DOI 10.1007/s00165-010-0164-1
BCS © 2010
Formal Aspects of Computing (2011) 23: 711–737

Formal Aspects of Computing

Compositional refinement in agent-based security protocols

A. K. McIver[1] and C. C. Morgan[2]

[1]Department of Computer Science, Macquarie University, Sydney, NSW 2109, Australia.
[2]School of Computational Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia.
E-mail: carrollm@cse.unsw.edu.au

Abstract. A truly secure protocol is one which *never* violates its security requirements, no matter how bizarre the circumstances, provided those circumstances are within its terms of reference. Such cast-iron guarantees, as far as they are possible, require formal, rigorous techniques: proof or model-checking. Informally, they are difficult or impossible to achieve. Our rigorous technique is *refinement*, until recently not much applied to security. We argue its benefits by using refinement-based program algebra to develop several security case studies. That is one of our contributions here. The soundness of the technique follows from its compositional semantics, one which we defined (elsewhere) to support a specialisation of standard refinement by enriching standard semantics with information that tracks correlations between hidden state and visible behaviour. A further contribution is to extend the basic theory of secure refinement (Morgan in Mathematics of program construction, Springer, Berlin, vol. 4014, pp. 359–378, 2006) with special features required by our case studies, namely agent-based systems with complementary security requirements, and looping programs.

Keywords: Refinement of security; Formalised secrecy; Hierarchical security reasoning; Compositional semantics

1. Introduction

In engineering, one of the most challenging issues for complex software systems is to ensure that any security requirements are faithfully implemented, even if those requirements are restricted only to a specific subset of the possible threats. In this paper we demonstrate that abstract modelling and refinement can be used for doing that, as a basis for a formal analysis of noninterference security properties. *Noninterference* security addresses the extent to which observable program behaviour reveals information about the parts of the program's state intended to be kept secret. If we partition the state into a "visible" part v and a "hidden" part h, we can observe the interference between h and v as programs execute; interpreting any correlations within a security context gives us the basis for a formalisation of noninterference. For example, the program "$v = 0$" reveals nothing about h, since its result is independent of h's value; compare this to $v := h \bmod 2$, which effectively publishes h's low order bit.

Correspondence and offprint requests to: A. K. McIver, Department of Computer Science, Macquarie University, Sydney, NSW 2109, Australia. e-mail: annabelle.mciver@mq.edu.au

As explained in our earlier work [Mor06], in formulating our approach we adopt the rationale underlying decades of experience in developing large systems (and also underlying successful software-design tools such as, e.g. Rodin and its precursors Atelier-B and the B-Toolkit http://www.deploy-project.eu). It is that a thorough understanding of the principal mechanisms intended to equip a system with a particular requirement will greatly increase the likelihood of its correct design and ultimate deployment. Thus access to *simple* abstract models to describe security defences will provide the designer with a comprehensive impression of how the resulting system might be attacked or abused during operation.

The soundness of our method relies on "security refinement," an extension of (classical) refinement that preserves noninterference properties (as well as classical, functional ones), and that gives compositionality and hierarchical proof an emphasis unusual for security-protocol development. Those features are emphasised because they are essential for scale-up and deployment into arbitrary contexts: in security protocols, the influence of the deployment environment can be particularly subtle.

In relation to other approaches, such as model checking, ours is dual. We begin with a specification rather than an implementation, one so simple that its security and functional properties are self-evident—or are at least small enough to be subjected to rigorous algorithmic checking [RSG+00]. Then secure refinement ensures that noninterference-style flaws in the implementation code, no matter how many refinement steps are taken to reach it, must have already been present in that specification. Because the code of course is probably too large and complicated to understand directly, that property is especially beneficial.

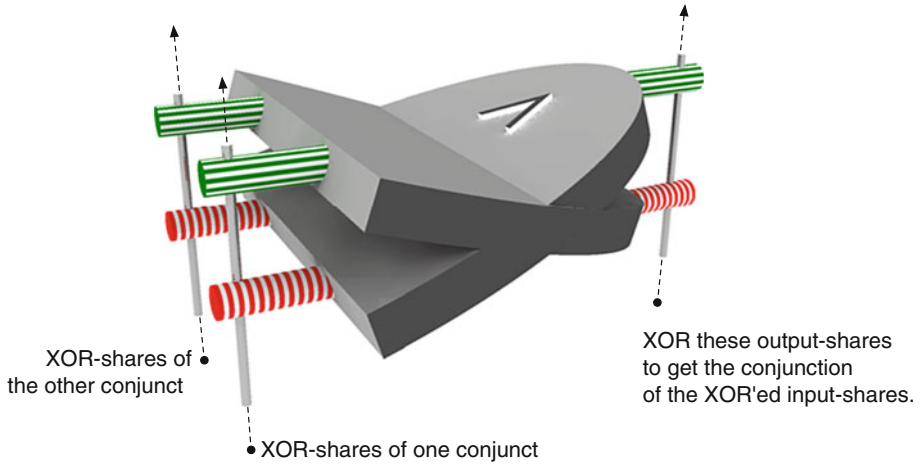
This paper is an extension of a conference paper first published in the proceedings of Formal Methods 2009 [MM09], but is significantly updated with new material. In particular we further develop our secure-refinement paradigm [Mor06, Mor09], extending the semantics and compositional reasoning to deal with loops (Sects. 3.2, 4.6) and including a thorough treatment of "atomicity" and an exploration of general agent-based programs (Sect. 5). The details—previously only sketched—are now given in their entirety, including full formal proofs of a selection of proof rules Sects. 5.2 and 5.3, with others required for our illustrative case studies summarised for convenience in Sects. 5.4 and 5.5. Our series of case studies illustrating the techniques now includes an additional case Sect. 10 that complements the others by illustrating a refinement *failure*. Taken together they achieve a development of a protocol for Yao's millionaires' Problem [Yao82], making a good case for compositionality's effectiveness in the secure-refinement paradigm [Mor06, Mor09], with the size and complexity of security applications that can be verified greatly extended. The millionaires' Problem is such an example because it includes four (sub-)protocols nested like dolls within it: our paradigm allows them to be treated separately, so that each can be understood in isolation. That contrasts with the millionaires' code "flattened" in Fig. 4 to the second-from-bottom level of abstraction: at face value it is impenetrable.

