sec. 9. Some conspicuous aspects of our approach are as follows.

Refinement is complementary to abstraction, and abstraction can be viewed in turn as demonic nondeterministic choice resolved at design-time; but it is well known that there are conceptual benefits to conflating abstraction with run-time demonic choice [8, 4, 16, 21] and so it is natural to include demonic choice in our security model in both the design- and run-time senses. Where this has been done by others, in some cases the non-interference model has been extended so that the “full range of nondeterminism” of the hidden variables must not be

dependent on visible variables’ observed values, but the nondeterminism cannot subsequently be reduced as refinement would ordinarily allow [18, 30]; and in other cases the requirement has been imposed that –while nondeterminism is allowed in the model– in the final implementation program there must be none of it remaining [29]. *One aspect* of our work is that, in contrast, we include both features: nondeterminism can be reduced; and some of it can remain in the implementation. This requires careful treatment of hidden- vs. visible nondeterminism, and is how we solved the Refinement Paradox [22, 17].

*A second aspect* is that our notion of adversary is quite strong: we allow perfect recall [11], that intermediate values of visible variables can be observed even if they subsequently are overwritten; and we allow an attacker’s observation of control flow, that conditionals’ Booleans expressions are (implicitly) leaked. We assume also that the program code is known. Doing all this effectively, while avoiding an infinite regress to attacks at the level of quarks, requires explicit treatment of atomicity at some point. We define that.

The reason for the strong adversary is that refinements must be effective locally: refinement of a small fragment in a large program must refine the whole program even if the refinement was proved only for the fragment. This is monotonicity — and without it no scaled-up development is possible. Since for some fresh local visible variable  $v'$  it is a refinement to insert assignment  $v':=v$  willy-nilly at almost any place in a program, we must live with the fact that  $v$ ’s value at that point will possibly be preserved (in some local  $v'$ ) in spite of  $v$ ’s subsequent overwriting: that is, although the unfortunate  $v':=v$  might not be there “now,” one developer must accept that a second developer in some other building might put it there “later” without asking. After all, if it’s a refinement (and it is) then he doesn’t *need* to ask.

Rather than make refinements unworkable, rather the strong-adversary assumptions make them more applicable. Distributed protocols (such as the Dining Cryptographers) can be treated as single “sequential” programs because the information hiding normally implied by non-interference’s end-to-end analysis does not apply. Indeed, if it did, the sequential formulation would seem to be hiding the transfer of messages and the interleaving of concurrent threads, and that would make it unsound. For example, if Agent  $A$  executes  $v:=h$  and Agent  $B$  then executes  $v:=0$  we can analyse this as the single sequential program  $v:=h;v:=0$  without having accidentally (and incorrectly) ignored the fact that  $A$  can observe  $v$  (and hence learn  $h$ ) before  $B$ ’s thread has begun the execution that would overwrite it.

*A third aspect* of our approach is that we concentrate on algebraic reasoning for proving refinement: although we do have both an operational model (Sec. 2) and a language of logical assertions (Sec. 3) for refinement-based security, we use those mainly for proving schematic program-fragment -equalities and refinements (Sec. 4). Those schemes, rather than the logic or the model, are then what is used in the derivation of specific programs. For this we need a program-level indication of information escape, analogous to the way in which the *assert* statement can embed Hoare-Logic pre- and post-conditions within program code [15, 21, 33].

This is our **reveal**  $E$  statement that publishes  $E$ 's value for all to read, but does not change any program variable: its purpose is to bring an extra expressivity that helps formulate general algebraic (in)equalities. Since functionally it acts as **skip** on program variables (having no effect at all), but *wrt* secrecy it does not act as **skip** (it releases information but **skip** does not), its behaviour considered alone will capture much of the flavour of what we intend to do.

Section 2 gives our relational-style operational model; Section 3 describes a corresponding modal logic based on the logic of knowledge; and Section 4 introduces our program algebra. Sections 5–8 demonstrate the approach on examples of increasing complexity.

Throughout we use left-associating dot for function application, so that  $f.x.y$  means  $(f(x))(y)$  or  $f(x, y)$ , and comprehensions/quantifications are written uniformly, as  $(\mathbf{Q}x:T \mid R \cdot E)$  for quantifier  $\mathbf{Q}$ , bound variable(s)  $x$  of type(s)  $T$ , range predicate  $R$  (probably) constraining  $x$  and element-constructor  $E$  in which  $x$  (probably) appears free. For sets the opening “ $\mathbf{Q}$ ” is “ $\{$ ” and the closing “ $\}$ ” is “ $\}$ ” so that e.g. the comprehension  $\{x, y: \mathbb{N} \mid y = x^2 \cdot z + y\}$  is the set of all natural numbers that exceed  $z$  by a perfect square exactly, that is  $\{z, z+1, z+4, \dots\}$ .

## 2 The Shadow model of security

### 2.1 Introduction; non-interference; logic of knowledge

Our operational model is loosely based on non-interference [12] and on the Kripke structures associated with the (modal) Logic of Knowledge [11]: it extends the former with concepts of the latter, and is targetted specifically at development of secure programs (in its terms) via a process of stepwise refinement. The “shadow” of the title refers to an extra semantic component that tracks a postulated attacker’s inferred knowledge, or ignorance, of hidden (high-security) variables.

The non-interference approach (in its simplest form) partitions variables into high-security- and low-security classes: we call them *hidden* and *visible* respectively. A “non-interference -secure” program then prevents an attacker’s inferring hidden variables’ initial values from initial and/or final visible variables’ values. Assuming for simplicity just two variables  $v, h$  of class visible, hidden resp. we consider in this simple approach a possibly nondeterministic program  $r$  that takes initial states  $(v, h)$  to sets of final visible states  $v'$  and is thus of type  $\mathcal{V} \rightarrow \mathcal{H} \rightarrow \mathbb{P}\mathcal{V}$ , where  $\mathcal{V}, \mathcal{H}$  are the value sets corresponding to the types of  $v, h$  respectively; note that we are ignoring final hidden values at this point. Such a program  $r$  is *non-interference -secure* just when for any initial visible value the set of possible final visible values is independent of the initial hidden value [18, 30, 26]:

$$(\forall v: \mathcal{V}; h_0, h_1: \mathcal{H} \cdot r.v.h_0 = r.v.h_1) .$$

Our first extension of the simple approach is to concentrate on final- (rather than initial) hidden values and therefore model programs by the slightly more elaborate type  $\mathcal{V} \rightarrow \mathcal{H} \rightarrow \mathbb{P}(\mathcal{V} \times \mathcal{H})$ . For two such programs  $r_{\{1,2\}}$  we say that  $r_1 \sqsubseteq r_2$ , that  $r_1$  “is (securely) refined by”  $r_2$ , just when the following both hold:

- (i) For any initial state  $v, h$  each possible  $r_2$  outcome is also a possible  $r_1$  outcome, that is

$$(\forall v: \mathcal{V}; h: \mathcal{H} \cdot r_1.v.h \supseteq r_2.v.h) \text{ .}^4$$

This is the normal “can reduce nondeterminism” form of refinement, i.e. it is the classical form.

- (ii) For any initial state  $v, h$  and final state  $v'$  possible for  $r_2$  (i.e. for which there exists a compatible  $h'_2$ ), each  $h'_1$  that  $r_1$  can produce with that  $v, v'$  can also be produced by  $r_2$  with that same  $v, v'$ , that is

$$\left( \begin{array}{l} (\forall v, v': \mathcal{V}; h, h'_1, h'_2: \mathcal{H} \cdot (v', h'_2) \in r_2.v.h \wedge (v', h'_1) \in r_1.v.h) \\ \Rightarrow (v', h'_1) \in r_2.v.h \end{array} \right) \text{ .}$$

This second condition says that for any particular visible final  $v'$  the attacker’s “ignorance” of  $h'$ ’s compatible with that  $v'$  cannot be decreased by the refinement from  $r_1$  to  $r_2$ : whenever we must consider out of ignorance that some  $h'_1$  is possible for  $r_1$ , we must be forced to consider that same  $h'_1$  to be possible for  $r_2$  as well. This extended “secure” refinement is thus more restrictive than the classical.

In fact, in this moderately extended approach, the two conditions (i), (ii) together do not allow ignorance strictly to increase: refinement then boils down to a simple policy of allowing decrease of nondeterminism in  $v$  but not in  $h$ . But strict increase of hidden nondeterminism will be possible (3) in the more ambitious approach we introduce below.

As an example of the above we restrict all our variables’ types so that  $\mathcal{V} = \mathcal{H} = \{0, 1\}$ , and we let  $r_1$  be the maximally nondeterministic program that can produce from any initial values  $(v, h)$  any one of the four possible  $(v', h')$  final values in  $\mathcal{V} \times \mathcal{H}$ . Then the program  $r_2$  that produces only the two final values in  $\{(0, 0), (0, 1)\}$  is a refinement of  $r_1$  that strictly reduces nondeterminism in  $v$  but not in  $h$ , and is (therefore) still secure. But the program  $r'_2$  that produces only the two final values in  $\{(0, 0), (1, 1)\}$  is not a secure refinement, because it reduces nondeterminism in  $h$  (as well).

Thus  $r_1$  allows any behaviour, and  $r_2$  simply reduces the nondeterminism by limiting its outputs to  $v'=0$  only; but, even with the limited outputs, an attacker of  $r_2$  can gain no more knowledge of  $h'$ ’s value than it would have had from attacking  $r_1$  instead. So  $r_1 \sqsubseteq r_2$ . An attacker of  $r'_2$  however can deduce  $h'$ ’s value from having seen  $v'$ ’s, since the program guarantees they will be equal. Since that attack is not possible on  $r_1$ , we have  $r_1 \not\sqsubseteq r'_2$ .

---

<sup>4</sup> Some researchers [2] do not consider the final  $h$  here: for our purposes that would make our program operators non-monotonic for refinement (thus a failure of compositionality). That is, if for hidden  $h$  the assignments  $h:=0$  and  $h:=1$  are the same, then for visible  $v$  the compositions  $h:=0; v:=h$  and  $h:=1; v:=h$  should not be different.

## 2.2 The Shadow of $h$

In  $r_1$  above, when the final value  $v'$  was 0 the corresponding set of associated possible values of  $h'$  was  $\{0, 1\}$ . This set  $\{0, 1\}$  is called “The Shadow,” and represents explicitly an attacker’s ignorance of the hidden variables’ values. In  $r_2$  that shadow was the same (for  $v' = 0$ ); but in  $r'_2$  the shadow was smaller, and that is why we don’t consider  $r'_2$  to be a refinement of  $r_1$  as far as security is concerned.

