The Secret Art of Computer Programming

Annabelle K. McIver⋆

Dept. Computer Science, Macquarie University, NSW 2109 Australia

Abstract. "Classical" program development by refinement [12,2,3] is a technique for ensuring that source-level program code remains faithful to the semantic goals set out in its corresponding specification. Until recently the method has not extended to security-style properties, principally because classical refinement semantics is inadequate in security contexts [7].

The Shadow semantics introduced by Morgan [13] is an abstraction of probabilistic program semantics [11], and is rich enough to distinguish between refinements that do preserve noninterference security properties and those that don't. In this paper we give a formal development of Private Information Retrieval [4]; in doing so we extend the general theory of secure refinement by introducing a new kind of security annotation for programs.

Keywords: Proofs of security, program semantics, compositional security, refinement of ignorance.

1 Introduction

Abstraction and refinement are together one of the core techniques in any formal verifier's toolkit. Yet to date they are rarely applied in security analysis; indeed until recently refinement and security were considered uneasy bedfellows, with any attempt to reconcile the two bound for paradox and confusion [7].

Morgan's *Shadow semantics* [13] for "noninterference security" based originally on an abstraction of probabilistic program semantics [11] succeeded after all in bringing about a détente between nondeterminism (the mathematical encapsulation of abstraction) and hidden state (the mathematical encapsulation of secrets). *Noninterference* security [6] formalises our intuitive notion of "security leaks" — in programming terms it characterises scenarios where data intended to be kept private are exposed by inadvertent correlations with other observable program behaviour. By a careful treatment of nondeterminism and hidden state, the Shadow semantics automatically selects refinements which are "security-aware": a valid "secure refinement" is now not only functionally- but also security-wise compatible with its specification. In some cases this might mean absolute confidentiality; but there are many applications where the required functionality logically forces a disclosure, at least in part. Shadow security proofs guarantee therefore that any implementation leaks *no more* than the specification demands.

The Shadow approach is distinguished from other methods for security analysis in its emphasis on compositionality and the development-by-hierarchy that compositionality supports. Specifications are now programs too –though most likely inefficient and tacit as to algorithmic detail– yet as we have learned from many years' experience with the refinement calculus, a focus on what we want pays off "in spades" for understanding systems. Adding detail devolves to the validation of refinement steps, each one small enough for the proofs to be – almost– automatic, and furthermore achieved at the source level. And, as for classical refinement, we often call on specifications of sub-protocols wherever this simplifies the reasoning, leading to the method's ability to accommodate protocols of unbounded state [10].

Our contribution. in this paper is a formal development of a scheme for Private Information Retrieval in public databases [4]. In doing so we extend the theory by the introduction of "visibility annotations" for reasoning about the extent to which a secret is revealed during program execution.

We begin with a summary and commentary on the basics for non-interference security using the Shadow semantics. Throughout we use left-associating dot for function application, so that $f.x.y$ means $(f(x))(y)$ or $f(x,y)$, and we take (un-)Currying for granted where necessary. Comprehensions/quantifications are written uniformly, as $(Qx{:}T|R{\cdot}E)$ for quantifier 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 "$(Q$" is "$\{$" and the closing "$)$" is "$\}$" so that e.g. the comprehension $\{x,y{:}\mathbb{N} \mid y{=}2^x \ \cdot\ yz\}$ is the set of numbers $z, 2z, 4z, \cdots$.

2   Semantics for Programming with Secrets

A non-interference -secure program is one where an attacker (discussed below) cannot infer "hidden" variables' initial values from "visible" variables' values (initial or final). With just two variables v, h of class visible, hidden resp. suppose a possibly nondeterministic program r takes initial states (v, h) to sets of final visible states v' and so is of type $\mathcal{V} \to \mathcal{H} \to \mathbb{P}\mathcal{V}$, where $\mathcal{V}, \mathcal{H}$ are the value sets corresponding to the types of v, h. Such a program r is then *non-interference -secure* just when for any initial visible the set of possible final visibles is independent of the initial hidden [8,15], that is for any $v{:}\mathcal{V}$ we have $\big(\forall h_0, h_1{:}\mathcal{H} \cdot r.v.h_0 = r.v.h_1\big)$.

In our approach [13] we extend this view, in several stages. The first is to concentrate on final- (rather than initial) hidden values and therefore to model programs as $\mathcal{V}{\to}\mathcal{H}{\to}\mathbb{P}(\mathcal{V}{\times}\mathcal{H})$. For two such programs $r_{\{1,2\}}$ we say that $r_1 \sqsubseteq r_2$, that r_1 "is securely refined by" r_2, whenever both the following hold:

(i) For any initial state v, h each possible r_2 outcome v', h' is also a possible r_1 outcome, that is for all $v{:}\mathcal{V}$ and $h{:}\mathcal{H}$ we have $r_1.v.h \supseteq r_2.v.h$.
This is the classical "can reduce nondeterminism" form of refinement.

(ii) For all $v\colon\mathcal{V}, h\colon\mathcal{H}$ and $v'\colon\mathcal{V}$ satisfying $\big(\exists h_2'\colon\mathcal{H}\cdot(v',h_2')\in r_2.v.h\big)$, we have
that $(v',h')\in r_1.v.h$ implies $(v',h')\in r_2.v.h$ for all $h'\colon\mathcal{H}$.
This second condition says that for any observed visibles v, v' and any initial h the attacker's "deductive powers" w.r.t. final h''s cannot be improved by refinement: there can only be more possibilities, never fewer.

In this simple setting, as an example restrict all our variables' types so that $\mathcal{V}=\mathcal{H}=\{0,1\}$, and let r_1 be the program that can produce from any initial values (v, h) any one of the four possible (v', h') final values in $\mathcal{V}\times\mathcal{H}$ (so that the final values of v and h are uncorrelated). Then the program r_2 that can produce only the two final values $\{(0,0),(0,1)\}$ is a secure refinement of r_1; but the program r_3 that produces only the two final values $\{(0,0),(1,1)\}$ is not a secure refinement (although it is a classical one).

The difference between r_2 and r_3 is that although r_2 reduces r_1's visible nondeterminism, it does not affect the hidden nondeterminism in h'. In r_3, however, variables v' and h' have become correlated.