In Sect. 3 we set out the semantics for secure refinement; and in Sect. 7 we begin our series of case studies, in increasing order of complexity; but before any of that, in Sect. 2, we introduce multi-party computations. In the conclusions Sect. 13 we set out our *strategic goals* for the whole approach.

Notation 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 $(\mathsf{Q}x{:}\ T|R{\cdot}E)$ 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 $\{x, y{:}\,\mathbb{N} \mid y = x^2 \cdot z + y\}$ is the set of numbers $z, z+1, z+4, \ldots$ that exceed z by a perfect square exactly. When R is **true** we omit it, so that the quantification $\big(\forall\, x{:}\,\mathbb{N} \cdot x \geq 2\big)$ is the formula stating that all natural numbers are at least 2 (which is false). Finally when E is missing in a set comprehension, it is taken to be the tuple formed by the bound variables.

2. Secure multi-party computation: an overview

In *multi-party computations* (MPC's) separate agents hold their own shares of a collective computation. During the computation agents exchange those shares rather than the actual values, and only at the end are the shares combined to reveal the true value overall; the computation is *secure* if no information is released until that point. In Fig. 1 for example we illustrate an AND-gate shared between two agents A, B: agent A sees only his own shares directly, and during the computation (as we describe below in Sect. 9.2) he comes to see only one half of agent B's shares, but not the other.



Agent A sees the upper shares (horizontally striped), the two inputs and one output; B sees the lower shares (vertically striped). The upper/lower exclusive-or of the two outputs is the conjunction of the left- and right inputs' separate upper/lower xor's.

Fig. 1. $\oplus$-shared conjunction

More generally, we specify a typical two-party MPC as

$$\begin{array}{l}\mathbf{vis}_A\ a;\ \mathbf{vis}_B\ b;\ \mathbf{vis}\ x;\\ x := a \otimes b,\end{array} \tag{1}$$

in which two agents A and B, with their respective variables a and b visible only to them separately, *somehow collaboratively calculate* the result $a \otimes b$ and publish it in the variable x; but they reveal nothing (more) about a, b in the process, either to each other or to a third party. Our declaration $\mathbf{vis}_A\ a$ means that only A can observe the value of a (similarly for B); and the undecorated **vis** means x is observable to all, in this case both A, B and third parties. For example, if $\otimes$ were conjunction then (1) specifies that A (knowing a and seeing x) learns b when a holds, but not otherwise; and a third party learns the exact values a, b only when they are both true. Although the assignment (1) cannot be executed directly when A and B are physically distributed, nevertheless the security and functionality properties it *specifies* are self-evident once $\otimes$ is fixed. But the "somehow collaboratively calculate" above is a matter of *implementing* the specification, using local variables of limited visibility and exchange of messages between the agents. We will see much of that in Sect. 8; and it is not self-evident at all.

An *unsatisfactory* implementation of (1) might involve a real-time *trusted third party* (rTTP): both A, B submit their values to an agent C which performs the computation privately and announces only the result. But this agent C is a corruptible bottleneck and, worse, it would learn a, b in the process. The rTTP can be avoided by appealing to the sub-protocol *oblivious transfer* (OT) [Rab81, Riv99] in which a TTP (no "r") participates only off-line and before the computation proper begins: his agent C is not a bottleneck, and it does not learn a or b. Applied to Fig. 1 this scheme determines which one of B's lower, input-shares should be released to A for the computation to proceed correctly.

Our main case study is Yao's millionaires A, B of Sect. 11 who compare their fortunes, the natural numbers a, b, without revealing either: only the Boolean $a < b$ is published. For us it is a showcase exemplar, because it makes our point of layered development so well: it uses the Lovers' II protocol (Sect. 9.2), in turn using Lovers' I (Sect. 9.1), then appealing to OT (Sect. 8), which depends on the encryption lemma (EL) (Sect. 7); moreover our treatment of the main loop (Sect. 11.3, unbounded riches) abstracts from the loop body (Sect. 11.1, the two-bit millionaires). Layering and compositionality are conspicuous, our technique's specialty; and our dealing easily with unbounded state is another innovation.

Our contribution in detail is thus to formalise and prove a number of exemplary noninterference-style security protocols *while moving through layers of abstraction* and *in some cases with unbounded state*. We aim for a method with the potential to scale, and to be automated, and moreover one which would guide a designer to an understanding of the implications of his proposed design—a paramount criterion for critical software. The millionaires illustrate the hierarchical approach: when written out in full, the code comprises roughly 30 intricate lines (Fig. 4); only abstraction controls this complexity. Finally, the proofs are lengthy; but crucially *they are boring*, comprising many tiny steps similar to those already automated in probabilistic program algebra [MCMG08], and thus easily checked.

3. The Shadow Model of security and refinement

We begin by summarising the Shadow Model of security [Mor06] for straight-line (terminating) programs. It extends the *noninterference model* of security [GM84] to determine an attacker's inferred knowledge of hidden (high-security) variables at each point in the computation; once that inferred knowledge is available in the semantics, we can then require that it *not be increased* by secure refinement.