In the shadow semantics we model this explicit ignorance-set, so that our final program state is extended to a triple  $(v', h', H')$  with  $H'$  a subset of  $\mathcal{H}$  — in each triple the  $H'$  contains exactly those (other) values that  $h'$  might have had. The (extended) output-triples of the three example programs are then respectively

$$\begin{aligned} r_1 &— \{(0, 0, \{0, 1\}), (0, 1, \{0, 1\}), (1, 0, \{0, 1\}), (1, 1, \{0, 1\})\} \\ r_2 &— \{(0, 0, \{0, 1\}), (0, 1, \{0, 1\})\} \\ r'_2 &— \{(0, 0, \{0\}), (1, 1, \{1\})\} , \end{aligned}$$

and we can see  $r_1 \sqsubseteq r_2$  because  $r_1$ ’s set of outcomes includes all of  $r_2$ ’s. But e.g. the outcome  $(0, 0, \{0\})$  of  $r'_2$  does not occur among  $r_1$ ’s outcomes, nor is there even an  $r_1$ -outcome  $(0, 0, H')$  with  $H' \subseteq \{0\}$  that would satisfy (ii). That is why we say that  $r_1 \not\sqsubseteq r'_2$  for security.

Now –the final enhancement– for sequential composition of shadow-enhanced programs also initial triples  $(v, h, H)$  must be dealt with, since the final triples of a first component become initial triples for the second. We therefore define the full shadow semantics, in the next subsection, by showing how those triples are related by program execution.

## 2.3 The Shadow semantics of atomic programs

A “non-shadow,” call it *classical* program  $r$  is effectively an input-output relation between  $\mathcal{V} \times \mathcal{H}$  -pairs. Its shadow version  $\langle r \rangle$  is a relation between  $\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H}$  -triples and is defined as follows:

**Definition 1. Atomicity** Given a standard program  $r: \mathcal{V} \rightarrow \mathcal{H} \rightarrow \mathbb{P}(\mathcal{V} \times \mathcal{H})$  we define its *atomic shadow version*  $\langle r \rangle: \mathcal{V} \rightarrow \mathcal{H} \rightarrow \mathbb{P}\mathcal{H} \rightarrow \mathbb{P}(\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H})$  so that  $\langle r \rangle.v.h.H \ni (v', h', H')$  just when

- (i) we have both  $r.v.h \ni (v', h')$
- (ii) and  $H' = \{h': \mathcal{H} \mid (\exists h: H \cdot r.v.h \ni (v', h'))\}$  .

The final shadow component is generated from the initial shadow component and any nondeterminism present in the program.  $\square$

As a first example, let the syntax  $x: \in S$  denote the standard program that chooses variable  $x$ ’s value from a set  $S$ , which we assume to be non-empty. From Def. 1 we have that

- (i) A choice of visible  $v$  has no effect on  $h, H$   
because  $\langle v: \in S \rangle.v.h.H = \{v': S \cdot (v', h, H)\}$  , but

- (ii) A choice of hidden  $h$  introduces ignorance  
because  $\langle h:\in S \rangle.v.h.H = \{h':S \cdot (v, h', S)\}$  and finally
- (iii) An assignment of hidden to visible “collapses” any ignorance that might  
be there because  $\langle v:=h \rangle.v.h.H = \{(h, h, \{h\})\}$  .

From (ii) and (iii) above we can therefore see that in the sequential composition  $\langle h:\in S \rangle; \langle v:=h \rangle$  the first statement introduces ignorance –we do not know  $h$ ’s exact value “at the semicolon”– but the second statement then removes it because we can deduce  $h$ ’s value, at the end, by observing  $v$ . The composition as a whole is nondeterministic, because  $v, h$ ’s common final value is drawn arbitrarily from  $S$ ; but the nondeterminism is observable.

In general atomicity is not preserved by composition (indeed one expects it not to be); but in many simple cases it is preserved.

**Lemma 1.** *atomicity and composition* Given two programs  $r_{\{1,2\}}$  over  $v, h$  we have  $\langle r_1; r_2 \rangle = \langle r_1 \rangle; \langle r_2 \rangle$  just when  $v$ ’s *intermediate* value, i.e. “at the semicolon,” can be deduced from its *endpoint* values, i.e. initial and final, possibly in combination. The semicolon denotes relational composition in both cases, of pairs on the left and of triples on the right.

*Proof:* Given in App. A. □

In fact this lemma is more significant when its conditions are *not* met than when they are. It means for example that we cannot conclude from Lem. 1 that  $\langle v:=h; v:=0 \rangle = \langle v:=h \rangle; \langle v:=0 \rangle$ , since on the left the intermediate value of  $v$  cannot be deduced from its endpoint values: for  $h$  is not visible at the beginning and  $v$  itself has been “erased” at the end. And indeed from Def. 1

- (i) On the left we have  $\langle v:=h; v:=0 \rangle.v.h.H = \{(0, h, H)\}$ , whereas
- (ii) On the right we have  $(\langle v:=h \rangle; \langle v:=0 \rangle).v.h.H = \{(0, h, \{h\})\}$  .

This phenomenon is called *perfect recall* [11] –that  $v$ ’s temporary receipt of  $h$  is seen by an attacker even though it is subsequently overwritten– and it is a feature (not a bug). It is due to our refinement-oriented point of view, as we now explain.

## 2.4 Refinement vs. atomicity: *Gedanken* experiments

Perfect recall is necessary because refinement must be *monotonic*, i.e. (A) that refinement of a program portion must refine the whole program; and we insist additionally (B) that classical refinements involving  $v$  only must remain valid. Both principles (A,B) are required in order to be able to develop large programs via local reasoning over small portions.

For example, without perfect recall the overwriting of  $v$  would prevent program  $v:=h; v:\in\{0,1\}$  from revealing  $h$ . Yet from (B) we have  $v:\in\{0,1\} \sqsubseteq v:=v$ ; and then from (A) we have  $(v:=h; v:\in\{0,1\}) \sqsubseteq (v:=h; v:=v)$  — and it is a contradiction of secure refinement that the *lhs* does not reveal  $h$  but the *rhs* does. Thus the premise –that recall is not perfect– is false.

There is a similar experiment for conditionals: because (A,B) imply the refinement

$$\sqsubseteq \text{ if } h=0 \text{ then } v:\in\{0,1\} \text{ else } v:\in\{0,1\} \text{ fi} \\ \sqsubseteq \text{ if } h=0 \text{ then } v:=0 \quad \text{else } v:=1 \quad \text{fi} ,$$

we must accept that the **if**-test reveals its outcome, in this case whether  $h=0$  holds. And nondeterministic choice  $P_1 \sqcap P_2$  is visible to the attacker because each of the two branches  $P_{\{1,2\}}$  can be refined separately in a similar way.

## 2.5 Declared atomicity

If there is a code fragment  $P$  that we know will be executed atomically at runtime, we can write it  $\langle P \rangle$  using the notation of Def. 1. This will however have two consequences:

- (i) At runtime the atomicity must be guaranteed for  $P$ 's execution, and
- (ii) At design-time only equality reasoning can be used within  $P$ .

With respect to (ii) we mean that  $P \sqsubseteq P'$  does not allow us to conclude the refinement  $\langle P \rangle \sqsubseteq \langle P' \rangle$ . We can however conclude the equality  $\langle P \rangle = \langle P' \rangle$  from  $P = P'$ .

## 2.6 Summary of semantics

The Shadow Semantics of a small imperative language is given in Fig. 1 for non-looping constructs. The only non-traditional command is **reveal** that gives the value of some expression to the attacker directly, but changes no program variables; note it does change the shadow.

Refinement between programs is defined as follows:

**Definition 2.** *Refinement* For programs  $P_{\{1,2\}}$  we say that  $P_1$  is refined by  $P_2$  and write  $P_1 \sqsubseteq P_2$  just when for all  $v, h, H$  we have

$$(\forall (v', h', H'_2): \llbracket P_2 \rrbracket.v.h.H \cdot \\ (\exists H'_1: \mathbb{P}\mathcal{H} \mid H'_1 \subseteq H'_2 \cdot (v', h', H'_1) \in \llbracket P_1 \rrbracket.v.h.H) ) .$$

This means that for each initial triple  $(v, h, H)$  every final triple  $(v', h', H'_2)$  produced by  $P_2$  must be “justified” by a triple  $(v', h', H'_1)$ , with equal or smaller ignorance, produced by  $P_1$ .<sup>5</sup>  $\square$

For example, from Fig. 1 we have that  $\llbracket h:=0 \sqcap h:=1 \rrbracket.v.h.H$  produces the outcome  $\{(v, 0, \{0\}), (v, 1, \{1\})\}$  whereas  $\llbracket h:\in\{0,1\} \rrbracket.v.h.H$  produces the outcome  $\{(v, 0, \{0,1\}), (v, 1, \{0,1\})\}$ . Thus

$$h:=0 \sqcap h:=1 \quad \sqsubseteq \quad h:\in\{0,1\} \tag{3}$$

<sup>5</sup> This is the Smyth Order [31] on sets of outcomes that is induced by the order “ $(v, h, H_1) \sqsubseteq (v, h, H_2)$  iff  $H_1 \subseteq H_2$ ” on individual outcomes.

	<u>Program <math>P</math></u>	<u>Semantics <math>\llbracket P \rrbracket.v.h.H</math></u>	
Publish a value	<b>reveal</b> $E.v.h$	$\{ (v, h, \{h': H \mid E.v.h' = E.v.h\}) \}$	
Assign to visible	$v := E.v.h$	$\{ (E.v.h, h, \{h': H \mid E.v.h' = E.v.h\}) \}$	★
Assign to hidden	$h := E.v.h$	$\{ (v, E.v.h, \{h': H \cdot E.v.h'\}) \}$	★
Choose visible	$v \in S.v.h$	$\{ v' : S.v.h \cdot (v', h, \{h': H \mid v' \in S.v.h'\}) \}$	★
Choose hidden	$h \in S.v.h$	$\{ h' : S.v.h \cdot (v, h', \{h': H; h'' : S.v.h' \cdot h''\}) \}$	★
Sequential composition	$P_1; P_2$	$(\llbracket P_1 \rrbracket; \llbracket P_2 \rrbracket).v.h.H$	
Demonic choice	$P_1 \sqcap P_2$	$\llbracket P_1 \rrbracket.v.h.H \cup \llbracket P_2 \rrbracket.v.h.H$	
Conditional	<b>if</b> $E.v.h$ <b>then</b> $P_t$ <b>else</b> $P_f$ <b>fi</b>	$\begin{aligned} & \llbracket P_t \rrbracket.v.h.\{h': H \mid E.v.h' = \mathbf{true}\} \\ & \quad \triangleleft E.v.h \triangleright \\ & \llbracket P_f \rrbracket.v.h.\{h': H \mid E.v.h' = \mathbf{false}\} \end{aligned}$	

The commands  $P$  marked ★ satisfy  $\llbracket P \rrbracket = \langle \text{“classical semantics of } P \text{”} \rangle$ , and we call them the *atomic* commands, meaning semantically so. Note that **reveal** is therefore not security-atomic, even though it is a syntactic atom.

**Fig. 1.** Semantics of non-looping commands

is an example of a strict refinement where the two commands differ only by a strict increase of ignorance: they have equal nondeterminism functionally, but in one case ( $\sqcap$ ) it can be observed by the attacker and in the other case ( $\in$ ) it cannot. For example the “more ignorant” triple  $(v, 0, \{0, 1\})$  is strictly justified by the “less ignorant” triple  $(v, 0, \{0\})$ , where we say “strictly” because  $\{0\} \subset \{0, 1\}$ .