2.1   The Shadow H of h Records h's Inferred Values

In r_1 above the set of possible final values of h' was $\{0,1\}$ for each v' separately. This set is called "The Shadow," and represents explicitly an attacker's ignorance of h': it is the smallest set of possibilities he can infer. In r_2 that shadow was the same; but in r_3 the shadow was smaller, just $\{v'\}$ for each v', and that is why r_3 was not a secure refinement of r_1.

In the shadow semantics we track this inference, so that our program state becomes a triple (v, h, H) with H a subset of $\mathcal{H}$ — and in each triple the H contains exactly those (other) values that h *might have had*, including the one it actually *does* have. The (extended) output triples of the three example programs are then respectively

$$r_1 \text{ --- } \{(0,0,\{0,1\}),\ (0,1,\{0,1\}),\ (1,0,\{0,1\}),\ (1,1,\{0,1\})\}$$
$$r_2 \text{ --- } \{(0,0,\{0,1\}),\ (0,1,\{0,1\})\}$$
$$r_3 \text{ --- } \{(0,0,\{0\}),\ (1,1,\{1\})\}\ ,$$

and we have $r_1\sqsubseteq r_2$ because r_1's set of outcomes includes all of r_2's. But for r_3 we find that its outcome $(0,0,\{0\})$ 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, again, is why $r_1\not\sqsubseteq r_3$.

For sequential composition of shadow-enhanced programs, not only final- but also initial triples (v, h, H) must be dealt with: the final triples of a first component become initial triples for a second. We now define the shadow semantics exactly, in stages, by showing how those triples are generated for straight-line programs.

2.2   The Shadow Semantics of Atomic Programs

A classical program r is an input-output relation between $\mathcal{V}\times\mathcal{H}$ -pairs. Considered as a single, atomic action its shadow-enhanced semantics $\mathsf{addShadow}.r$ is a relation between $\mathcal{V}\times\mathcal{H}\times\mathbb{P}\mathcal{H}$ -triples and is defined as follows:

	Program P	Semantics $[\![P]\!].v.h.H$	
Publish a value	**reveal** $E.v.h$	$\{\,(v,\ h,\ \{h'\!:H \mid E.v.h' = E.v.h\})\,\}$	
Assign to visible	$v := E.v.h$	$\{\,(E.v.h,\ h,\ \{h'\!:H \mid E.v.h' = E.v.h\})\,\}$	$\star$
Assign to hidden	$h := E.v.h$	$\{\,(v,\ E.v.h,\ \{h'\!:H \cdot E.v.h'\})\,\}$	$\star$
Choose visible	$v :\in S.v.h$	$\{v'\!:S.v.h \cdot\ (v',\ h,\ \{h'\!:H \mid v' \in S.v.h'\})\,)\,\}$	$\star$
Choose hidden	$h :\in S.v.h$	$\{h'\!:S.v.h \cdot\ (v,\ h',\ \{h'\!:H; h''\!:S.v.h' \cdot h''\})\,)\,\}$	$\star$
Execute atomically	$\langle\!\langle P \rangle\!\rangle$	$\mathsf{addShadow}.(\text{``classical semantics of } P\text{''})$	
Sequential composition	$P_1 ; P_2$	$\mathsf{lift}.[\![P_2]\!].([\![P_1]\!].v.h.H)$	
Demonic choice	$P_1 \sqcap P_2$	$[\![P_1]\!].v.h.H \cup [\![P_2]\!].v.h.H$	
Conditional	**if** $E.v.h$ **then** P_t **else** P_f **fi**	$[\![P_t]\!].v.h.\{h'\!:H \mid E.v.h' = \textbf{true}\}$	
		$\lhd E.v.h \rhd$	
		$[\![P_f]\!].v.h.\{h'\!:H \mid E.v.h' = \textbf{false}\}$	

The syntactically atomic commands A marked $\star$ have the property that $A = \langle\!\langle A \rangle\!\rangle$. This is deliberate: syntactic atoms execute atomically. The function $\mathsf{lift}.[\![P_2]\!]$ applies $[\![P_2]\!]$ to all triples in its set-valued argument, un-Currying each time, and then takes the union of all results.

The extension to many variables $v_1, v_2, \cdots$ and $h_1, h_2, \cdots$, including local declarations, is straightforward [13, 14].

Fig. 1. Semantics of non-looping commands

Definition 1. *Atomic shadow semantics.* Given a classical program $r \colon \mathcal{V} \to \mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathcal{H})$ we define its *shadow enhancement* $\mathsf{addShadow}.r$ of type $\mathcal{V} \to \mathcal{H} \to \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H})$ so that $\mathsf{addShadow}.r.v.h.H \ni (v', h', H')$ just when

(i) we have both $r.v.h \ni (v', h')$ — *classical*

(ii) and $H' = \{h'\!:\mathcal{H} \mid \big(\exists h''\!:H \cdot r.v.h'' \ni (v', h')\big)\}$.   — *shadow*

$\square$

Clause (i) says that the classical projection of $\mathsf{addShadow}.r$'s behaviour is the same as the classical behaviour of just r itself. Clause (ii) says that the final shadow H' contains all those values h' compatible with allowing the original hidden value to range as h'' over the initial shadow H.

2.3 Security-Aware Program Refinement

Equality of programs is a special case of refinement, whence compositionality is a special case of monotonicity: two programs with equal semantics in isolation must remain equal in all contexts. With those ideas in place, we define refinement as follows:

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

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

with $\llbracket\cdot\rrbracket$ as defined in Fig. 1.

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 the existence of a triple (v', h', H'_1), with equal or smaller shadow, produced by P_1 under the same circumstances.   □

3   Programming with Hidden State

What makes security analysis difficult is the seeming incompatibility of both keeping a secret *and* using it in "public computations." In this section we summarise the characteristics of the Shadow semantics that allow us to analyse the extent to which information is revealed at runtime.

Runtime visibility and *in*-visibility. A *visible* variable is one whose runtime value can be "observed" after each (atomic) execution. For example, the resolution of the nondeterministic choice in the program $v{:}\in\{0,1\}$ can be determined simply by reading the final value of the visible variable v. Assignments to *hidden* variables, in contrast, cannot be observed directly. Thus the program $h{:}\in\{0,1\}$ reveals nothing about h at runtime beyond what can be gleaned statically by examining the source code: we deduce that it is either 0 or 1; but we don't know which.