In its original form (without loops), noninterference partitions variables between high-security- and low-security classes: we call them *hidden* and *visible*. A noninterference-secure program is then one where an attacker cannot infer *hidden* variables' initial values from *visible* variables' values (initial or final). With just two variables v, h of class visible, hidden resp. a possibly nondeterministic program thus takes initial states (v, h) to sets of final visible states v' and has meaning (denotation) 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.[1] Such a program r is *noninterference-secure* just when for any initial visible the set of possible final visibles is independent of the initial hidden [LJ00, SS01], that is when for any $v : \mathcal{V}$ we have $\big(\forall\, h_0, h_1 : \mathcal{H} \cdot r.v.h_0 = r.v.h_1\big)$.[2]

In our approach [Mor06] we extended this view, in several stages. The first was 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 $v : \mathcal{V}$, $h : \mathcal{H}$, and $v' : \mathcal{V}$ satisfying the condition $\big(\exists\, h_2' : \mathcal{H} \cdot (v', h_2') \in r_2.v.h\big)$, we have also for all $h' : \mathcal{H}$ that $(v', h') \in r_1.v.h$ implies $(v', h') \in r_2.v.h$.
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' cannot be improved by refinement: there can only be more possibilities for h', never fewer.

In this simple setting the two conditions together do not yet allow an attacker's ignorance of h strictly to increase: secure refinement seems to boil down to allowing decrease of nondeterminism in v but not in h. But strict increase of hidden nondeterminism *is* possible: we meet it later, in Sect. 4.3.

Still in the 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).

This is because r_2 reduces r_1's visible nondeterminism, but does not affect the hidden nondeterminism in h'. In r_3, however, variables v' and h' are correlated: they are either both 1, or both 0.

[1] This is isomorphically $(\mathcal{V} \times \mathcal{H}) \to \mathbb{P}\mathcal{V}$ in its "un-Curried" form. Further formulations include $(\mathcal{V} \times \mathcal{H}) \leftrightarrow \mathcal{V}$ or even $\mathbb{P}(\mathcal{V} \times \mathcal{H} \times \mathcal{V})$. Because all these are essentially the same, we refer uniformly to a "relational" semantics; and we will occasionally move between them without comment.

[2] We say, e.g. "program r" for brevity, meaning more precisely an r that is the meaning of some program.

3.1. Triples: the Shadow H of h records h's inferred values

In r_1 above the set of possible final values h' was $\{0, 1\}$ for each final v' separately. This set is called *The Shadow*, and represents explicitly an attacker's ignorance of that h': it is the smallest set of possibilities he must consider possible, by inference. 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. The (extended) output triples of the three example programs are then, respectively,

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_3 — $\{(0, 0, \{0\}),\ (1, 1, \{1\})\}$,

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 is, again, why $r_1 \not\sqsubseteq r_3$.

3.2. A Shadow Model for refinement and nontermination

As we have seen [Mor06], distinguishing between hidden and visible state is necessary to identify those refinements that preserve noninterference properties. In this section we introduce a formal model which extends that idea to include nontermination. We begin with two important properties of triples and of sets of them:

Definition 3.1 *Validity and consistency* We say that a triple (v, h, H) is *valid* when $h \in H$; we say that a set of valid triples C is *consistent* if whenever $(v, h, H) \in C$ then also $(v, h', H) \in C$ for any (other) $h' \in H$.

Let $\mathcal{T}$ be the set $\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H}$. The order we sketched in (i) and (ii) for pairs, above, can be expressed for triples by the order $\sqsubseteq_{\mathcal{T}}$ with $(v, h, H) \sqsubseteq_{\mathcal{T}} (v, h, H')$ just when $H \subseteq H'$ (and the first 2 components are equal); but that does not yet consider nontermination. To add nontermination we introduce a distinguished state $\bot$, and in this setting we regard it as a *visible* phenomenon. We extend $\sqsubseteq_{\mathcal{T}}$ over $\mathcal{T}$ to an order $\sqsubseteq_{\bot}$ over $\mathcal{T}_{\bot}$ defined $\mathcal{T} \cup \{\bot\}$, so that $\bot$ is subordinate to all triples: thus for $t, t' : \mathcal{T}_{\bot}$ we define

$$t \sqsubseteq_{\bot} t' := (t = \bot) \vee (t \sqsubseteq_{\mathcal{T}} t'). \tag{2}$$

Then we lift (2) to a pre-order between sets of triples (or $\bot$). For $A, A' \subseteq \mathcal{T}_{\bot}$ we define

$$\begin{aligned} & A \sqsubseteq_{\mathbb{P}} A' := \big(\forall\, t' : A' \cdot \big(\exists\, t : A \cdot t \sqsubseteq_{\bot} t'\big)\big) \\ \text{and} \quad & A \sim_{\mathbb{P}} A' := A \sqsubseteq_{\mathbb{P}} A' \text{ and } A' \sqsubseteq_{\mathbb{P}} A, \end{aligned} \tag{3}$$

using the Smyth construction [Smy78] for powerdomains: it is the standard way of making a pre-order (3) over sets from an underlying partial order (2) over elements. It implies, for example, that when $\bot \in A$ then $A \sqsubseteq_{\mathbb{P}} A'$ for any A', so that the presence of nontermination in A makes it subordinate to any other subset. On the other hand when nontermination is not present, the order reduces to the (Smyth) order for original noninterference.