### 3 The Logic of Ignorance

With the  $(v, h, H)$ -triple semantics of Sec. 2.6 comes an assertion logic over the triples; it is based on the Logic of Knowledge and its interpretation over Kripke structures [11]. We call it *The Logic of Ignorance*.

As in Hoare Logic for sequential programs [15] we interpret first-order predicate formulae over program states by making the program variables’ values act as constants in the logic. To that we add a *possibility modality* so that  $P\phi$  means (roughly) that  $\phi$  holds for some “possible” value  $h \in H$  rather than necessarily for the actual current value of  $h$ , where  $\phi$  is a classical formula. In fact we have  $\phi \Rightarrow P\phi$  because a property of our semantics is that  $h \in H$  for all triples  $(v, h, H)$  we consider: what is true must also be possible. In general however we do not have  $P\phi \Rightarrow \phi$ , since what is possible is not necessarily true.

#### 3.1 Interpretation of modal formulae

The assertion language contains function- (including constant-) and relation symbols as needed, among which we distinguish the (program-variable) constant symbols *visibles* in some set  $V$  and *hiddens* in  $H$ ; as well there are the

usual (logical) variable symbols in  $L$  over which we allow  $\forall, \exists$  quantification. The visibles, hidden and variables are collectively the *scalars*  $X := V \cup H \cup L$  with  $V, H, L$  assumed disjoint.

A *structure* comprises a non-empty domain  $\mathcal{D}$  of values, together with functions and relations over it that interpret the function- and relation symbols mentioned above; within the structure we name the partial functions  $v, h$  that interpret visibles and hidden respectively; we write their types  $V \mapsto \mathcal{D}$  and  $H \mapsto \mathcal{D}$  respectively (where the “crossbar” indicates the potential partiality of the function). We don’t bother naming the interpretations of function- and relation symbols, as they do not vary from one program state to another.

A *valuation* is a partial function from scalars to  $\mathcal{D}$ , thus typed  $X \mapsto \mathcal{D}$ ; one valuation  $w'$  can override another  $w$  so that for scalar  $x$  we have that  $(w \triangleleft w').x$  is  $w'.x$  if  $w'$  is defined at  $x$  and is  $w.x$  otherwise. The valuation  $x \mapsto d$  is defined only at variable  $x$ , where it takes value  $d$ .

A *state*  $(v, h, H)$  comprises a visible-  $v$ , hidden-  $h$  and *shadow*- part  $H$ ; the last, in  $\mathbb{P}(H \mapsto \mathcal{D})$ , is a *set* of valuations over hidden only. All the states that we consider satisfy  $h \in H$ .

We define truth of  $\Phi$  at  $(v, h, H)$  under valuation  $w$  by induction in the usual style, writing  $(v, h, H), w \models \Phi$ . For term  $t$  let  $\llbracket t \rrbracket.v, h, w$  be its value interpretation determined inductively from the valuation  $v \triangleleft h \triangleleft w$  and the (implicit) interpretation of function symbols. Then our formula interpretation is as defined as in Fig. 2 [11, pp. 79,81].

### 3.2 Shadow-sensitive Hoare-triples; revelations

As is normal, we say that  $\{\Phi\} prog \{\Psi\}$  just when any initial state  $(v, h, H) \models \Phi$  can lead via  $\llbracket prog \rrbracket$  only to final states  $(v', h', H') \models \Psi$ ; typically  $\Phi$  is called the *precondition* and  $\Psi$  is called the *postcondition*.

We illustrate Shadow-sensitive Hoare-triples with the **reveal**  $E$  command: we have for any classical  $\phi$  that<sup>6</sup>

$$\{P(E=E_0 \wedge \phi)\} \mathbf{reveal} E \{P\phi\}, \quad \text{where } E_0 \text{ is } E \text{ with all its hidden variables 0-subscripted.} \quad (4)$$

It is verified as follows:

$$\begin{aligned} & (v, h, H) \models P(E=E_0 \wedge \phi) \\ \text{and } & \llbracket \mathbf{reveal} E \rrbracket.v, h, H \ni (v', h', H') \end{aligned}$$

iff “Fig. 2, for some  $\hat{h} \in H$  and  $h_0 = (h_0 \mapsto h.h)$ ”

$$\begin{aligned} & (v, \hat{h}, H), (w \triangleleft h_0) \models E=E_0 \wedge \phi \\ \text{and } & \llbracket \mathbf{reveal} E \rrbracket.v, h, H \ni (v', h', H') \end{aligned}$$

iff “Fig. 2”

$$\begin{aligned} & (v, \hat{h}, H), (w \triangleleft h_0) \models E=E_0 \\ \text{and } & (v, \hat{h}, H), (w \triangleleft h_0) \models \phi \\ \text{and } & \llbracket \mathbf{reveal} E \rrbracket.v, h, H \ni (v', h', H') \end{aligned}$$

<sup>6</sup> We use upper case for modal formulae, and lower case for classical.

- $(v, h, H), w \models R.t_1 \dots t_K$  for relation symbol  $R$  and terms  $t_{\{1 \dots K\}}$  iff the tuple  $(\llbracket t_1 \rrbracket.v.h.w, \dots, \llbracket t_K \rrbracket.v.h.w)$  is an element of the interpretation of  $R$  in  $\mathcal{D}^K$ .
- $(v, h, H), w \models t_1 = t_2$  iff  $\llbracket t_1 \rrbracket.v.h.w = \llbracket t_2 \rrbracket.v.h.w$ .
- $(v, h, H), w \models \neg\Phi$  iff  $(v, h, H), w \not\models \Phi$ .
- $(v, h, H), w \models \Phi_1 \wedge \Phi_2$  iff  $(v, h, H), w \models \Phi_1$  and  $(v, h, H), w \models \Phi_2$ .
- $(v, h, H), w \models (\forall x \cdot \Phi)$  iff  $(v, h, H), w \triangleleft (x \mapsto d) \models \Phi$  for all  $d$  in  $\mathcal{D}$ .
- $(v, h, H), w \models P\Phi$  iff  $(v, \hat{h}, H), w \models \Phi$  for some  $\hat{h}$  in  $H$ .

We write just  $(v, h, H) \models \Phi$  when  $w$  is empty, and  $\models \Phi$  when  $(v, h, H) \models \Phi$  for all  $(v, h, H)$  with  $h \in H$ , and we take advantage of the usual “syntactic sugar” for other operators. Thus for example we can show  $\models \Phi \Rightarrow P\Phi$  for all  $\Phi$ , a fact which we mentioned earlier. Similarly we can assume *wlog* that modalities are not nested, since we can remove nestings via the validity  $\models P\Phi \equiv (\exists c \cdot \Phi_{h \leftarrow c} \wedge P(h=c))$ .

As a convenience we allow 0-subscripted hidden variables (e.g.  $h_0$ ) within the modality to refer to the actual rather than potential hidden value; for that we extend the last clause above to read

- $(v, h, H), w \models P\Phi$  iff  $(v, \hat{h}, H), w \triangleleft (h_0 \mapsto h.h) \models \Phi$  for some  $\hat{h}$  in  $H$ .

Thus for example  $P(h = \neg h_0)$  means that whatever value Boolean  $h$  might have, we must consider also its negation to be possible: we do not (cannot, if that formula holds) know it exactly.

**Fig. 2.** Interpretation of Logic of Ignorance

---


$$\begin{array}{ll}
\text{iff} & E.v.\hat{h} = E.v.h \qquad \text{“Fig. 2; Fig. 1”} \\
& \text{and } (v, \hat{h}, H), (w \triangleleft h_0) \models \phi \\
& \text{and } v' = v \wedge h' = h \wedge H' = \{h' : H \mid E.v.h' = E.v.h\} \\
\\
\text{iff} & \hat{h} \in H' \qquad \text{“third line simplifies first; equalities”} \\
& \text{and } (v', \hat{h}, H), (w \triangleleft h'_0) \models \phi \\
& \text{and } v' = v \wedge h' = h \wedge H' = \{h' : H \mid E.v.h' = E.v.h\} \\
\\
\text{iff} & \hat{h} \in H' \qquad \text{“classical } \phi \text{ has shadow-independent interpretation:} \\
& \text{and } (v', \hat{h}, H'), (w \triangleleft h'_0) \models \phi \qquad \text{thus can replace } H \text{ by } H' \text{”} \\
& \text{and } v' = v \wedge h' = h \wedge H' = \{h' : H \mid E.v.h' = E.v.h\} \\
\\
\text{implies} & (v', \hat{h}, H'), (w \triangleleft h'_0) \models \phi \qquad \text{“for some } \hat{h} \in H' \text{”} \\
\\
\text{iff} & (v', h', H'), w \models P\phi \qquad \text{“Fig. 2”}
\end{array}$$

That was not a pretty calculation but, having done it once, we can use (4) forever.

In fact the precondition in (4) is the *weakest* such with respect to postcondition  $P\phi$ , and we thoroughly explore ignorance-based weakest preconditions

elsewhere [8, 23, 25]. Using that, we can give some examples of assertion-based reasoning about revelations.

In the items below, let  $\Psi$  be the assertion  $P((h \bmod 2 = h_0 \bmod 2) \wedge h=3)$ , generated by (4) applied to **reveal**  $(h \bmod 2) \{P(h=3)\}$ , which we can simplify as follows:

$$\begin{aligned} & P((h \bmod 2 = h_0 \bmod 2) \wedge h=3) \\ = & P((3 \bmod 2 = h_0 \bmod 2) \wedge h=3) \\ = & P((1 = h_0 \bmod 2) \wedge h=3) \\ = & (h \bmod 2)=1 \wedge P(h=3) , \end{aligned}$$

that is **odd**  $h \wedge P(h=3)$ . With that we consider the following examples:

- |       |                                                                                                                                                                                                                                                                                                                                                                                                  |       |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| (i)   | Does $h:\in\{1,3\}$ ; <b>reveal</b> $(h \bmod 2)$ establish $P(h=3)$ ?<br>Command $h:\in\{1,3\}$ establishes both conjuncts of $\Psi$ .                                                                                                                                                                                                                                                          | Yes ✓ |
| (ii)  | Does $h:\in\{1,5\}$ ; <b>reveal</b> $(h \bmod 2)$ establish $P(h=3)$ ?<br>Command $h:\in\{1,5\}$ does not establish $P(h=3)$ , which is the second conjunct of $\Psi$ . Given the source code, it is obvious that $h$ cannot be 3 finally.                                                                                                                                                       | No ×  |
| (iii) | Does $h:\in\{2,3\}$ ; <b>reveal</b> $(h \bmod 2)$ establish $P(h=3)$ ?<br>Command $h:\in\{1,5\}$ does not establish <b>odd</b> $h$ , which is the first conjunct of $\Psi$ . One possible outcome is that 0 is revealed, which precludes $h$ 's being finally 3.                                                                                                                                 | No ×  |
| (iv)  | Does $(h:=1 \sqcap h:=3)$ ; <b>reveal</b> $(h \bmod 2)$ establish $P(h=3)$ ?<br>The left-hand command $h:=1$ –a demonic possibility which we must take into account– establishes the first conjunct of $\Psi$ but not the second. Because the nondeterminism is visible (unlike Case (i)), if the left branch is taken –and it might be– then from the source code we know that $h$ cannot be 3. | No ×  |

Note especially the difference between (i) and (iv). In the former, the non-determinism occurs within an atomic command, and is therefore hidden; but, in the latter, it occurs between atomic commands, and is therefore observable.