Interaction and information flow. More interesting is when visible and invisible variables interact, for that is where correlations are formed. Direct publication of the hidden state results in a direct correlation, for example $v{:=}h$ effectively announces h's value. Moreover once the information is in the public domain, no amount of track-covering can erase the knowledge. The program $v{:=}h; v{:=}0$ also leaks h, *even though v is overwritten immediately afterwards* — that is because our *attack model* [10] assumes that an observer can see the the results of visible computations after each "atomic step," which is normally defined by sequential composition (but see atomicity below). In addition an observer may make deductions based on his run-time observations and the structure of the program code. Thus in principle attackers have perfect recall [13,14].

This curious interaction of hidden and visible assignments means sequential composition becomes a somewhat strange operator — for instance it no longer satisfies the rule $(v{:=}h; v{:=}0) = v{:=}0$. Luckily these idiosyncracies are limited to visible/hidden interactions, with the classical rules continuing to apply as normal in the cases where the reasoning is entirely between visible variables.

Compositionality and refinement. Two programs are judged to be the same if and only if they are both functionally equivalent *and* have identical "security defences." The latter is crucial to our hierarchical development method, for it implies that one program may be replaced by its equivalent *in any context*,

without fear of unanticipated security flaws. In our examples below we will use not-necessarily-executable programs as specifications to articulate our overall security goals.

When reasoning about programs we are able to assume the normal structural rules, so for example $P \sqcap Q \sqsubseteq P$, and (**if** $E.v.h$ **then** P_t **else** P_f **fi**); $Q =$ **if** $E.v.h$ **then** $P_t; Q$ **else** $P_f; Q$ **fi**. We also use the fact that decreasing visibility is always a secure refinement, *i.e.* $[\![\mathbf{vis}\ x \cdots]\!] \sqsubseteq [\![\mathbf{hid}\ x \cdots]\!]$, where we have used "visibility declarations" (discussed below) to assign the visibility attribute to the variable x.

Atomicity: controlling granularity. Explicit atomicity is necessary for hiding the results of intermediate computations when secrecy demands it. For example the process of encryption typically is achieved as a result of a number of steps, and it is only safe to publish the final result after obliterating the intermediate computations. We use $\langle\!\langle P \rangle\!\rangle$ to mean that the internals of program P are not revealed at runtime — and within those brackets $\langle\!\langle \cdot \rangle\!\rangle$ we can therefore use classical *equality* reasoning. Proper refinement however is not allowed.

That is, within the safety of atomicity brackets, classical equality reasoning is reinstated so that $\langle\!\langle v{:=}h; v{:=}0 \rangle\!\rangle = \langle\!\langle v{:=}0 \rangle\!\rangle$; but we cannot for example reason via refinement that $(h{:=}0 \sqcap h{:=}1) \sqsubseteq h{:=}0$ implies

$$h{:}\in \{0,1\} \quad = \quad \langle\!\langle h{:=}0 \sqcap h{:=}1 \rangle\!\rangle \quad \sqsubseteq \quad \langle\!\langle h{:=}0 \rangle\!\rangle \quad = \quad h{:=}0 \ ,$$

becuase the middle (refinement) step fails.

Removing atomicity brackets is possible only under certain circumstances. The following lemma sets out one such case.

Lemma 1. *atomicity and composition [10].* Given two programs $P_{\{1,2\}}$ over v, h we have $\langle\!\langle P_1; P_2 \rangle\!\rangle = \langle\!\langle P_1 \rangle\!\rangle; \langle\!\langle P_2 \rangle\!\rangle$ 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 is interpreted classically on the left, and as in Fig. 1 on the right. $\qquad\square$

Lem. 1 prevents us from removing the atomicity brackets for $\langle\!\langle v{:=}h; v{:=}0 \rangle\!\rangle$, but allows it for $\langle\!\langle v{:=}\{0,1\}; h{:=}v{\oplus}E \rangle\!\rangle$, for example. In the former case the intermediate value of v (equal to the hidden h) cannot be deduced from its final value (the constant 0); in the latter case, v's final value is the same as its intermediate value, and atomicity offers no further protection.

Before beginning our real case studies, we elaborate on our treatment of multi-agent systems, and encryption.

4   Agents, Views and Proofs

Our cases studies below are all examples of "multi-agent systems" in that they are composed of a number of independent components, which collaborate to achieve an overall goal. When secrecy is an issue, each agent only has a "partial view"

of the system state, and has complementary security goals with respect to the other agents and to the system as a whole. We use the extension of the Shadow semantics introduced elsewhere [10] to express the differing views of the agents in the system. Essentially the simple semantics can reflect a single agent's viewpoint.

Multiple agents, and the attacker's capabilities. Let A be an agent in a multi-agent system; the above simple semantics reflects A's viewpoint, say, by interpreting variables declared to be **vis**$_{list}$ as visible (**vis**) variables if A is in *list* and as hidden (**hid**) variables otherwise. More precisely,

  – **var** means the associated variable's visibility is unknown or irrelevant.
  – **vis** means the associated variable is visible to all agents.
  – **hid** means the associated variable is hidden from all agents.
  – **vis**$_{list}$ means the associated variable is visible to all agents in the (non-empty) list, and is hidden from all others (including third parties).
  – **hid**$_{list}$ means the associated variable is hidden from all agents in the list, and is visible to all others (including third parties).

For example $[\![\mathbf{vis}_A\ a; \mathbf{vis}_B\ b; \mathbf{vis}\ c;\ c{:=}a{\oplus}b]\!]$ from A's viewpoint the specification would be interpreted with a and c visible and b hidden; for B the interpretation hides a instead of b. For a third party X, say, both a, b are hidden but c is still visible. We say that a system is *generally secure* provided that it is *specifically secure* (as determined by the Shadow semantics) from all its viewpoints. For us this means that the proof must be checked for all those viewpoints; happily many of these can be carried out schematically.