Our model for noninterference secure programs is built on this as follows. We include $\bot$ formally among the (valid) triples and use $\mathcal{T}_{\bot}$ to be the valid triples of $\mathcal{T}$ together with the valid triple $\bot$. Then we write $\mathbb{S}\mathcal{T}$ for the nonempty and "maximally" consistent subsets of $\mathcal{T}_{\bot}$, where a consistent subset C of $\mathcal{T}_{\bot}$ is *maximal* just when for all consistent C' with $C \sim_{\mathbb{P}} C'$ we have $C' \subseteq C$. Since consistency is preserved by union, there is for any consistent C always a unique maximally consistent $C\Uparrow$ such that $C \sim_{\mathbb{P}} C\Uparrow$. Note that $\sqsubseteq_{\mathbb{P}}$ becomes a partial order (no longer just a pre-order) when restricted to $\mathbb{S}\mathcal{T}$, and that in fact for $C_{\{1,2\}} \in \mathbb{S}\mathcal{T}$ we have $C_1 \sqsubseteq_{\mathbb{P}} C_2$ just when $C_1 \supseteq C_2$, which is the usual situation for Smyth-constructed orders.[3]

Programs are then interpreted as certain functions from $\mathcal{T}_{\bot}$ to $\mathbb{S}\mathcal{T}$. The functions are those meanings r that preserve the underlying order in two senses: that whenever $t \sqsubseteq_{\bot} t'$ we have $r.t \sqsubseteq_{\mathbb{P}} r.t'$; and that $r.\bot = \mathcal{T}_{\bot}$. The first property is monotonicity (between $\sqsubseteq_{\bot}$ and $\sqsubseteq_{\mathbb{P}}$), and the second is strictness. Monotonicity implies that program refinement is preserved by contexts (Sect. 4.5 below), and strictness means that programs cannot recover from abortion.

[3] Directly from the definition of $\sqsubseteq_{\mathbb{P}}$ we have that if $C_1 \sqsubseteq_{\mathbb{P}} C_2$ then also $C_1 \sim_{\mathbb{P}} (C_1 \cup C_2)$, whence $(C_1 \cup C_2) \subseteq C_1$ because C_1 is maximal. Thus $C_2 \subseteq C_1$.

Definition 3.2 *Model for noninterference security* The model for noninterference refinement is defined as the pair $(\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}, \sqsubseteq_{\mathbb{P}})$, where $\mathcal{T}_\bot$ is the valid triples in $\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H}$ together with the valid "triple" $\bot$, and $\mathbb{S}\mathcal{T}$ is the set of maximally consistent subsets of $\mathcal{T}_\bot$, and $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$ is the set of monotone, strict functions between the sets given. The order $\sqsubseteq_{\hookrightarrow}$ on functions r, r' in $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$ is defined pointwise in the usual way, thus

$$r \sqsubseteq_{\hookrightarrow} r' \quad \text{iff} \quad \big(\forall\, t{:}\mathcal{T}_\bot \cdot r.t \sqsubseteq_{\mathbb{P}} r'.t\big)\,.$$

We write $\amalg$ for the $(\sqsubseteq_{\hookrightarrow})$-least element in $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$: it is the function that returns $\mathcal{T}_\bot$ applied to any triple.

From now on we write just $\sqsubseteq$ for all of these orders; relying on context to determine which is meant is harmless, since they are so closely related.

A final technical detail we need, for the semantic definitions to come, is up-completeness. We have

Lemma 3.1 Existence of least upper bounds *Let $r_0 \sqsubseteq r_1 \sqsubseteq \cdots$ be a $(\sqsubseteq)$-increasing chain of programs (actually, their meanings) in $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$. Then there is a program $\big(\bigsqcup i \mid i{\geq}0 \cdot r_i\big)$ also in $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$ that is least upper bound of the chain.*

Proof Since the order on the functions determined pointwise from the order on their codomain, we first consider the argument there, viz. within $\mathbb{S}\mathcal{T}$, where in fact the least upper bound of $C_0 \sqsubseteq C_1 \sqsubseteq \cdots$ is simply their intersection $\widehat{C} := \big(\cap i \mid i \geq 0 \cdot C_i\big)$. To show that, we recall that $\sqsubseteq$ is defined (although as a pre-order) on all of $\mathbb{P}\mathcal{T}_\bot$, so that the lub-properties of $\widehat{C}$ as a *set*, that it is an upper bound and that it is a minimal one, follow from standard properties of the Smyth construction. That leaves to be shown that $\widehat{C}$ is in fact within $\mathbb{S}\mathcal{T}$, for which we have

$\widehat{C}$ *contains only valid triples* Trivial property of intersection.

$\widehat{C}$ *is consistent* Trivial property of intersection.

$\widehat{C}$ *is maximal* Suppose $\widehat{C}{\sim}C$ for some consistent $C \in \mathbb{P}\mathcal{T}_\bot$. Then for all i we have $C_i \sqsupseteq \widehat{C}{\sim}C{\sim}C{\Uparrow}$, whence $C_i \sqsupseteq C{\Uparrow}$ and so $C_i \supseteq C{\Uparrow}$ since both C_i and $C{\Uparrow}$ are in $\mathbb{S}\mathcal{T}$. Thus $\widehat{C} = \big(\cap i \mid i \geq 0 \cdot C_i\big) \supseteq C{\Uparrow} \supseteq C$, and so $\widehat{C}$ is maximal.

Hence $\widehat{C}$ is in $\mathbb{S}\mathcal{T}$.

We finish off by showing that the proposed least upper bound, the function $\widehat{r}.t := \big(\cap i \mid i \geq 0 \cdot r_i.t\big)$ determined pointwise, is monotonic and strict. Monotonicity follows from distribution of $(\supseteq)$ through $\big(\cap i \cdot \cdots\big)$; and strictness is trivial. □

In the next section we shall use $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$ (from Definition 3.2) for the semantics of a small programming language.