## 4 The Algebra of Ignorance

### 4.1 Assertions: the historical motivation

The Hoare-logic method of program correctness involves “hybrid” formulae (the triples) that are built from two formal languages: the programming language, and the assertion language. Thus  $\{\phi\}prog\{\psi\}$  in its partial correctness interpretation holds just when every terminating execution of *prog* from an initial state satisfying  $\phi$  is guaranteed to deliver a final state satisfying  $\psi$ .

The command **assert**  $\phi$  is typically defined to act as **skip** in states satisfying  $\phi$  and to “abort” (or give some error message) otherwise [33, 21]. Assuming that “abort” is refined by anything, we can see that the classical program-algebraic inequality

$$\mathbf{assert} \phi; prog \sqsubseteq prog; \mathbf{assert} \psi$$

has the same meaning as  $\{\phi\}prog\{\psi\}$  does. It *encodes* the Hoare-triple entirely within the programming language and its in-built notion of refinement, thus within the program algebra: if  $\phi$  does not hold in the initial state, then the refinement goes through because the entire left side aborts; if  $\phi$  does hold, then the refinement goes through only if the right-hand side does *not* abort by delivering some final state of *prog* for which the subsequent **assert**  $\psi$  aborts.

## 4.2 Revelations: the modern analogue of assertions for security

By analogy with Sec. 3.2 we can express ignorance-logical properties of program fragments entirely within the programming language by using a special-purpose command encoding the ignorance formulae. There are two main idioms.

In the first we have

$$\mathbf{assert} \phi; prog \sqsubseteq \mathbf{reveal} E; prog, \quad (5)$$

where of course refinement is now understood in the sense of Def. 2, that is non-classically because it preserves ignorance as well as functional properties. If  $\phi$  does not hold in the initial state, then the refinement goes through; otherwise, it goes through only if *prog* reveals the initial value of *E* “anyway” (so that the explicit **reveal** *E* on the right “does no further damage.”) Thus (5) expresses “If  $\phi$  holds initially, then *prog* reveals the initial value of *E*.”

In the second we have

$$\mathbf{assert} \phi; prog \sqsubseteq prog; \mathbf{reveal} E,$$

expressing “If  $\phi$  holds initially, then *prog* reveals the final value of *E*.”

Here are some examples, in which our variables take numeric values. We have the refinement

$$\mathbf{assert} v \neq 0; v := v \times h \sqsubseteq \mathbf{reveal} h; v := v \times h$$

because *h*’s initial value can be deduced by dividing *v*’s final value by its initial value, provided that initial value was not zero. The refinement does not go through without the assertion, since in the *v*-initially-zero case we cannot deduce *h*’s value. For final values we have

$$\mathbf{assert} v = 0; h := v \times h \sqsubseteq h := v \times h; \mathbf{reveal} h$$

because when *v* is zero we can see that *h*’s final value must be zero too, although in that case  $h := v \times h$  still tells us nothing about *h*’s initial value. The refinement does not go through without the assertion, since in the *v*-initially-nonzero case we cannot deduce *h*’s final value without knowing what its initial value was.

Further idioms are possible, for example with revelations on both sides.

## 4.3 A calculus of revelations<sup>7</sup>

We now set out some of the program-algebra associated with revelations; much use of the identities will be made later.

<sup>7</sup> This is the title of the presentation on which the current paper is based [24].

**Replacing one revelation by another** Provided that truth of  $\phi$  implies the equality  $F = f(E)$  for some function  $f$  depending (optionally) on other visible variables, we have

$$\mathbf{assert} \ \phi; \mathbf{reveal} \ E \sqsubseteq \mathbf{reveal} \ F . \quad (6)$$

These are some examples:

$\mathbf{assert} \ h=0; \mathbf{skip} \sqsubseteq \mathbf{reveal} \ h$	Here $f$ is the constant function 0.
$\mathbf{reveal} \ h \sqsubseteq \mathbf{reveal} \ h \oplus 1$	... that is $h-1 \mathbf{max} \ 0$ .
$\mathbf{reveal} \ h \oplus 1 \not\sqsubseteq \mathbf{reveal} \ h$	Initial values 0,1 not distinguished.
$\mathbf{assert} \ h>0; \mathbf{reveal} \ h \oplus 1 \sqsubseteq \mathbf{reveal} \ h$	If $h>0$ then $(\oplus 1)$ is injective.

**Combining revelations** In all cases we have

$$\mathbf{reveal} \ E; \mathbf{reveal} \ F = \mathbf{reveal} \ (E, F) . \quad (7)$$

Here is an example over Booleans and exclusive-or:

$\mathbf{reveal} \ x \oplus y; \mathbf{reveal} \ y \oplus z$	“Write $\oplus$ for exclusive-or”
$= \mathbf{reveal} \ (x \oplus y, y \oplus z)$	“(7)”
$= \mathbf{reveal} \ (x \oplus y, x \oplus z)$	“(6) in both directions”
$= \mathbf{reveal} \ x \oplus y; \mathbf{reveal} \ x \oplus z .$	“(7) in both directions”

**Equivalence with assignment to local visible** In all cases we have

$$\mathbf{reveal} \ E = \llbracket \mathbf{vis} \ v \cdot v := E \rrbracket , \quad (8)$$

highlighting the fact that scope (local vs. global) and visibility (**vis** vs. **hid**) are orthogonal: in spite of the fact that  $v$  is temporary, ultimately “popped from the stack and discarded,” assigning to it while it is there does reveal the value assigned.

An example of this is given in the next section.

#### 4.4 Example: specifications and the encryption lemma

For Booleans, or isomorphically  $\{0, 1\}$ -valued variables  $x, y$  we write  $x \oplus y := E$  to abbreviate the specification statement  $x, y: [x \oplus y = E]$  in the style of the Refinement Calculus [21, 3, 1], thus a command that sets  $x, y$  nondeterministically to make their exclusive-or equal to  $E$ . We define the command to be atomic, so that  $\llbracket x \oplus y := E \rrbracket = \langle x, y: [x \oplus y = E] \rangle$ .

A common pattern for this is  $\llbracket \mathbf{vis} \ v; \mathbf{hid} \ h' \cdot v \oplus h' := h \rrbracket$  in the context of a declaration **hid**  $h$ . It is functionally equivalent to **skip** because it assigns only to local variables; we show it is Shadow-equivalent to **skip** also, i.e. that its effect of assigning to visible  $v$  reveals nothing about  $h$ . We have

$$\begin{aligned}
& \llbracket \mathbf{vis} \ v; \ \mathbf{hid} \ h' \cdot v \oplus h' := h \rrbracket \\
= & \llbracket \mathbf{vis} \ v; \ \mathbf{hid} \ h' \cdot \langle v, h' : [v \oplus h' = h] \rangle \rrbracket && \text{“defined above”} \\
= & \llbracket \mathbf{vis} \ v; \ \mathbf{hid} \ h' \cdot \langle v : \in \{0, 1\}; h' := h \oplus v \rangle \rrbracket && \text{“standard equality” } \star \\
= & \llbracket \mathbf{vis} \ v; \ \mathbf{hid} \ h' \cdot \langle v : \in \{0, 1\}; \langle h' := h \oplus v \rangle \rrbracket && \text{“Lem. 1”} \\
= & \llbracket \mathbf{vis} \ v; \ \mathbf{hid} \ h' \cdot v : \in \{0, 1\}; h' := h \oplus v \rrbracket && \text{“Fig. 1”} \\
= & \llbracket \mathbf{vis} \ v \cdot v : \in \{0, 1\}; \llbracket \mathbf{hid} \ h' \cdot h' := h \oplus v \rrbracket \rrbracket && \text{“move scopes”} \\
= & \llbracket \mathbf{vis} \ v \cdot v : \in \{0, 1\} \rrbracket && \text{“assignment to local hidden is skip”} \\
= & \mathbf{skip} \ . && \text{“assignment of visibles to local visible is skip”}
\end{aligned}$$

But at  $\star$  we could have written instead

$$\begin{aligned}
= & \llbracket \mathbf{vis} \ v; \ \mathbf{hid} \ h' \cdot \langle h' : \in \{0, 1\}; v := h' \oplus h \rangle \rrbracket && \text{“standard equality”} \\
= & \llbracket \mathbf{hid} \ h' \cdot h' : \in \{0, 1\}; \llbracket \mathbf{vis} \ v \cdot v := h' \oplus h \rrbracket \rrbracket && \text{“Lem. 1; Fig. 1; move scopes” } \dagger \\
= & \llbracket \mathbf{hid} \ h' \cdot h' : \in \{0, 1\}; \mathbf{reveal} \ h' \oplus h \rrbracket \ , && \text{“(8)” } \ddagger
\end{aligned}$$

with both  $\dagger, \ddagger$  being interesting formulations often used in protocols: they too are therefore equal to **skip**. Each sets a hidden local Boolean  $h'$  randomly and publishes its exclusive-or with some global hidden  $h$ ; the reasoning above shows rigorously (and formally) that no information about  $h$  is released by doing that.

We call that *The Encryption Lemma* and make much use of it below.

## 5 The two cryptographers<sup>8</sup>

Two cryptographers are about to choose from the trolley, but there are only two desserts there: a lavish cream cake, and a small biscuit. To avoid a series of insincere “after-you” exchanges, they engage in this simple protocol: a single coin is flipped privately between them; each secretly writes his dessert choice on his own napkin if the coin shows heads, or the opposite choice if it shows tails; then they hand their folded-over napkins to the waiter.

If the waiter tells them their napkin-choices differ, they can safely take the two desserts and select their actual preferences once the waiter has gone away; otherwise, to avoid embarrassment, they will forego dessert altogether.

The protocol ensures that neither cryptographer knows the other’s choice before he makes his own choice; and, whatever happens, the waiter does not find out which of them greedily chose the cream cake.<sup>9</sup>

Here is a Shadow-analysis of the protocol. Let the two cryptographers be  $A$  and  $B$  with Boolean variables  $a, b$  recording whether each wants the cream cake respectively. Boolean  $c$  is the shared coin. We do not model the waiter

<sup>8</sup> While based on Chaum’s *Dining Cryptographers* [6], the story for this tiny example has been especially invented to illustrate piecwise construction of a protocol that ultimately will be quite complex. This is the smallest portion, the first step.