   Visibility declarations can be thought of as placing access restrictions on variables; it does not mean that the value of the variables must always remain unknown to agents not on its visibility list: that depends on the code, since *e.g.* hidden h is known to all once the statement $v{:=}h$ has been executed. They do however have an impact on which refinements will be judged ultimately to be valid.

5   The General Encryption Lemma

Our first case study is a small "toolkit" security idiom which occurs in many protocols: it is the splitting into two pieces of some hidden information, with only "one half" of it then subsequently revealed: the key to the protocols is that this does not introduce a security vulnerability. Perhaps the simplest case is

$$[\![\ \mathbf{vis}\ v;\ \mathbf{hid}\ h;\quad h{:}{\in}\{0,1\}; v{:=}E{\oplus}h\ ]\!]\ , \tag{1}$$

where all types are Boolean (equiv. $\{0, 1\}$) and $\oplus$ is exclusive-or. No matter what the visibility characteristics of E might be, the code above reveals nothing (more) about it. In this section, we will discuss a symmetric version of this, and in more general terms than Booleans and exclusive-or.

5.1   The Symmetric Encryption Lemma

With (1) as motivation, we reason about two agents A, B in some context where expression E is meaningful. We take A's point of view, and show as follows that (1) is equivalent to **skip**, and so changes nothing (global) but –more significantly– *reveals* nothing about E:

$$
\begin{array}{lll}
 & [\![\ \mathbf{vis}_A\ a; \mathbf{vis}_B\ b; & (a{\oplus}b){:=}E\ ]\!] & \text{``from (1)''} \\
= & [\![\ \mathbf{vis}_A\ a; \mathbf{vis}_B\ b; & \langle\!\langle (a{\oplus}b){:=}E \rangle\!\rangle\ ]\!] & \text{``statement is atomic already''} \\
= & [\![\ \mathbf{vis}_A\ a; \mathbf{vis}_B\ b; & \langle\!\langle a{:}{\in}\mathcal{E}; b{:=}E{\oplus}a \rangle\!\rangle\ ]\!] & \text{``}\mathcal{E}\text{ is the type of }a,b,E;\text{ see (i) below'' } \heartsuit \\
= & [\![\ \mathbf{vis}_A\ a; \mathbf{vis}_B\ b; & \langle\!\langle a{:}{\in}\mathcal{E} \rangle\!\rangle; \langle\!\langle b{:=}E{\oplus}a \rangle\!\rangle\ ]\!] & \text{``atomicity lemma''} \\
= & [\![\ \mathbf{vis}_A\ a; \mathbf{vis}_B\ b; & a{:}{\in}\mathcal{E}; b{:=}E{\oplus}a\ ]\!] & \text{``statements are atomic anyway''} \\
= & [\![\ \mathbf{vis}_A\ a; a{:}{\in}\mathcal{E}; & [\![\ \mathbf{vis}_B\ b; \quad b{:=}E{\oplus}a\ ]\!]\ ]\!] & \text{``}b\text{ is not free in }\mathcal{E};\text{ see (ii) below'' } \heartsuit \\
= & [\![\ \mathbf{vis}_A\ a; & a{:}{\in}\mathcal{E}; \mathbf{skip}\ ]\!] & \text{``}b\text{ is hidden from }A\text{'' } \flat \\
= & [\![\ \mathbf{vis}_A\ a; & a{:}{\in}\mathcal{E}\ ]\!] & \text{``}\mathbf{skip}\text{''} \\
= & \mathbf{skip}\ . & & \text{``}a\text{ is a local visible''}
\end{array}
$$

The proof for B's point of view is symmetric.[1] The crucial features $\heartsuit$ of the derivation are these:

(i) The correctness of this step has both classical and security aspects. The classical aspect is simply that we must have $(E{\oplus}a) \oplus a = E$.

   The security aspect is that, within atomicity brackets $\langle\!\langle \cdot \rangle\!\rangle$, only *equality* reasoning is allowed; proper refinement is not, and this concerns the introduction of the type-set $\mathcal{E}$. That set must capture *precisely* the possible values of a that could result from the (previous) statement $(a{\oplus}b){:=}E$, no more and no less — otherwise it's not an equality. Putting that in words we would say "For all values of E and all $a{\in}\mathcal{E}$ there must be some $b{\in}\mathcal{E}$ so that $a = E{\oplus}b$, and furthermore $\mathcal{E}$ contains all the values that a could have."

(ii) In this step we moved $a{:}{\in}\mathcal{E}$ out of the scope of b. This is possible only because in choosing $\mathcal{E}$ from which to pick a we were able to ignore b, i.e. that the choice-range for a is independent of b (and E).

In the next section we illustrate the above Boolean-based encryption with a simple scheme for secure messaging.

6   Secure Messaging in an Untrusted Medium

Sender S is eager to tell R a secret but, as they live far apart, he cannot whisper it in his ear. Instead he sends it with messengers X, Y even though he does not trust either one separately not to read the message he is delivering. First S splits s into two "shares" s_x and s_y in such as way that their exclusive-or is equal to s, *i.e.* so that $s_x \oplus s_y = s$. He gives s_x to X and s_y to Y with the instruction to

---

[1] The $\flat$ is referred to in §8.2.

$\mathbf{vis}_S\ s; \mathbf{vis}_R\ r;$
$\mathbf{vis}_{SX}\ s_x; \mathbf{vis}_{SY}\ s_y;$
$\mathbf{vis}_X\ x; \mathbf{vis}_Y\ y;$
$\mathbf{vis}_{RX}\ r_x; \mathbf{vis}_{RY}\ r_y;$

$(s_x \oplus s_y) := s;$	$\Leftarrow$ S splits the message in two.
$(s_x \oplus s_y) := s;$	$\Leftarrow$ S splits the message in two.
$x, y := s_x, s_y;$	$\Leftarrow$ Messages sent from S to X and to Y separately.
$r_x, r_y := x, y;$	$\Leftarrow$ Messages sent from X and Y to R.
$r := r_x \oplus r_y$.	$\Leftarrow$ R recombines the two halves.

We write $(s_x \oplus s_y) := s$ for the (atomic) choice over all possibilities of splitting the message s, equivalent to the specification statement $s_x, s_y : [s_x \oplus s_y = s]$ and interpreted atomically[12].