4. A semantics for a small while-language

In Fig. 2 we set out the semantics of a small programming language which we will use for our case studies to follow: we discuss it in detail in Sect. 4.4, below, after giving some motivational examples. The language includes all the basic constructs, and indeed differs from classical relational presentations [Dij76] only in its treatment of visibility. That is, assignments to hidden and visible variables produce different effects with regard to the shadow component, as does hidden and visible nondeterminism. We use the term "classical semantics" to mean an interpretation of sequential programs which does not distinguish visibility; as a relation it delivers the set of outputs.

Each construct P is interpreted as $[\![P]\!]$, a function $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$ (which, as mentioned earlier, we sometimes use isomorphically in its Curried form). We show below that the semantics is well-defined, but before we do that we set out some examples, and discuss some of the constructs pertaining to noninterference specifically.

4.1. External and internal nondeterminism

Our definition of refinement is based on scale-up experiments with program algebra [Mor06, Mor09]. Our first observation is that the semantics enforces *perfect recall*, that visible variables reveal information even if subsequently overwritten and that any information revealed is captured by the semantics. This is because refinement must be *monotonic*, i.e. (A) that refinement of a program portion must refine the whole program; and (B) that conventional refinements involving v only must remain valid. Both principles (A, B) are required in order to be able to develop large (secure) programs via local reasoning over small portions.

Without perfect recall, overwriting 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 would be a violation of secure refinement for the *rhs* to reveal h while the *lhs* does not. Thus the premise "without perfect recall" is false.

A similar experiment applies to conditionals: because (A, B) validates

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

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

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. In general, if $P_1 \sqsubseteq P_2$, then 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 ignorance, produced by P_1 under the same circumstances.

From Fig. 2 we have, e.g. that $[\![h := 0 \sqcap h := 1]\!].v.h.H$ is $\{(v, 0, \{0\}), (v, 1, \{1\})\}$, yet the strictly more refined $[\![h :\in \{0, 1\}]\!].v.h.H$ is $\{(v, 0, \{0, 1\}), (v, 1, \{0, 1\})\}$. This is thus an example of a strict refinement where the two commands differ only by an increase of ignorance: they have equal nondeterminism classically, but in one case ($\sqcap$) it can be observed by the attacker and in the other case ($:\in$) it cannot. The "more ignorant" triple $(v, 0, \{0, 1\})$ is strictly justified by the "less ignorant" triple $(v, 0, \{0\})$, where we say "strictly" because $\{0\} \subset \{0, 1\}$.

4.2. Atomic programs

A classical and terminating program (i.e. one where we do not distinguish visibility, and without divergence) describes the input–output relationship between $\mathcal{V} \times \mathcal{H}$-pairs. Considered as a single, atomic action its shadow-enhanced semantics $\mathsf{addShadow}.r$ (i.e. where we take visibility into account) will then be a function in $\mathcal{T} \to \mathbb{P}\mathcal{T}$, determined as follows[4]:

Definition 4.1 *Atomic shadow semantics* Given a classical program meaning $r: \mathcal{V} \to \mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathcal{H})$ we define its *shadow enhancement* $\mathsf{addShadow}.r$ of type $\mathcal{T} \to \mathbb{P}\mathcal{T}$ so that $\mathsf{addShadow}.r.v.h.H \ni (v', h', H')$ just when both

(i) $\quad r.v.h \ni (v', h')$ — *classical*

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

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.

As a first example, let the syntax $x :\in S$ denote the classical program that chooses variable x's value from a nonempty set S. Assume here only that S is constant, not depending on v, h. Then from Definition 4.1 we have that

(i) Choosing v affects only v because
$$\mathsf{addShadow}.(v :\in S).v.h.H = \{v': S \cdot (v', h, H)\}$$

(ii) Choosing h affects both h *and* H, possibly introducing ignorance because
$$\mathsf{addShadow}.(h :\in S).v.h.H = \{h': S \cdot (v, h', S)\}$$

(iii) An assignment of hidden to visible "collapses" ignorance because
$$\mathsf{addShadow}.(v := h).v.h.H = \{(h, h, \{h\}\}$$

[4] A treatment of imposing atomicity on diverging classical programs is possible, but here we forgo it in the interests of simplicity.

	Program P	Pre-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	$\langle\!\langle P \rangle\!\rangle$	addShadow.("classical semantics of P")	
Sequential composition	$P_1; P_2$	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**	$(\!|P_t|\!).v.h.\{h'\!: H \mid E.v.h' = \textbf{true}\}$ **if** $E.v.h$ **else** $(\!|P_f|\!).v.h.\{h'\!: H \mid E.v.h' = \textbf{false}\}$	
Loop	**while** G **do** P **od**	$(\mu L).v.h.H$ where L is the unique function satisfying $[\![\textbf{if } G \textbf{ then skip else } P; X \textbf{ fi}]\!] = L.[\![X]\!]$ for all program texts X	

The pre-semantics $(\!|P|\!)$ of P of the same type $\mathcal{T} \to \mathbb{P}\mathcal{T}$ delivered e.g. by addShadow.r above; it is converted into actual semantics $[\![P]\!]$ via two technical "housekeeping" details. First we impose strictness uniformly, so that $[\![P]\!].\bot := \mathcal{T}_\bot$. Otherwise $[\![P]\!].v.h.H := ((\!|P|\!).v.h.H)\Uparrow$; that is we "close-up" the definitions given by choosing the maximally consistent representative. (That this is well defined follows from the compound defintions' being $(\sim)$-congruences, in turn a consequence of their monotonicity: apply it both ways.) Note however that for the loop we use $[\![\cdot]\!]$ in the definition of the functional L, since the least-fixed point is defined only up to equivalence in a pre-order (a nuisance); but in a partial order (under the usual conditions) it is unique, which we need for a definition of this kind.