<sup>9</sup> The original story ends differently. Without a protocol, the two diners do engage in “after-you” protestations, each believing that the first choice will out of politeness have to be the small cracker; eventually however one diner just chooses the cake. Outraged, the other protests “If  $I$  had chosen first, I’d have taken the cracker!” “Well,” replies the first, “That’s exactly what you’ve got.”

explicitly, because his function of ensuring “oblivious choices” is outside our terms of reference: we do not address the issue of possible protocol violations. The *specification* of the protocol is just

**hid**  $a, b$ : **Bool** ·  
**reveal**  $a \equiv b$  ,

where the declarations of the hidden  $a, b$  are global: we assume them in subsequent manipulations of this example.

The specification says clearly that whether  $a$  and  $b$  agree is to be revealed but nothing else, and in fact it is hard to think of a clearer way of saying this. And although revealing  $a \equiv b$  reveals  $a$ 's value to  $B$  by implication (and vice versa), this does not need any special treatment: it cannot be avoided, and so there is no need to mention it.<sup>10</sup> Thus “but nothing else” above, an informal phrase, carries the sense of “unless unavoidable.”

With the declarations as given, both  $a, b$  hidden, the observer is “the public” who thus cannot observe either one directly. The *implementation* under those same declarations, that is the protocol above, is derived algebraically as follows:

$$\begin{aligned}
& \mathbf{reveal} \ a \equiv b \\
= & \ \mathbf{skip}; \qquad \qquad \qquad \text{“classical reasoning”} \\
& \mathbf{reveal} \ a \equiv b \\
= & \ \llbracket \mathbf{hid} \ c: \mathbf{Bool} \cdot \qquad \qquad \qquad \text{“Encryption Lemma, Sec. 4.4,} \\
& \quad c: \in \mathbf{Bool}; \qquad \qquad \qquad \text{and that } (\mathbf{reveal} \ a \equiv b) = (\mathbf{reveal} \ a \oplus b) \text{ by (6)”} \\
& \quad \mathbf{reveal} \ a \equiv c \\
& \quad \rrbracket; \\
& \mathbf{reveal} \ a \equiv b \\
= & \ \llbracket \mathbf{hid} \ c: \mathbf{Bool} \cdot \qquad \qquad \qquad \text{“adjust scopes”} \\
& \quad c: \in \mathbf{Bool}; \\
& \quad \mathbf{reveal} \ a \equiv c; \\
& \quad \mathbf{reveal} \ a \equiv b \\
& \quad \rrbracket \\
= & \ \llbracket \mathbf{hid} \ c: \mathbf{Bool} \cdot \qquad \qquad \qquad \text{“Reveal Calculus, example following (7):} \\
& \quad c: \in \mathbf{Bool}; \qquad \qquad \qquad (a \equiv c, a \equiv b) \text{ determines } (a \equiv c, b \equiv c) \\
& \quad \mathbf{reveal} \ a \equiv c; \qquad \qquad \qquad \text{and vice versa”} \\
& \quad \mathbf{reveal} \ b \equiv c \\
& \quad \rrbracket .
\end{aligned}$$

<sup>10</sup> We formalise this observation by observing that with the altered declarations **hid**  $a$ ; **vis**  $b$ , that is  $B$ 's point of view, we have the equality  $(\mathbf{reveal} \ a \equiv b) = (\mathbf{reveal} \ a \equiv b; \mathbf{reveal} \ a)$  from (6) and  $b$ 's being visible.

Note (and recall from the introduction) that our strong assumptions for the adversary mean that it is sound to model this distributed protocol with a single sequential program: adversaries' access to the individual threads is modelled by the assumption of perfect recall.

In App. B we illustrate some conventions for abbreviating the presentation of derivations like the one above.

## 6 The three cryptographers<sup>11</sup>

Three cryptographers have just had lunch, and ask for the bill. The waiter says that the bill has already been paid; and the cryptographers want to determine whether one of them paid it or whether it was paid by the NSA. In the case that one of them paid, none of the other cryptographers nor anyone else is to be able to determine which one it was. They proceed as follows.

They are sitting at a round table,<sup>12</sup> and each of the three adjacent pairs flips a coin between them that only that pair can see; thus each cryptographer can see two coins, because he is a member of two such pairs.

Each cryptographer then announces whether he paid; but if the two coins he sees show different faces, he lies. If an odd number of cryptographers claim to have paid, then indeed one did, but no-one (except him) knows who it was; otherwise the lunch was paid for by the NSA.

### 6.1 Helping three cryptographers by considering one at a time

Rather than give a direct derivation in the style of Sec. 5, we build this protocol up from smaller components. We imagine a single cryptographer  $X$  with Boolean  $x$  who has access to two coins  $l, r$  on his left and right. The left one is already flipped; the right one he must flip himself; and then he reveals the exclusive-or of all three values. That amounts to the fragment

$$\begin{array}{l} \mathbf{var} \ l, r: \mathbf{Bool}; \ \mathbf{hid} \ x: \mathbf{Bool} \cdot \\ \left. \begin{array}{l} r: \in \mathbf{Bool}; \\ \mathbf{reveal} \ l \oplus x \oplus r, \end{array} \right\} \text{Protocol } X \end{array}$$

in which for the moment we are not giving the visibility type of  $l, r$ .

<sup>11</sup> Three diners is Chaum's example exactly.

<sup>12</sup> This Arthurian concept is one of Formal Methods' great contributions to computing.



$$\begin{aligned}
(11) &= \llbracket \mathbf{hid} \, l, r: \mathbf{Bool} \cdot && \text{“Revelation Calculus; adjust scopes”} \\
&\quad l: \in \mathbf{Bool}; \\
&\quad r: \in \mathbf{Bool}; \\
&\quad \mathbf{reveal} \, l \oplus (a \oplus b) \oplus r \\
&\quad \rrbracket; \\
&\quad \mathbf{reveal} \, a \oplus b \oplus c \\
&= \mathbf{reveal} \, a \oplus b \oplus c, && \text{“Encryption Lemma for } l, r \text{ together”}
\end{aligned}$$

which is our specification for the Three Cryptographers.

To finish the three cryptographers’ protocol we now simply replace the specification of  $A, B$ ’s sub-protocol by its implementation, which was given earlier. Because the monotonicity property of refinement, actually equality in this case, we do not need to do any further checking. The immediate result, thus obtained “for free” from (10)  $\sqsubseteq$  (9), is

$$\begin{aligned}
&\mathbf{hid} \, a, b, c: \mathbf{Bool} \cdot \\
&\quad \mathbf{reveal} \, a \oplus b \oplus c \\
&= \llbracket \mathbf{hid} \, l, m, r: \mathbf{Bool} \cdot && \text{“replace } A, B \text{ specification above} \\
&\quad l: \in \mathbf{Bool}; && \text{by its implementation from earlier”} \\
&\quad m: \in \mathbf{Bool}; \\
&\quad \mathbf{reveal} \, l \oplus a \oplus m; \\
&\quad r: \in \mathbf{Bool}; \\
&\quad \mathbf{reveal} \, m \oplus b \oplus r; \\
&\quad \mathbf{reveal} \, l \oplus c \oplus r \\
&\quad \rrbracket .
\end{aligned}$$

In Sec. 8 we will do the same step-by-step construction within a loop, thus dealing with arbitrarily many cryptographers.

## 6.2 On expressiveness and “caveats”

An informal specification of the Three Cryptographers might state that whether the NSA paid is to be learned without at the same time learning whether any particular cryptographer paid. Except of course that cryptographer himself, who knows it anyway... Similarly, as we saw, it is unavoidable that each of the Two Cryptographers learns what the other chose, given that he knows his own choice and comes to know whether the other’s differs.

Thus if the first sentence above were formalised, as a logical assertion to be met by the implemented code, it would be too strong. The *caveat* (A) is that when we write (somehow) “for all  $i, j: 1..3$  cryptographer $_i$  does not know whether cryptographer $_j$  paid,” we must add (when we remember) “provided  $i \neq j$ .”

Similarly there is an implicit assumption that at most one cryptographer paid (where “implicit” means “probably we forgot to mention that the first time around”). If two cryptographers paid (B), then the outcome will be “NSA paid”

when in fact it did not: two of the three cryptographers did. So another caveat is added: “Assuming that at most one cryptographer paid. . .”

In fact neither of these two problems bother us if we use refinement. In both cases (A,B) it is obvious from the *specification* **reveal**  $a \oplus b \oplus c$  what behaviour we should expect in all situations, no matter how bizarre, and we do not have to add extra “caveat” clauses to some assertion in order to accommodate them. More importantly, we do not have to worry about whether we have added *enough* caveat clauses. the *Oblivious Transfer Protocol* [27, 10, 28], specified  $a := b_i$  and in which  $A$  reads into  $a$  his choice indexed  $i: \{1, 2\}$  of one of two messages  $b_{1,2}$  that  $B$  holds, without  $A$ ’s learning anything about the message he did not choose and without  $B$ ’s learning anything about the index  $i$  of the choice  $A$  made. Except that in the case  $m_1 = m_2$  we must accept (C) that  $A$  does learn about the message he did not choose, because it is equal to the one he did choose. . .

Again, from the specification  $a := b_i$  it is obvious what happens in (C), and we do not have to introduce caveats to accommodate it. (We gave a rigorous derivation of the Oblivious Transfer Protocol in our earlier report [25].)

### 6.3 On points of view

In the derivation of Sec. 6.1 all three variables  $a, b, c$  are declared hidden, and so our conclusions apply only to adversaries for whom they actually *are* all-three hidden: the general public. To show that as well that no cryptographer learns the thoughts of another, say that  $C$  does not learn about whether  $A$  or  $B$  paid (unless of course  $C$  did pay, in which case he knows that  $A$  and  $B$  did not. . . another caveat we can ignore), we would vary the declarations **hid**  $a, b$ ; **vis**  $c$  and do the derivation under those conditions.

In general, sometimes the same derivation steps go through for all viewpoints; but sometimes they do not, and then we must choose different intermediate refinement steps depending on “who’s looking.” When that happens, it’s equivalent to a case analysis and can fairly be considered a disadvantage: thus we try to find derivations that go through for all viewpoints in the same way.

## 7 Loops and fixed-points

As an example of how loops are treated, the code of (1) in Sec. 1, slightly modified, is shown to satisfy a simple specification: we will prove the equality

```

hid  $h: \mathbb{N}$  .
    reveal  $h \div 2$ ;
     $h := h \bmod 2$ 

= while  $h > 1$  do
     $h := h - 2$ 
end .

```

That is, not only does the loop change the value of  $h$  (in an obvious way), but the repeated conditional tests reveal all but the low bit of  $h$ 's original value. This leaking occurs because it is a refinement (an equality) to unfold a loop, which produces an **if** command, and we have already seen how refinement causes leakage in the conditionals of **if**'s.

Terminating loops are the unique fixed-points of their associated program functionals, and so to prove equality between a loop and some specification it is enough to show the specification satisfies the loop's functional. In the example above, that means we show

```

hid  $h: \mathbb{N} \cdot$ 
  reveal  $h \div 2$ ;
   $h := h \bmod 2$ 

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

```

for which the techniques we have already will suffice.