Fig. 2. Abstract messaging with non-colluding messengers

deliver their messages to R. Once R receives the two halves he can reassemble them at his leisure to reveal s. The code, including its visibility declarations, is set out at Fig. 2.

Clearly this scheme transfers s to R; as for security, it seems intuitive that if s is split so that neither X nor Y learns its contents, then the message passing reveals no more. Our goal in this section is to check formally that the intuition is sound. We begin with an "obviously correct" specification, namely an atomic transaction between R and S:

$$\mathbf{vis}_S\ s;\ \ \mathbf{vis}_R\ r;\ \ r := s\ , \tag{2}$$

which is "as if" the message were indeed whispered; but that is not directly executable because r and s are local only to R and S respectively. Nevertheless it precisely sets out the limited circulation of s — X and Y are excluded from the the visibility lists, and therefore neither X nor Y can know s. The next step is to ensure that the restricted circulation is maintained in spite of introducing untrustworthy agents.

Following the refinement tradition, we gradually introduce the message-passing infrastructure, making sure as we do so that neither by publication nor by careless program structure can X or Y glean anything about s. As we introduce detail it becomes important to identify what is already known, and by whom — we use a new technique of "visibility annotations"[2] to formalise exactly that.

Definition 3. *The statement* $\mathbf{reveal}_{list}\ E$ *is just* $\mathbf{reveal}\ E$ *if the viewpoint is in agent-list list, and is* $\mathbf{skip}$ *otherwise.*

Definition 4. *We say that an expression E is effectively list-visible at a point in a program just when putting a statement* $\mathbf{reveal}_{list}\ E$ *there would not alter the program's meaning.*

---

[2] Thanks to Carroll Morgan for suggesting visibility annotations.

In our case we need to know at what point in the transaction we can assume who knows what; in practice to determine the visibility of an expression we use the visibility declarations as well as other information which has already been revealed. Thus an expression is said to be effectively visible (at a point) just when its value is determined by variables visible (at that same point) and any other expressions that are effectively visible at that point.

Now we begin with the simple specification (2), embellishing it until we reach the message-passing scheme at Fig. 2. At each stage we sill use visibility annotations, visibility declarations or simple program algebra to justify the equality between programs.

Step 1: Visibility annotations. We start by analysing the visibility of s both before and after the assignment in (2); we use the visibility annotations. First, it is clear that r is effectively S-visible after the statement, and that s is effectively R-visible both before and after. Obvious or not, we check this as follows: we use Def. 4 to put **reveal**$_S$ r and **reveal**$_R$ s before and after the assignment.

First, we see that r is effectively S-visible after the assignment:

$$
\begin{aligned}
&\quad r{:=}\,s; \textbf{reveal}_S\ r \\
&=\quad r{:=}\,s; \textbf{reveal}_S\ s \qquad\qquad\qquad\qquad\qquad \text{`` } r = s \text{ at that point''} \\
&=\quad r{:=}\,s\ . \qquad\qquad\qquad\qquad \text{``}s \text{ is } S\text{-visible by declaration''}
\end{aligned}
$$

Similarly s R-visible after the assignment:

$$
\begin{aligned}
&\quad r{:=}\,s; \textbf{reveal}_R\ s \\
&=\quad r{:=}\,s; \textbf{reveal}_R\ r \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{``as above''} \\
&=\quad r{:=}\,s\ .
\end{aligned}
$$

And finally s is r-visible *before* the assignment:

$$
\begin{aligned}
&\quad (\textbf{reveal}_R\ s); r{:=}\,s \\
&=\quad r{:=}\,s; \textbf{reveal}_R\ r \qquad\qquad\qquad\qquad\qquad \text{`` } s \text{ is unchanged''} \\
&=\quad r{:=}\,s\ . \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{``as above''}
\end{aligned}
$$

The last one is interesting, since *operationally* one would be inclined to say that s is not R-visible before the statement, since we "can't yet know s" before that assignment has occurred. But here (yet again) is where a logical view helps us to avoid confusions that operational reasoning can cause.

Referring to the "attack model" sketched above, we'd say under an attack from R we'd have that s is visible before the statement $r{:=}\,s$ just when R really can see it. But he can't see it, can he...? Nevertheless he can reason as if he could: whatever reasoning he wanted to do with s at that point he simply defers, first allowing the program to run one further step. Then s really is visible (by inference, since it's now sitting in r), and then R can go back and continue the reasoning based on s that he had put on hold.

Step 2: Splitting the message. Now we have learned about R and S's viewpoints, we can start adding details of the message-passing. We use encryption to split s, but we need to show that still only R and S learn s. What we need to show is that

$$(2) \quad = \quad [\![\mathbf{vis}_{RXS} \ s_x; \mathbf{vis}_{RYS} \ s_y; \quad (s_x \oplus s_y) := s]\!]; r := s \ ,$$

where we have used the specification statement to make mutually secret shares s_x and s_y.

Here although the encryption guarantees that neither X nor Y learn anything, to ensure equality with the specification, we need to check that the security refinement holds from all points of view, and that includes R and S. The problematic case is R, because on the right since R can see s_x and s_y, he would learn the secret before the assignment to his variable r. Although we don't really "care" about that (after all, he is the intended recipient of s) in our formal proof we are *made to care*, and rightly so — information can be unintentionally leaked and if an agent learns something "early" then he becomes a security risk when he was not intended to be. In this case early knowledge is not a problem, as our visibility analysis above has already checked for us.

1. From S's point of view, everything is visible in the new block (no security problems), and the (generalised) assignment is to new local variables (no classical problems).
2. From X (Y)'s point of view, it's an instance of the encryption lemma.
3. From R's point of view (the only interesting one), we would formerly have been stuck because s_x, s_y are both visible to R but s is hidden from R. But now we can see that although s is hidden from R by declaration, nevertheless it is R-visible (from Step 1 above) and so this case reduces to (1).