The syntactically atomic commands A marked $\star$ have the property that $A = \langle\!\langle A \rangle\!\rangle$. (Note that **reveal** is not marked.) This is deliberate: syntactic atoms execute atomically.

The function 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 [Mor06, Mor09].

Note that semantics in this form assumes that v, h stand for *all* of the program variables (via a Cartesian product); multiple visibles and hiddens, and local variables, are addressed below.

Fig. 2. Semantics of commands

From (ii) and (iii) the composition addShadow.$(h :\in S)$; addShadow.$(v := h)$ first introduces ignorance: we do not know h's exact value "at the semicolon." But then the ignorance is removed: we deduce h's value, at the end, by observing v. The composition (ii); (iii) as a whole is nondeterministic, and it yields $\{x\!: S \cdot (x, x, \{x\})\}$ with v, h's common final value x drawn arbitrarily from S; but whatever that value turns out to be, it is known that h has it because H is a singleton.

4.3. The reveal statement

The nonclassical command **reveal** E publishes an expression E but changes no program variables; however it does possibly change the shadow. As an example consider **reveal** $h \div 2$, which reveals whether h is odd or even. Applied to the triples $(v, h, \{0, 1, 2\})$ it results in $\{(v, 0, \{0, 2\}), (v, 0, \{0, 2\}), (v, 1, \{1\})\}$, so that if h was odd, then its value (in this case) is entirely revealed. Note that **reveal** E only ever changes the shadow, and can only release information, so that **reveal** $E \sqsubseteq$ **skip**.

We return to reasoning with **reveal** statements in Sect. 5.5 below.

4.4. The programming language and its well-definedness

The definitions are given in Fig. 2 in terms of a "pre-semantic function" $(\!|\cdot|\!)$ that concentrates on the essence, avoiding clutter by not bothering with strictness or the need to construct maximally consistent outputs. Those details are handled uniformly, to give the "actual" semantic function $[\![\cdot]\!]$ as explained there.

We now show that the semantics in Fig. 2 is well-defined in terms of delivering meanings in the space defined in Definition 3.2. An issue specific to our construction is that the expressions on the right in Fig. 2 must all denote consistent subsets of $\mathbb{P}\mathcal{T}$: that is routine to verify for all but the loop, using structural induction for the compound cases. For the loop we refer to Lemma 3.1.

That leaves the other, general issue that $(\!|P|\!)$ must be monotonic (which suffices for monotonicity of $[\![P]\!]$).

Lemma 4.1 Monotonicity base case *For any syntactically atomic program P defined by the programming language of Fig. 2, the interpretation $(\!|P|\!)$ is monotonic.*

Proof This is clear by inspection, since for atomic statements (including **reveal**) if we suppose that $(v, h, H) \sqsubseteq (v, h, H')$ then $H \subseteq H'$, and the definitions for $(\!|P|\!)$ in these cases are all $(\subseteq)$-monotonic in H.

For atomicity brackets $\langle\!\langle P \rangle\!\rangle$ we rely on a similar $(\subseteq)$-monotonicity wrt. H in Definition 4.1. □

Now we proceed with the compound commands, via a technical lemma concerning composition of semantic functions:

Lemma 4.2 Monotonicity of functional composition *Let p, p', q be of type $\mathcal{T} \to \mathbb{P}\mathcal{T}$. If $p \sqsubseteq p'$, then also the following inequalities hold*:

$$\mathsf{lift}.q \circ p \;\sqsubseteq\; \mathsf{lift}.q \circ p' \;\text{ and }\; \mathsf{lift}.p \circ q \;\sqsubseteq\; \mathsf{lift}.p' \circ q.$$

Proof Let $t \in \mathcal{T}$. For the first inequality we note by assumption that $p.t \sqsubseteq p'.t$, which means that for any $u' \in p'.t$ there is some $u \in p.t$ such that $u \sqsubseteq u'$. By monotonicity (of q) we have that $q.u \sqsubseteq q.u'$. Now we may deduce that $\mathsf{lift}.q \circ p \sqsubseteq \mathsf{lift}.q \circ p'$ by first taking the union over all $u' \in p'.t$, and then over all $u \in p.t$.

The proof of $\mathsf{lift}.p \circ q \sqsubseteq \mathsf{lift}.p' \circ q$ is similar, relying only on Definition 3.2 and the definition of lift (Fig. 2). □

Lemma 4.2 gives us immediately the monotonicity of sequential compositions. Monotonicity for conditionals and for demonic choice are clear by inspection, based on the (assumed) monotonicity of their components.

For loops it is the usual argument that by mathematical induction over the terms of which the least fixed point is the limit we have that each term is monotonic individually; and taking limits preserves monotonicity.

4.5. Monotonicity of contexts

In Sect. 4.4 above we showed that for all programs P and triples t, t' with $t \sqsubseteq t'$ we have $[\![P]\!].t \sqsubseteq [\![P]\!].t'$. This is not the same thing (quite) as $(\sqsubseteq)$-monotonicity of contexts, since that requires that for two programs $P \sqsubseteq P'$ and context $\mathcal{C}$ we have $\mathcal{C}(P) \sqsubseteq \mathcal{C}(P')$. The argument taking the former to the latter is standard for small, imperative **while** languages, and so we omit it.

4.6. A discussion of looping programs

Looping programs are defined as fixed points. From Sect. 4.4 we see that the function created from the loop-body and the "**if** ... **fi**" of the condition defines a monotone function; Lemma 3.1 guarantees that the fixed point is well-defined.