We start with the right-hand side, since it has more structure (thus suggesting appropriate moves), and the left-hand side is a smaller target:

```

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

= if  $h > 1$  then                                     “add assertion”
  assert  $h > 1$ ;
   $h := h - 2$ ;
  reveal  $h \div 2$ ;
   $h := h \bmod 2$ 
fi

= if  $h > 1$  then                                     “commute commands”
  assert  $h > 1$ ;
  reveal  $(h - 2) \div 2$ ;
   $h := h - 2$ ;
   $h := h \bmod 2$ 
fi

```



=  $\llbracket$  **hid**  $l, r: \mathbf{Bool} \cdot$  “As in Sec. 6.1”  
     **reveal**  $l \oplus (\oplus n: \mathbb{N} \mid 0 \leq n < N \cdot a[n]) \oplus r;$   $\mid$   
     **reveal**  $l \oplus a[N] \oplus r$   
 $\rrbracket$  ,

intending to implement the right-barred portion (i.e. having a “|” at right) as a loop.

For that loop, we refer again to Sec. 6.1, which suggests using a loop body built on the fragment

```

r:∈ Bool;
reveal  $l \oplus a[n] \oplus r;$ 
n:= n+1 ,

```

and it turns out that a **repeat-until** works better in this instance. With that in mind we propose as the next step for the right-barred portion, above, the code

=  $\llbracket$  **vis**  $n: \mathbb{N};$  **hid**  $m \cdot$   
      $m, n := l, 0;$   
     **repeat**  
          $r: \in \mathbf{Bool};$   
         **reveal**  $m \oplus a[n] \oplus r;$   
          $m, n := r, n+1$   
     **until**  $n = N$   $\mid$   
 $\rrbracket$  ,

where we have had to introduce a temporary variable  $m$  to avoid over-writing the initially flipped  $l$  that will be needed at the end by Cryptographer  $N$ . In order to establish this equality, we use the techniques of Sec. 7 to show that the right-barred **repeat-until** is equal to this straight-line fragment:

```

r:∈ Bool;
reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ 
m, n:= r, N

```

For the loop (and its functional) as given, that means we must work towards the program fragment immediately above from this fragment below:

```

r:∈ Bool;
reveal  $m \oplus a[n] \oplus r;$ 
m, n:= r, n+1;

if  $n < N$  then
    r:∈ Bool;
    reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r;$ 
    m, n:= r, N
fi .

```

```

vis  $N:\mathbb{N}$ ; hid  $a[0..N]:\mathbf{Bool}$  ·
    reveal ( $\oplus n:\mathbb{N} \mid 0 \leq n \leq N \cdot a[n]$ )
=  $\llbracket$  vis  $n:\mathbb{N}$ ; hid  $l, m, r:\mathbf{Bool}$  ·                                     “Reasoning in this section”
     $l:\mathbf{Bool}$ ;
     $m, n:=l, 0$ ;
    repeat
       $r:\mathbf{Bool}$ ;
      reveal  $m \oplus a[n] \oplus r$ ;
       $m, n:=r, n+1$ 
    until  $n=N$ ;
    reveal  $l \oplus a[N] \oplus r$ 
 $\rrbracket$ 

```

**Fig. 3.** Specification and implementation for the thousand-and-one cryptographers.

---

Since this derivation is “more of the same” material that we have illustrated in earlier sections, we put it in App. D.

As we remarked in Sec. 6.1, monotonicity of refinement (equivalently, the congruence of our program operators) means that no further reasoning is necessary when we pull the pieces of this section together. That gives the overall equality shown in Fig. 3.

## 9 Advantages; disadvantages; comparisons; conclusions

Provable program refinement is the established scientific technique relating specifications to software code; it is hard to achieve, but brings with it a recognised quality to the workmanship of the code it produces. “Provable security refinement” –or something very like– is the technique we propose here with similar implications to quality.

Our proposed mathematical model for secure refinement has been inspired by a number of other works; our contribution has been to select and fuse several well-known techniques to produce a reasoning tool that can be applied at the source level, and our focus on reasoning *at the level of source code* is the most obvious feature setting us apart from other researchers. Earlier work setting out the theory [23, 24] outlined in more detail how this approach relates to other techniques. In summary it shares many similarities with the Logic of Knowledge [14] but is less general. The semantic technique is based on a version of noninterference which distinguishes “high-security” variables from “low security”, and similar techniques have been suggested by Leino [18] and Sabelfeld [30].

However our overriding motivation is to be able to prove security properties about program code relative to specific assumptions about the operating context.

But code –even without security implications– is hard to understand; with security in the mix it can rise to a higher order of impenetrability, and finding security flaws in such code is an unending task. In 1988 Goldwasser, Micali and Rivest [13] were the first to introduce the idea of “provable security”; it was highly innovative for its time but set the foundations to place security on a scientific footing, and has led to many theoretical results about cryptographic protocols and their relationship to their underlying cryptographic primitives. Although we do not claim a technique as general or widely applicable as Goldwasser and Micali’s work, we do claim a source level method following its fundamental principals which is applicable to some security properties. Here our attacker –a feature of their work– is the programmer who might (maliciously or not) attempt to use a program in a context for which it was not designed; secure refinement means exactly that the implemented code has the same (or better) security properties as the specification. The crucial advantage of this is that the specification suffers exactly the same security flaws as the implementation, whatever they might be, and is exposed to *the same attacks*. Specifications by tradition only state the designer’s ideal requirements and avoid the issues of implementation, and –as with traditional functional properties– it is only at the abstract level that designers have any chance of understanding their designs: this is where security issues should be considered.

## References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *ICALP '06: Proceedings (Part II) of the 33rd International Colloquium on Automata, Languages and Programming*, pages 107–118. Springer, 2006.
3. R.-J.R. Back. On the correctness of refinement steps in program development. Report A-1978-4, Dept Comp Sci, Univ Helsinki, 1978.
4. R.-J.R. Back. A calculus of refinements for program derivations. *Acta Inf*, 25:593–624, 1988.
5. Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and its applications. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, page 322. Springer, 2000.
6. D. Chaum. The Dining Cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.
7. Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 198–209, New York, NY, USA, 2004. ACM.
8. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. K. Engelhardt, Y. Moses, and R. van der Meyden. Unpublished report, Univ NSW, 2005.
10. Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.
11. R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

12. J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc IEEE Symp on Security and Privacy*, pages 75–86, 1984.
13. S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen message attacks. *SIAM J. on Computing*, 17:281–308, 1988.
14. J.Y. Halpern and K.R. O’Neill. Anonymity and information hiding in multiagent systems. In *Proc 16th IEEE Computer Security Foundations Workshop*, pages 75–88, 2003.
15. C.A.R. Hoare. An axiomatic basis for computer programming. *Comm ACM*, 12(10):576–80, 583, October 1969.
16. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
17. J. Jacob. Security specifications. In *IEEE Symposium on Security and Privacy*, pages 14–23, 1988.
18. K.R.M. Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–38, 2000.
19. Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL ’05: Proc. 32nd ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, pages 158–170, New York, NY, USA, 2005. ACM.
20. Heiko Mantel. Preserving information flow properties under refinement. In *Proc IEEE Symp Security and Privacy*, pages 78–91, 2001.
21. C.C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994. [web.comlab.ox.ac.uk/oucl/publications/books/PfS/](http://web.comlab.ox.ac.uk/oucl/publications/books/PfS/).
22. C.C. Morgan. Of probabilistic *wp* and *CSP*. In A. Abdallah, C.B. Jones, and J.W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years*. Springer, 2005.
23. C.C. Morgan. *The Shadow Knows*: Refinement of ignorance in sequential programs. In T. Uustalu, editor, *Math Prog Construction*, volume 4014 of *Springer*, pages 359–78. Springer, 2006. *Treats Dining Cryptographers*.
24. C.C. Morgan. A calculus of revelations, 2008. Presented at *VSTTE ’08*, Toronto. <http://www.cs.stevens.edu/~naumann/vstte-theory-2008/>.
25. C.C. Morgan. *The Shadow Knows*: Refinement of ignorance in sequential programs. *Science of Computer Programming*, 74(8), 2009. *Treats Oblivious Transfer*.
26. C.C. Morgan and A.K. McIver. Unifying *wp* and *wlp*. *Inf Proc Lett*, 20(3):159–64, 1996. Available at [?, key MM95].
27. Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University, 1981. Available at [eprint.iacr.org/2005/187](http://eprint.iacr.org/2005/187).
28. R. Rivest. Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser. Technical report, M.I.T., 1999. [//theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf](http://theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf).
29. A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996.
30. A. Sabelfeld and D. Sands. A PER model of secure information flow. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
31. M.B. Smyth. Power domains. *Jnl Comp Sys Sci*, 16:23–36, 1978.
32. Niklaus Wirth. Program development by stepwise refinement. *Comm ACM*, 14(4):221–7, 1971.
33. Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432, June 1966.

## A Proof of Lem. 1 from Sec. 2.3

We take two standard programs  $r_{\{1,2\}}$  and compare the two forms of sequential composition. First, with overall atomicity we have

$$\begin{aligned}
& \langle r_1; r_2 \rangle . v . h . H \ni (v', h', H') \\
\text{iff} & \quad (r_1; r_2) . v . h \ni (v', h') \quad \text{“Def. 1”} \\
& \text{and } H' = \{h'' : H, h' : \mathcal{H} \mid (r_1; r_2) . v . h'' \ni (v', h') \cdot h'\} \\
\text{iff} & \quad \text{“forward relational composition”} \\
& (\exists v^b : \mathcal{V}, h^b : \mathcal{H} \mid r_1 . v . h \ni (v^b, h^b) \wedge r_2 . v^b . h^b \ni (v', h')) \\
& \text{and } H' = \{h'' : H, h' : \mathcal{H}, v^\sharp : \mathcal{V}, h^\sharp : \mathcal{H} \\
& \quad \mid r_1 . v . h'' \ni (v^\sharp, h^\sharp) \wedge r_2 . v^\sharp . h^\sharp \ni (v', h') \\
& \quad \cdot h'\}
\end{aligned}$$