Step 3: Delivering the messages. The next step introduces the messengers X and Y, who now carry their halves in variables x and y and give them to R.

$$
\begin{array}{lll}
& \mathbf{vis}_R \ r; \ \mathbf{vis}_S \ s; \ r := s & \\
= & (s_x \oplus s_y) := s; r := s & \text{``}\mathbf{vis}_{RXS} \ s_x; \mathbf{vis}_{RYS} \ s_y\text{''} \\
= & (s_x \oplus s_y) := s; & \text{``}\mathbf{vis}_{RX} \ x; \mathbf{vis}_{RY} \ y\text{''} \\
& x, y := s_x, s_y; & \\
& r := s & \\
= & (s_x \oplus s_y) := s; & \text{``}\mathbf{vis}_R \ r_x, r_y\text{''} \\
& x, y := s_x, s_y; & \\
& r_x, r_y := x, y; & \\
& r := s & \\
= & (s_x \oplus s_y) := s; & \text{``program algebra''} \\
& x, y := s_x, s_y; & \\
& r_x, r_y := x, y; & \\
& r := r_x \oplus r_y \ . & \\
\end{array}
$$

For the final step from here to Fig. 2, we use the general refinement rule for reducing visibilities, replacing $\mathbf{vis}_{RXS} \ s_x; \mathbf{vis}_{RYS} \ s_y$ by $\mathbf{vis}_X \ s_x; \mathbf{vis}_Y \ s_y$.

7   Secure Remote Computations

We now take another step towards our principal case study. Private Information Retrieval is very similar to secure message-passing as above, but includes structured set-valued messages, and remote computation. We begin by working towards a more general instance of the encryption lemma.

7.1   The Exclusive-or Algebra of Subsets

We take as our type $\mathcal{E}$ the powerset $\mathbb{P}[0..N)$ of the natural numbers below N, which we will abbreviate $\mathbb{P}N$. For our operation $\oplus$ we take the *symmetric set-difference*, which we will write Δ so that for $N_{0,1}\in\mathbb{P}N$ we have $N_0\Delta N_1 = N_0-N_1 \cup N_1-N_0$[3]. As payoff for our generality above, we have immediately for $E\in\mathbb{P}N$ the equality

$$\llbracket\ \mathbf{vis}_A\ a\colon\mathbb{P}N;\mathbf{vis}_B\ b\colon\mathbb{P}N;\ \ (a\Delta b):=E\ \rrbracket\quad=\quad\mathbf{skip}\ .\tag{3}$$

It's just the encryption lemma for subsets. Here's how we can use it.

7.2   Secure Use of a Remote Super-Computer

Suppose some user-agent U wants to compute $y:=F.x$ with $\mathbf{vis}_A\ x,y$, so that the variables involved are visible only to him. (We do not specify the types of x,y at this stage.) The function F is public; but unfortunately it is so complicated that A does not have the resources to compute it. His first thought is to ship y off to a super-computer -agent A who will compute it for him, thus he hopes for

$$y:=F.x\quad\sqsubseteq\quad\llbracket\ \mathbf{vis}_A\ a,a';\quad a:=x;\ a':=F.a;\ y:=a'\ \rrbracket\ ,$$

in which $a:=x$ sends the argument from U to A, and $y:=a'$ returns the result. The computation $a':=F.a$ is then carried out entirely by A.

Although this is a classical refinement (obviously), it is not a secure one: the problem is that A learns the values of x,y, and they are supposed to be private to U.

Now let us suppose that the function F distributes $\oplus$ (over the types of x,y), that is that $F.(x_0\oplus x_1) = F.x_0\oplus F.x_1$. Moreover we assume that U values his privacy so much that he is prepared to pay for *two* super-computer runs, the second one's being run by Agent B. He now proposes the refinement

$$y:=F.x\quad\sqsubseteq\quad\llbracket\ \mathbf{vis}_A\ a;\mathbf{vis}_B\ b;\quad(a\oplus b):=x;\ y:=F.a\oplus F.b\ \rrbracket$$

in which, to reduce clutter, we have suppressed the assignments (like $a:=x$ above) that are simply to do with passing values from one agent to another.

The classical correctness of this second refinement-proposal depends on the $\oplus$-distributivity of F, which we have assumed; but what about its security correctness? That follows from the Encryption Lemma, since we can derive

---

[3] This operator Δ really is just exclusive-or $\oplus$ in different clothes: regard the sets as characteristic functions, and then apply the ordinary Boolean exclusive-or pointwise to those functions.

$\quad$ $y := F.x$

$=$ $\quad$ $[\![$ $\mathbf{vis}_A\ a; \mathbf{vis}_B\ b; \quad (a{\oplus}b) := x\ ]\!]$;$\qquad\qquad$ "Encryption Lemma"
$\quad\quad$ $y := F.x$

$=$ $\quad$ $[\![$ $\mathbf{vis}_A\ a; \mathbf{vis}_B\ b$;$\qquad\qquad\qquad\qquad\qquad$ "scope and context"
$\quad\quad$ $(a{\oplus}b) := x$;
$\quad\quad$ $y := F.(a{\oplus}b)$
$\quad$ $]\!]$

$=$ $\quad$ $[\![$ $\mathbf{vis}_A\ a; \mathbf{vis}_B\ b$;$\qquad\qquad\qquad\qquad$ "$\oplus$-distributivity of F"
$\quad\quad$ $(a{\oplus}b) := x$;
$\quad\quad$ $y := F.a \oplus F.b$
$\quad$ $]\!]$

This solves U's privacy problems — though he does have to pay for two runs of the function F.

7.3   Explicit Message-Passing

Naturally the two statements $(a{\oplus}b) := x$ and $y := F.a{\oplus}F.b$ above must themselves be implemented via explicit message passing. For the first we argue by analogy with the two-messengers approach of §6, as follows:

$\quad$ $(a{\oplus}b) := x$

$=$ $\quad$ $[\![$ $\mathbf{vis}_{UA}\ x_A; \mathbf{vis}_{UB}\ x_B$;$\qquad$ "Encryption Lemma, scoping and context"
$\quad\quad$ $(x_A{\oplus}x_B) := x$;
$\quad\quad$ $a, b := x_a, x_b$
$\quad$ $]\!]$

$\sqsubseteq$ $\quad$ $[\![$ $\mathbf{vis}_U\ x_A, x_B$;$\qquad\qquad\qquad\qquad\qquad$ "reduce visibility"
$\quad\quad$ $(x_A{\oplus}x_B) := x$;
$\quad\quad$ $a, b := x_a, x_b$
$\quad$ $]\!]$.