Interestingly this definition, together with our assumption of total recall, means that an attacker is able to deduce the number of times a terminating loop has iterated. Referring to Fig. 2 we unfold the loop once to obtain the equality

$$\textbf{while } G \textbf{ do } P \textbf{ od} = \textbf{if } G \textbf{ then } \;P;\; (\textbf{while } G \textbf{ do } P \textbf{ od}) \textbf{ fi},$$

implying that on any single iteration the attacker can observe whether the loop terminated that time. Repeating the argument implies—in principle—that, akin to "timing attacks," he can use knowledge of the number of iterations to deduce other facts about the hidden state. For example consider this program operating over hidden natural numbers h, h' in $\mathbb{N}$:

$$h' := 0;\;\; \textbf{while } h' < h \textbf{ do } h' := h' + 1 \textbf{ od}. \tag{4}$$

It increments hidden h' until it reaches h, so revealing h's value in the process to any observer who can count the loop iterations and knows the code. Our observer has both these capabilities; and thus we expect that the above should be equivalent to

$$h' := h;\;\; \textbf{reveal } h, \tag{5}$$

and in Sect. 6 we show this to be so.

4.7. Multiple agents, and the attacker's capabilities

In a multi-agent system each agent has a limited knowledge of the system state, determined by his *point of view*; and different agents have different views. The above semantics reflects A's viewpoint, say, by interpreting variables declared to be $\mathbf{vis}_{\text{list}}$ as visible (v) variables if A is in list and as hidden (h) 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.
- $\mathbf{vis}_{\text{list}}$ means the associated variable is visible to all agents in the (nonempty) list, and is hidden from all others (including third parties).
- $\mathbf{hid}_{\text{list}}$ means the associated variable is hidden from all agents in the list, and is visible to all others (including third parties).

For example in (1), from A's viewpoint the specification would be interpreted with a and x 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 x is still visible.

From agent A's point of view (say) an attacker uses a run-time debugger to single-step through an execution of the program. Each step's size is determined by atomicity, either implied syntactically or given by $\langle\!\langle \cdot \rangle\!\rangle$; when the program is paused, the current point in the program source-code is indicated; and hovering over a variable reveals its value provided its annotation (in this case) makes it visible to A, e.g. "yes" for $\mathbf{vis}_A$ or $\mathbf{hid}_B$, and "no" for $\mathbf{hid}_A$ or $\mathbf{vis}_B$.

Conventionally, a successful attack is one that "breaks the security." For us, however, a successful attack is one that *breaks the refinement*: if we claim that $P \sqsubseteq Q$, and yet an attacker subjects Q to hostile tests that reveal something P cannot reveal, then our claimed refinement must be false (and we'd better review the reasoning that seemed to prove it). Crucially however we will have suffered a failure of calculation, not of guesswork: only the former can be audited.

The conventional view is a special case of ours: if P reveals nothing, then $P \sqsubseteq Q$ means that also Q must reveal nothing. Thus a successful attack with such a specification P is one in which Q is forced to reveal anything at all.

Finally, if a refinement is valid yet an insecurity is discovered (relative to some informal requirement), then the security-preservation property of refinement means that the insecurity *was already present* in the specification.

4.8. Agent X and compositionality

Observe that the agent-centered approach is not per se compositional. To ensure compositionality—and in particular that refinement is preserved *by all contexts*—we must include consideration of an "abstract agent" called agent X when proving overall refinement. Agent X is an unnamed agent who represents any possible future agents which might emerge if the program is ever used as a subprotocol within a hierarchical design. In our proofs, agent X's viewpoint is determined by the projection such that only universally visible variables become the visible state, and all other subscripted variables become hidden.

Thus two programs P, P' containing named agents A, B, ... are deemed to be related $P \sqsubseteq P'$ *only* if they are so refined with respect to all declared viewpoints, together with agent X's viewpoint. Our case study in Sect. 10 below illustrates further agent X's role in ensuring compositionality.

5. Compositional proof rules

In this section we gather together a set of proof rules which arose as common patterns during the detailed analysis of the case studies below. Some of the rules—only summarised—have already appeared elsewhere; for the others we explain how they are justified by the semantics set out above. We will see all of them illustrated in Sects. 6–11.

5.1. Properties—and utility—of atomicity brackets $\langle\!\langle \cdot \rangle\!\rangle$

The atomicity brackets $\langle\!\langle \cdot \rangle\!\rangle$ treat their contents as a single classical command, and thus classical *equality* (although not classical refinement) can be used within them. In simple cases atomicity is preserved by sequential composition, but not in general:

Lemma 5.1 atomicity and composition *Given two classical terminating 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. 2 on the right.*

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

$$[\![\langle\!\langle P_1; P_2 \rangle\!\rangle]\!].v.h.H \ni (v', h', H')$$

iff $(r_1;\ r_2).v.h \ni (v', h')$ "Definition 4.1"
and $H' = \{h'' : H, h' : \mathcal{H} \mid (r_1;\ r_2).v.h'' \ni (v', h') \cdot h'\}$

iff $\big(\exists\, v^\flat : \mathcal{V}, h^\flat : \mathcal{H} \mid r_1.v.h \ni (v^\flat, h^\flat) \wedge r_2.v^\flat.h^\flat \ni (v', h')\big)$ " composition"
and $H' = \{h'' : H, h' : \mathcal{H}, v^\sharp : \mathcal{V}, h^\sharp : \mathcal{H} \mid r_1.v.h'' \ni (v^\sharp, h^\sharp) \wedge r_2.v^\sharp.h^\sharp \ni (v', h') \cdot h'\}$