On the other hand, with piecewise atomicity we have

$$\begin{aligned}
& \langle \langle r_1 \rangle; \langle r_2 \rangle \rangle . v . h . H \ni (v', h', H') \\
\text{iff} & \quad \text{“forward relational composition”} \\
& (\exists v^\natural : \mathcal{V}, h^\natural : \mathcal{H}, H^\natural : \mathbb{P}\mathcal{H} \mid \\
& \quad \langle r_1 \rangle . v . h . H \ni (v^\natural, h^\natural, H^\natural) \wedge \langle r_2 \rangle . v^\natural . h^\natural . H^\natural \ni (v', h', H')) \\
\text{iff} & \quad \text{“Def. 1”} \\
& (\exists v^\natural : \mathcal{V}, h^\natural : \mathcal{H}, H^\natural : \mathbb{P}\mathcal{H} \mid \\
& \quad r_1 . v . h \ni (v^\natural, h^\natural) \\
& \quad \wedge H^\natural = \{h^+ : \mathcal{H}, h^- : H \mid r_1 . v . h^- \ni (v^\natural, h^+) \cdot h^+\} \\
& \quad \wedge r_2 . v^\natural . h^\natural \ni (v', h') \\
& \quad \wedge H' = \{h^+ : \mathcal{H}, h^- : H^\natural \mid r_2 . v^\natural . h^- \ni (v', h^+) \cdot h^+\}) \\
\text{iff} & \quad \text{“propositional calculus; eliminate } H^\natural\text{”} \\
& (\exists v^\natural : \mathcal{V}, h^\natural : \mathcal{H} \mid \\
& \quad r_1 . v . h \ni (v^\natural, h^\natural) \wedge r_2 . v^\natural . h^\natural \ni (v', h') \\
& \quad \wedge H' = \{h^+ : \mathcal{H}, h^- : \{h^+ : \mathcal{H}, h^- : H \mid r_1 . v . h^- \ni (v^\natural, h^+) \cdot h^+\} \\
& \quad \mid r_2 . v^\natural . h^- \ni (v', h^+) \\
& \quad \cdot h^+\} \\
\text{iff} & \quad \text{“rename bound variables”} \\
& (\exists v^\natural : \mathcal{V}, h^\natural : \mathcal{H} \mid \\
& \quad r_1 . v . h \ni (v^\natural, h^\natural) \wedge r_2 . v^\natural . h^\natural \ni (v', h') \\
& \quad \wedge H' = \{h^+ : \mathcal{H}, h^- : \{h^\pm : \mathcal{H}, h^\mp : H \mid r_1 . v . h^\mp \ni (v^\natural, h^\pm) \cdot h^\pm\} \\
& \quad \mid r_2 . v^\natural . h^- \ni (v', h^+) \\
& \quad \cdot h^+\} \\
\text{iff} & \quad \text{“collapse comprehensions”}
\end{aligned}$$

$$\begin{aligned}
& (\exists v^{\natural}: \mathcal{V}, h^{\natural}: \mathcal{H} \mid \\
& \quad r_1.v.h \ni (v^{\natural}, h^{\natural}) \wedge r_2.v^{\natural}.h^{\natural} \ni (v', h') \\
& \quad \wedge H' = \{h^{\pm}: \mathcal{H}, h^{\pm}: \mathcal{H}, h^{\mp}: H \\
& \quad \quad \mid r_1.v.h^{\mp} \ni (v^{\natural}, h^{\pm}) \wedge r_2.v^{\natural}.h^{\pm} \ni (v', h^{\pm}) \\
& \quad \quad \cdot h^{\pm}\} \quad )
\end{aligned}$$

iff “rename bound variables”

$$\begin{aligned}
& (\exists v^{\flat}: \mathcal{V}, h^{\flat}: \mathcal{H} \mid \\
& \quad r_1.v.h \ni (v^{\flat}, h^{\flat}) \wedge r_2.v^{\flat}.h^{\flat} \ni (v', h') \\
& \quad \wedge H' = \{h'': H, h': \mathcal{H}, h^{\sharp}: \mathcal{H} \\
& \quad \quad \mid r_1.v.h'' \ni (v^{\flat}, h^{\sharp}) \wedge r_2.v^{\flat}.h^{\sharp} \ni (v', h') \\
& \quad \quad \cdot h'\} \quad )
\end{aligned}$$

which is pretty close to the first case. The difference is in the scoping of the  $\exists$  because in the first case  $v^{\flat}$  and  $v^{\sharp}$  are independent whereas in the second case they are both  $v^{\flat}$ .

To bring them together we make the assumption that the intermediate  $v^{\flat}$  is determined by the extremal  $v$ 's alone — i.e. that we have

$$(\exists h, h^{\flat}, h': \mathcal{H} \mid r_1.v.h \ni (v^{\flat}, h^{\flat}) \wedge r_2.v^{\flat}.h^{\flat} \ni (v', h')) \quad \Rightarrow \quad v^{\flat} = D.v.v'$$

for some fixed function  $D$  (for “determined”). That enables us to take the first case further, as follows:

$$\begin{aligned}
& \text{iff} \quad \text{“introduce } D\text{”} \\
& \quad (\exists h^{\flat}: \mathcal{H} \mid r_1.v.h \ni (D.v.v', h^{\flat}) \wedge r_2.(D.v.v').h^{\flat} \ni (v', h')) \\
& \quad \text{and } H' = \{h'': H, h': \mathcal{H}, h^{\sharp}: \mathcal{H} \\
& \quad \quad \mid r_1.v.h'' \ni (D.v.v', h^{\sharp}) \wedge r_2.(D.v.v').h^{\sharp} \ni (v', h') \\
& \quad \quad \cdot h'\}
\end{aligned}$$

And for the second case we can continue

$$\begin{aligned}
& \text{iff} \quad \text{“introduce } D\text{”} \\
& \quad (\exists h^{\flat}: \mathcal{H} \mid \\
& \quad \quad r_1.v.h \ni (D.v.v', h^{\flat}) \wedge r_2.(D.v.v').h^{\flat} \ni (v', h') \\
& \quad \quad \wedge H' = \{h'': H, h': \mathcal{H}, h^{\sharp}: \mathcal{H} \\
& \quad \quad \quad \mid r_1.v.h'' \ni (D.v.v', h^{\sharp}) \wedge r_2.(D.v.v').h^{\sharp} \ni (v', h') \\
& \quad \quad \quad \cdot h'\} \quad )
\end{aligned}$$

which is equivalent to the first because  $h^{\flat}$  is not free in the second conjunct.

That completes the proof of Lem. 1.

## B Abbreviated/annotated derivations

It’s helpful to use annotation bars to highlight the portions of code that change. Bars on the right select lines that will change between this step and the next; bars on the left select lines that did change from the last step to this one. Here is an annotated version of the Two Cryptographers’ derivation of Sec. 5:

$$\begin{aligned}
& \text{reveal } a \equiv b \quad | \\
= & \left| \begin{array}{l} \text{skip;} \\ \text{reveal } a \equiv b \end{array} \right| \quad \text{“classical reasoning”} \\
= & \left| \begin{array}{l} \ll \text{hid } c: \mathbf{Bool} \cdot \\ \quad c: \in \mathbf{Bool}; \\ \quad \text{reveal } a \equiv c \\ \quad \gg; \\ \text{reveal } a \equiv b \end{array} \right| \quad \text{“Encryption Lemma, Sec. 4.4”} \\
= & \left| \begin{array}{l} \ll \text{hid } c: \mathbf{Bool} \cdot \\ \quad c: \in \mathbf{Bool}; \\ \quad \text{reveal } a \equiv c; \\ \quad \text{reveal } a \equiv b \\ \quad \gg \end{array} \right| \quad \text{“adjust scopes”} \\
= & \left| \begin{array}{l} \ll \text{hid } c: \mathbf{Bool} \cdot \\ \quad c: \in \mathbf{Bool}; \\ \quad \text{reveal } a \equiv c; \\ \quad \text{reveal } b \equiv c \\ \quad \gg \end{array} \right| \cdot \quad \begin{array}{l} \text{“Reveal Calculus, example following (7):} \\ \quad (a \equiv c, a \equiv b) \text{ determines } (a \equiv c, b \equiv c) \\ \quad \text{and vice versa”} \end{array}
\end{aligned}$$

Further compression is possible when we note that in derivations of large programs there will be many small steps, and consequently much “dormant” text that does not change from one step to the next. In that case we simply omit the lines that have no bar of either kind. Here is the above Two Cryptographers’ derivation with the dormant text removed:

$$\begin{aligned}
& \text{reveal } a \equiv b \quad | \\
= & \left| \begin{array}{l} \text{skip;} \\ \text{reveal } a \equiv b \end{array} \right| \quad \text{“classical reasoning”} \\
= & \left| \begin{array}{l} \ll \text{hid } c: \mathbf{Bool} \cdot \\ \quad c: \in \mathbf{Bool}; \\ \quad \text{reveal } a \equiv c \\ \quad \gg; \\ \text{reveal } a \equiv b \end{array} \right| \quad \text{“Encryption Lemma, Sec. 4.4”} \\
= & \left| \begin{array}{l} \text{reveal } a \equiv c; \\ \text{reveal } a \equiv b \\ \gg \end{array} \right| \quad \text{“adjust scopes”} \\
= & \left| \begin{array}{l} \text{reveal } a \equiv c; \\ \text{reveal } b \equiv c \end{array} \right| \quad \begin{array}{l} \text{“Reveal Calculus, example following (7):} \\ \quad (a \equiv c, a \equiv b) \text{ determines } (a \equiv c, b \equiv c) \\ \quad \text{and vice versa”} \end{array}
\end{aligned}$$

Finally, there may be points at which previously dormant lines must be re-introduced, either because they have “woken up” and are now to take part in a step, or because they are needed in order to round off the whole derivation. For that we use double bars on the left, as in this final step concluding the derivation above:

$$= \left\| \left[ \begin{array}{l} \mathbf{hid} \ c: \mathbf{Bool} \cdot \\ c: \in \mathbf{Bool}; \\ \mathbf{reveal} \ a \equiv c; \\ \mathbf{reveal} \ b \equiv c \end{array} \right. \right. \quad \text{“Restore surrounding context”} \\ \left. \right] \cdot$$

## C Abbreviated derivation of loop from Sec. 7

This is the abbreviated form of the derivation establishing

$$\mathbf{reveal} \ h \div 2; \ h := h \bmod 2 \quad \sqsubseteq \quad \mathbf{while} \ h > 1 \ \mathbf{do} \ h := h - 2 \ \mathbf{end}$$

from Sec. 7. It begins with  $W.lhs$ , where  $W$  is the fixed-point functional for the while-loop, and ends with just  $lhs$ .

$$\begin{aligned} & \mathbf{if} \ h > 1 \ \mathbf{then} && \text{“This is } W.lhs \text{”} \\ & \quad h := h - 2; \\ & \quad \mathbf{reveal} \ h \div 2; \\ & \quad h := h \bmod 2 \\ & \mathbf{fi} \\ \\ = & \left| \begin{array}{l} \mathbf{assert} \ h > 1; \\ h := h - 2; \\ \mathbf{reveal} \ h \div 2; \end{array} \right. && \text{“add assertion”} \\ \\ = & \left| \begin{array}{l} \mathbf{assert} \ h > 1; \\ \mathbf{reveal} \ (h - 2) \div 2; \\ h := h - 2; \end{array} \right. && \text{“commute commands”} \\ \\ = & \left| \begin{array}{l} \mathbf{reveal} \ h \div 2; \end{array} \right. && \text{“Revelation calculus; classical reasoning; remove assertion”} \\ \\ = & \left\| \begin{array}{l} \dots \\ \mathbf{else} \\ \quad \mathbf{assert} \ 0 \leq h \leq 1; \\ \quad \mathbf{reveal} \ h \div 2; \\ \quad h := h \bmod 2 \\ \mathbf{fi} \end{array} \right. && \text{“Add assertion; Revelation Calculus; classical reasoning”} \end{aligned}$$

$$\begin{aligned}
&= \left| \begin{array}{l} \mathbf{if } h > 1 \mathbf{ then skip else skip fi}; \\ \mathbf{reveal } h \dot{\div} 2; \\ h := h \bmod 2 \end{array} \right| \quad \text{“Remove assertion; classical reasoning”} \\
&= \left| \begin{array}{l} \mathbf{reveal } h > 1; \\ \mathbf{reveal } h \dot{\div} 2; \end{array} \right| \quad \text{“Revelation calculus”} \\
&= \left| \begin{array}{l} \mathbf{reveal } h \dot{\div} 2; \\ h := h \bmod 2, \end{array} \right| \quad \text{“Revelation calculus”}
\end{aligned}$$

which is *lhs* again.