For the second, similar reasoning (which we elide) gives

$\quad$ $y := F.a \oplus F.b$

$\sqsubseteq$ $\quad$ $[\![$ $\mathbf{vis}_U\ y_A, y_B$;$\qquad\qquad\qquad\qquad\qquad\qquad$ "as above"
$\quad\quad$ $\mathbf{vis}_A\ z_A; \mathbf{vis}_B\ z_B$;
$\quad\quad$ $z_A, z_B := F.a, F.b$;
$\quad\quad$ $y_A, y_B := z_A, z_B; \quad y := y_A \oplus y_B$
$\quad$ $]\!]$.

$\quad$ Put together with the refinement of the previous section (and exploiting monotonicity), we have the overall refinement shown in Fig. 3.

$y:= F.x$

$\sqsubseteq$ $[\![$ $\mathbf{vis}_U\ x_A, x_B, y_A, y_B;$ "composition of the above"
         $\mathbf{vis}_A\ z_A; \mathbf{vis}_B\ z_B;$

         $(x_A \oplus x_B) := x;$ $\Leftarrow$ Split x into two shares.
         $a, b := x_a, x_b;$ $\Leftarrow$ Send them to Agents A, B separately.
         $z_A, z_B := F.a, F.b;$ $\Leftarrow$ Agents A, B compute F on their respective arguments.
         $y_A, y_B := z_A, z_B;$ $\Leftarrow$ The results are sent back to U.
         $y := y_A \oplus y_B$ $\Leftarrow$ Agent U combines the result shares to get the answer.
      $]\!]$

Fig. 3. Using two remote super computers to calculate an expensive function privately

8   Private Information Retrieval

This is our principal case study. In publicly accessible databases security is not about protecting data, but rather about protecting users –this can be an issue if the data concerns medical or share price information– because the user may want his request to be confidential. Hence the objective of *private information retrieval schemes* (*PIR*) is that the *requests* themselves should remain anonymous.

It has been shown that when the data is stored on a single server (a "single-server model") the only way to achieve the anonymity of requests is for the user to download the entire database for local (and therefore private) perusal [4], but the cost of this confidentiality is extremely poor performance. Current research on *PIR* aims to minimise communication complexity, and in this section we study a scheme introduced by Chor et al. [4].

The idea is to uses some number $d \geq 2$ of copies of the database servers. As in the message-passing example above the user splits the request into d shares, sending each share to each server. The trick is to make sure that the shares (a) reveal no information about the actual request (to either server or a third party), and (b) can nevertheless be reconstructed by the user to reveal his actual request.

Chor explains that the performance reduction only emerges when in fact $d > 2$, but that the security aspects are well illustrated (but more easily!) for $d=2$. Following his advice we begin our formalisation for $d=2$, and in any case study only the security aspects in detail. We assume a database D of N (bit-sized) records addressable with an index $1 \leq i \leq N$; we use U for the user, and A, B for the two servers, each of which host (identical) copies D_A and D_B of D. Chor's informal description of the two-server model is as follows:

> Let U's secret request be some $1 \leq c \leq N$, and he wants to know $D.c$ (equivalently $D_A.c$ or $D_B.c$). He chooses randomly a subset $\mathcal{S} \in \mathbb{P}N$, and then sends (all of) $\mathcal{S}$ to A and $\mathcal{S} \odot c$ to B, where
>
> $$\mathcal{S} \odot c := \mathbf{if}\ (c \in S)\ \mathbf{then}\ \mathcal{S} \backslash c\ \mathbf{else}\ \mathcal{S} \cup \{c\}\ \mathbf{fi}.$$

Next A sends to U the result $y_A := (\oplus_{i \in \mathcal{S}} D_A.i)$, and B similarly sends $y_B := (\oplus_{i \in \mathcal{S} \odot z} D_B.i)$; finally U decrypts the two replies by computing $y_A \oplus y_B$.

The functional correctness of this scheme can be seen easily because of the definition of $\odot$. Note that $\mathcal{S} \odot c$ simply includes c if $c \notin \mathcal{S}$, or it removes it if it is already in $\mathcal{S}$. That means that c occurs in exactly one of $\mathcal{S}$ or $\mathcal{S} \odot c$, but all the other items in $\mathcal{S}$ appear in both subsets. Thus in the final computation of the exclusive or, all the terms $D.i$ cancel out except for $D.c$ and hence $y_A \oplus y_B = D.c$ as required.

The security correctness is slightly more involved, but still intuitive. Since the set $\mathcal{S}$ is chosen at random from all possible subsets of $\{1 \ldots N\}$ when a server receives the subset it does not know whether the real query c is contained in the subset or not.[4] Moreover $\mathcal{S} \odot c$ also appears equally likely amongst all subsets, therefore provided A and B do not collude, they are individually none the wiser as to the actual request.

8.1   Solving the *PIR* Problem with Algebra

Using our results from §5, we can legitimise Chor's approach easily.

First, note that Chor's $\mathcal{S} \odot c$ is just $\mathcal{S} \oplus \{c\}$ in our terms. This establishes the connection with exclusive-or. Second, Chor's operation $(\oplus_{i \in \mathcal{S}} D_A.i)$ (and equivalently $(\oplus_{i \in \mathcal{S}} D_B.i)$ is our function F — and it distributes $\oplus$. This means the refinement of Fig. 3 applies immediately, once we notice that $D.c = D_A.c = D_B.c = F.\{c\}$. Thus we obtain by instantiation the refinement of Fig. 4, in which our initial split $(x_A \Delta x_B) := \{c\}$ is equivalent to Chor's $x_A :\in \mathbb{P} N; x_B := x_A \odot c$.

8.2   Collusion and Visibility Declarations

The above derivation explicitly separates the U/A and U/B correspondence by enforced by the visibility declarations $\mathbf{vis}_A$ and $\mathbf{vis}_B$; for Chor that separation is articulated by the "non-collusion" assumption, and theorems there depend upon it. Here there is a similar dependency, and indeed the validity of refinement depends upon it.

To investigate what would happen if A and B do collude, we rename all the A/B variables to belong to a single server C variable, and attempt the same derivation. [5]This means that all $\mathbf{vis}_A; \mathbf{vis}_B$ declarations become $\mathbf{vis}_C$ — then a careful review of the proofs shows that the original encryption §5, on which the whole security is built fails at the step labelled $\flat$. In this case, the relabelling would make both a, b variables $\mathbf{vis}_C$, so that the comment "*b is hidden* ..." is invalid, preventing the replacement of the assignment to b with $\mathbf{skip}$.

---

[4] Of course if the two servers share their partial information by colluding then the value z is revealed. We discuss collusion later.

[5] We do this since we do not assume anything about the nature of the collusion, except that the servers are able to share all correspondence.

$u := D.c$

$\sqsubseteq$ $[\![$ $\mathbf{vis}_U$ $x_A, x_B \colon \mathbb{P}N; y_A, y_B \colon \mathsf{Bool};$ "instantiating Fig. 3"
          $\mathbf{vis}_A$ $z_A \colon \mathsf{Bool}; \mathbf{vis}_B$ $z_B \colon \mathsf{Bool};$

   $(x_A \Delta x_B) := \{c\};$ $\Leftarrow$ Split c into two "subset" shares.
   $a, b := x_a, x_b;$ $\Leftarrow$ Send to the servers separately.
   $z_A := (\oplus_{i \in a} D_A.i);$ $\Leftarrow$ Each computes the $\oplus$ of its shares.
   $z_B := (\oplus_{i \in b} D_B.i);$ $\Leftarrow \cdots$
   $y_A, y_B := z_A, z_B;$ $\Leftarrow$ Each sends the result back to the requester.
   $u := y_A \oplus y_B$ $\Leftarrow$ The results are $\oplus$-ed together.
 $]\!]$

Fig. 4. Using two remote databases to perform a lookup privately

8.3   Efficient Perfect Information Retrieval

The solution presented in §8 actually does not reduce the overhead on the network at all — in fact it is the same as the single-server solution where the whole database must be sent to U.

The full solution, combining privacy and a reduction in average network traffic — from $O(N)$ to $O(\sqrt{N})$ (for example) — needs strictly more than two servers, and a structured addressing scheme. Again each server is sent an apparently random set of requests for which it must compute the $\oplus$ of the results, and return to the user, who can then reassemble to uncover the request. Although the addressing scheme is somewhat detailed, the principles for correctness, and the machinery for proof remain the same, namely generalised encryption §5.1 and the exclusive-or algebra §7.1.

9   Conclusions and Future Work

We have shown how to validate a well known protocol for Perfect Information Retrieval using a novel refinement-style development. Our approach emphasises a hierarchical analysis which refinement supports, allowing us to use specifications of sub-protocols in our proofs. Critically the proofs are carried out ultimately at the level of source code, thus legitimising noninterference security goals at that level of detail.

The relationship to other formal semantics of non-intereference has been summarised in detail elsewhere [13,14]; it is comparable to Leino [8] and Sabelfeld [15], but differs in details; and it shares the goals of the pioneering work of Mantel [9] and Engelhardt [5].

Our work sits between two communities. On the one hand there are those who reason about code at the source level, and in some cases build (semi-)automated tools to help them do so. Reasoning that way about security however is quite rare; and this community generally does not study the advanced theoretical models of semantics for security for their own sake.

On the other hand, there are those who study or create the mathematics upon which cryptography and secrecy depend. But it is rare to find there a serious interest *as well* in the problems of transferring their insights to the source-code level[6].

We try to place our contribution in between the two groups, drawing inspiration from the concerns of both and hoping in return to contribute something towards bridging the gap.

Thus although there are many ingenious protocols involving secret information, there is as yet limited support for their code-level justification: most new algorithmic/theoretical insights are presented as a mixture of pseudo-code and English (or other natural language). Our work can be seen as an early step towards bridging the cryptographic/software gap.

Future work on this topic will be to develop a "probabilistic Shadow" to enable stronger cryptographic guarantees, quantitative rather than only qualitative, to be faithfully transferred to source-level computer code.

References

1. Abadi, M., Rogoway, P.: Reconciling two views of crytography (the computational soundness of formal encrytion) . In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872, pp. 3–22. Springer, Heidelberg (2000)
2. Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Back, R.-J.R.: Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam (1980)
4. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. J. ACM 45(6), 965–982 (1999)
5. Engelhardt, K., van der Meyden, R., Moses, Y.: A refinement theory that supports reasoning about knowledge and time. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 125–141. Springer, Heidelberg (2001)
6. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proc IEEE Symp on Security and Privacy, pp. 75–86 (1984)
7. Jacob, J.: Security specifications. In: IEEE Symposium on Security and Privacy, pp. 14–23 (1988)
8. Leino, K.R.M., Joshi, R.: A semantic approach to secure information flow. Science of Computer Programming 37(1–3), 113–138 (2000)
9. Mantel, H.: Preserving information flow properties under refinement. In: Proc IEEE Symp. Security and Privacy, pp. 78–91 (2001)
10. McIver, A.K., Morgan, C.C.: Sums and lovers: Case studies in security, compositionality and refinement. Submitted to Formal Methods 2009 (2009)

---

[6] How many serious programmers do not understand the simple theory of lists underlying *Mergesort*, say? Not many. How many serious implementations of Mergesort contain at least one bug? Probably quite a lot: because of aliasing, source-level reasoning over linked lists is difficult. There really is a gap.

   In the area of unifying the cryptological and formal methods communities there is some work, notably Abadi and Rogoway [1].

11. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Tech. Mono Comp. Sci. Springer, New York (2005)
12. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall, Englewood Cliffs (1994), `web.comlab.ox.ac.uk/oucl/publications/books/PfS/`
13. Morgan, C.C.: The Shadow Knows: Refinement of ignorance in sequential programs. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 359–378. Springer, Heidelberg (2006); Treats Dining Cryptographers
14. Morgan, C.C.: The Shadow Knows: Refinement of ignorance in sequential programs. Science of Computer Programming 74(8) (2009); Treats Oblivious Transfer
15. Sabelfeld, A., Sands, D.: A PER model of secure information flow. Higher-Order and Symbolic Computation 14(1), 59–91 (2001)