## D Derivation of the cryptographers’ repeat-until loop

A detail we suppressed in Sec. 8 is that the loop is intended to execute only when  $0 \leq n < N$ , and that our postulated equality is unlikely to hold otherwise; thus we will actually be proving a refinement here. Normally in program semantics one is interested in *least* fixed-points, and the technique  $f.x \sqsubseteq x \Rightarrow \mu.f \sqsubseteq x$  is not on its own sufficient, since it (in a looping case) would prove that the specification is a refinement of the loop, rather than the other way around. That is why the well known technique of loop variants is necessary.

In our case, however, we are assuming our loop terminates, and so we can exploit the equality of its least- and greatest fixed-points to use as our tool the fact that  $x \sqsubseteq f.x \Rightarrow x \sqsubseteq \nu.f \Rightarrow x \sqsubseteq \mu.f$ . That is what we prove below.

We express our reliance on  $0 \leq n < N$  by adding an assertion to the specification, and proceed to show that the code immediately below is a refinement of its left-barred portion:

$$\begin{aligned}
& \left| \begin{array}{l} r : \in \mathbf{Bool}; \\ \mathbf{reveal } m \oplus a[n] \oplus r; \\ m, n := r, n+1; \\ \mathbf{if } n < N \mathbf{ then} \\ \quad \mathbf{assert } 0 \leq n < N; \\ \quad r : \in \mathbf{Bool}; \\ \quad \mathbf{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r; \\ \quad m, n := r, N \\ \mathbf{fi} \end{array} \right| \\
\sqsubset & \left| \begin{array}{l} \mathbf{assert } 0 \leq n < N; \\ r : \in \mathbf{Bool}; \\ \mathbf{reveal } m \oplus a[n] \oplus r; \\ m, n := r, n+1; \\ \mathbf{if } n < N \mathbf{ then} \\ \quad r : \in \mathbf{Bool}; \\ \quad \mathbf{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r; \\ \quad m, n := r, N \\ \mathbf{fi} \end{array} \right| \quad \text{“Assertion more restrictive in this position”}
\end{aligned}$$

```

=   assert  $0 \leq n < N$ ;           "Make else explicit;
     $r \in \mathbf{Bool}$ ;                exploit initial assertion to introduce subsequent one"
    reveal  $m \oplus a[n] \oplus r$ ;
     $m, n := r, n+1$ ;
    if  $n < N$  then
       $r \in \mathbf{Bool}$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;
       $m, n := r, N$ 
    else
      assert  $n = N$ 
    fi

=   assert  $0 \leq n < N$ ;           "Use assertion to rationalise if-branches,
     $r \in \mathbf{Bool}$ ;                by adding trivial code to else"
    reveal  $m \oplus a[n] \oplus r$ ;
     $m, n := r, n+1$ ;
    if  $n < N$  then
       $r \in \mathbf{Bool}$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;
       $m, n := r, N$ 
    else
      assert  $n = N$ ;
       $n := N$                        // Has no effect since  $n = N$  already.
    fi

=   assert  $0 \leq n < N$ ;           "Remove increment of  $n$ , since it is overwritten anyway"
     $r \in \mathbf{Bool}$ ;
    reveal  $m \oplus a[n] \oplus r$ ;
     $m := r$ ;
    if  $n+1 < N$  then
       $r \in \mathbf{Bool}$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r$ ;
       $m, n := r, N$ 
    else
      assert  $n+1 = N$ ;
       $n := N$ 
    fi

```

= **assert**  $0 \leq n < N$ ; “Move code into **if**-branches”  
**if**  $n+1 < N$  **then**  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r$ ;  
|  $m := r$ ;  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus (\oplus i: \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r$ ;  
|  $m, n := r, N$   
**else**  
| **assert**  $n+1 = N$ ;  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r$ ;  
|  $m := r$ ;  
|  $n := N$   
**fi**

= **assert**  $0 \leq n < N$ ; “Exploit assertion in **else** branch, then remove it”  
**if**  $n+1 < N$  **then**  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r$ ;  
|  $m := r$ ;  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus (\oplus i: \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r$ ;  
|  $m, n := r, N$   
**else**  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;  
|  $m := r$ ;  
|  $n := N$   
**fi**

= **assert**  $0 \leq n < N$ ; “Move code out of **if**-branches”  
**if**  $n+1 < N$  **then**  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r$ ;  
|  $m := r$ ;  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus (\oplus i: \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r$ ;  
**else**  
|  $r \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$   
**fi**;  
|  $m, n := r, N$

= **assert**  $0 \leq n < N$ ; “Introduce temporary hidden  $r'$ ”  
**if**  $n+1 < N$  **then**  
| **[[** **hid**  $r' : \mathbf{Bool}$  ·  
|  $r' : \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r'$ ;  
|  $m := r'$ ;  
| **]]**;  
|  $r : \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus (\oplus i : \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r$ ;  
**else**  
 $r : \in \mathbf{Bool}$ ;  
**reveal**  $m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$   
**fi**;  
 $m, n := r, N$

= **assert**  $0 \leq n < N$ ; “Reorder; standard reasoning”  
**if**  $n+1 < N$  **then**  
|  $r : \in \mathbf{Bool}$ ;  
| **[[** **hid**  $r' : \mathbf{Bool}$  ·  
|  $r' : \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r'$ ;  
| **reveal**  $r' \oplus (\oplus i : \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r$ ;  
|  $m := r'$   
| **]]**  
**else**  
 $r : \in \mathbf{Bool}$ ;  
**reveal**  $m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$   
**fi**;  
 $m, n := r, N$

= **assert**  $0 \leq n < N$ ; “Revelation calculus”  
**if**  $n+1 < N$  **then**  
|  $r : \in \mathbf{Bool}$ ;  
| **[[** **hid**  $r' : \mathbf{Bool}$  ·  
|  $r' : \in \mathbf{Bool}$ ;  
| **reveal**  $m \oplus a[n] \oplus r'$ ;  
| **reveal**  $m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;  
|  $m := r'$   
| **]]**  
**else**  
 $r : \in \mathbf{Bool}$ ;  
**reveal**  $m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$   
**fi**;  
 $m, n := r, N$

```

=   assert  $0 \leq n < N$ ;           “Eliminate superfluous assignment to  $m$ ; reorder”
    if  $n+1 < N$  then
       $r \in \mathbf{Bool}$ ;
       $\llbracket$  hid  $r' \in \mathbf{Bool} \cdot$ 
         $r' \in \mathbf{Bool}$ ;
        reveal  $m \oplus a[n] \oplus r'$ 
       $\rrbracket$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;
    else
       $r \in \mathbf{Bool}$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ 
    fi;
     $m, n := r, N$ 

=   assert  $0 \leq n < N$ ;           “Encryption Lemma”
    if  $n+1 < N$  then
       $r \in \mathbf{Bool}$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ 
    else
       $r \in \mathbf{Bool}$ ;
      reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ 
    fi;
     $m, n := r, N$ 

=   assert  $0 \leq n < N$ ;           “Collapse if”
     $r \in \mathbf{Bool}$ ;
    reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;
     $m, n := r, N$ 

```

and we are done. An abbreviated version of the above is given in App. E.

The equality in Fig. 3 (rather than refinement, which is what we have proved here) is justified by both programs’ being deterministic and terminating: in that case refinement reduces to equality. But even without that, refinement would still be good enough.

## E Abbreviated version of App. D

```

 $r \in \mathbf{Bool}$ ;
reveal  $m \oplus a[n] \oplus r$ ;
 $m, n := r, n+1$ ;
if  $n < N$  then
  assert  $0 \leq n < N$ ;
   $r \in \mathbf{Bool}$ ;
  reveal  $m \oplus (\oplus i: \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r$ ;
   $m, n := r, N$ 
fi

```



$$\begin{aligned}
&= \left| \begin{array}{l} \llbracket \text{hid } r' : \mathbf{Bool} \cdot \\ \quad r' : \in \mathbf{Bool}; \\ \quad \text{reveal } m \oplus a[n] \oplus r'; \\ \quad m := r'; \\ \quad \rrbracket; \\ r : \in \mathbf{Bool}; \end{array} \right| \quad \text{“Introduce temporary hidden } r' \text{”} \\
&= \left| \begin{array}{l} r : \in \mathbf{Bool}; \\ \dots \\ \text{reveal } r' \oplus (\oplus i : \mathbb{N} \mid n+1 \leq i < N \cdot a[i]) \oplus r; \\ \quad m := r' \\ \quad \rrbracket \end{array} \right| \quad \text{“Reorder; standard reasoning”} \\
&= \left| \begin{array}{l} r : \in \mathbf{Bool}; \\ \dots \\ \text{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r; \\ \quad m := r' \\ \quad \rrbracket \end{array} \right| \quad \text{“Revelation calculus”} \\
&= \left| \begin{array}{l} r : \in \mathbf{Bool}; \\ \llbracket \text{hid } r' : \mathbf{Bool} \cdot \\ \quad r' : \in \mathbf{Bool}; \\ \quad \text{reveal } m \oplus a[n] \oplus r' \\ \quad \rrbracket; \\ \text{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r; \end{array} \right| \quad \begin{array}{l} \text{“Eliminate superfluous} \\ \text{assignment to } m; \\ \text{reorder”} \end{array} \\
&= \left| \begin{array}{l} \text{if } n+1 < N \text{ then} \\ \quad r : \in \mathbf{Bool}; \\ \\ \quad \text{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r \\ \text{else} \\ \quad r : \in \mathbf{Bool}; \\ \quad \text{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r \\ \text{fi}; \end{array} \right| \quad \text{“Encryption Lemma”} \\
&= \left| \begin{array}{l} \text{assert } 0 \leq n < N; \\ r : \in \mathbf{Bool}; \\ \text{reveal } m \oplus (\oplus i : \mathbb{N} \mid n \leq i < N \cdot a[i]) \oplus r; \\ m, n := r, N \end{array} \right| \quad \text{“Collapse if”}
\end{aligned}$$

Although this is not easily browsed, by suppressing parts of the code that are not altered it gives a better indication the amount of reasoning involved: it is not very much.