On the other hand, with piecewise atomicity we have

$$[\![\langle\!\langle P_1 \rangle\!\rangle; \langle\!\langle P_2 \rangle\!\rangle]\!].v.h.H \ni (v', h', H')$$

iff " composition"
$(\exists\, v^\natural : \mathcal{V}, h^\natural : \mathcal{H}, H^\natural : \mathbb{P}\mathcal{H} \mid [\![\langle\!\langle P_1 \rangle\!\rangle]\!].v.h.H \ni (v^\natural, h^\natural, H^\natural) \wedge [\![\langle\!\langle P_2 \rangle\!\rangle]\!].v^\natural.h^\natural.H^\natural \ni (v', h', H'))$

iff "Definition 4.1"
$(\exists\, v^\natural : \mathcal{V}, h^\natural : \mathcal{H}, H^\natural : \mathbb{P}\mathcal{H} \mid$
$\quad r_1.v.h \ni (v^\natural, h^\natural)$
$\quad \wedge\ H^\natural = \{h^+ : \mathcal{H}, h^- : H \mid r_1.v.h^- \ni (v^\natural, h^+) \cdot h^+\}$
$\quad \wedge\ r_2.v^\natural.h^\natural \ni (v', h')$
$\quad \wedge\ H' = \{h^+ : \mathcal{H}, h^- : H^\natural \mid r_2.v^\natural.h^- \ni (v', h^+) \cdot h^+\}\ )$

iff "propositional calculus; eliminate $H^\natural$"
$(\exists\, v^\natural : \mathcal{V}, h^\natural : \mathcal{H} \mid$
$\quad r_1.v.h \ni (v^\natural, h^\natural) \wedge r_2.v^\natural.h^\natural \ni (v', h')$
$\quad \wedge\ H' = \{h^+ : \mathcal{H}, h^- : \{h^+ : \mathcal{H}, h^- : H \mid r_1.v.h^- \ni (v^\natural, h^+) \cdot h^+\} \mid r_2.v^\natural.h^- \ni (v', h^+) \cdot h^+\}\ )$

iff "rename bound variables"
$(\exists\, v^\natural : \mathcal{V}, h^\natural : \mathcal{H} \mid$
$\quad r_1.v.h \ni (v^\natural, h^\natural) \wedge r_2.v^\natural.h^\natural \ni (v', h')$
$\quad \wedge\ H' = \{h^+ : \mathcal{H}, h^- : \{h^\pm : \mathcal{H}, h^\mp : H \mid r_1.v.h^\mp \ni (v^\natural, h^\pm) \cdot h^\pm\} \mid r_2.v^\natural.h^- \ni (v', h^+) \cdot h^+\}\ )$

iff "collapse comprehensions"
$(\exists\, v^\natural : \mathcal{V}, h^\natural : \mathcal{H} \mid$
$\quad r_1.v.h \ni (v^\natural, h^\natural) \wedge r_2.v^\natural.h^\natural \ni (v', h')$
$\quad \wedge\ H' = \{h^+ : \mathcal{H}, h^\pm : \mathcal{H}, h^\mp : H \mid r_1.v.h^\mp \ni (v^\natural, h^\pm) \wedge r_2.v^\natural.h^\pm \ni (v', h^+) \cdot h^+\}\ )$

iff “rename bound variables”

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

which is quite 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

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

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

. . . iff “introduce D”

$$
\begin{array}{ll}
 & \big(\exists\, h^\flat{:}\mathcal{H} \mid r_1.v.h \ni (D.v.v', h^\flat) \wedge r_2.(D.v.v').h^\flat \ni (v', h')\big) \\
\text{and} & H' = \{h''{:}H, h'{:}\mathcal{H}, h^\sharp{:}\mathcal{H} \\
 & \qquad\qquad \mid r_1.v.h'' \ni (D.v.v', h^\sharp) \wedge r_2.(D.v.v').h^\sharp \ni (v', h') \\
 & \qquad\qquad \cdot h'\}
\end{array}
$$

And for the second case we can continue

. . . iff “introduce D”

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

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

That completes the proof. □

Lemma 5.1 is as significant when its conditions are *not* met as when they are. It means for example that we cannot conclude from Lemma 5.1 that $\langle\!\langle v := h;\ v := 0\rangle\!\rangle = \langle\!\langle v := h\rangle\!\rangle;\ \langle\!\langle v := 0\rangle\!\rangle$, 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 Definition 4.1[5]

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

(ii) Whereas on the right we have $(\langle\!\langle v := h\rangle\!\rangle;\ \langle\!\langle v := 0\rangle\!\rangle).v.h.H = \{(0, h, \{h\})\}$

This is perfect recall again. More interesting is the utility of introducing atomicity temporarily in a derivation, as illustrated in Sect. 7 below: when applicable, we can infer security properties via (simpler) classical equalities within $\langle\!\langle\cdot\rangle\!\rangle$.

5.2. Classical reasoning within atomicity brackets

The semantics allows much of the reasoning to be classical, provided it is equality reasoning within atomicity brackets.

Lemma 5.2 Classical equalities *Let P, Q be programs. If $P = Q$ holds when P, Q are interpreted classically, then $\langle\!\langle P\rangle\!\rangle = \langle\!\langle Q\rangle\!\rangle$ holds also.*

Proof Follows directly from Definition 4.1. □

Observe that we cannot weaken Lemma 5.2's assumption of equality to refinement. For example, classically we have the refinement $v{:}\in\{0, 1\};\ h{:}\in\{0, 1\} \sqsubseteq h{:}\in\{0, 1\};\ v := h$ (and in fact it is a strict refinement) yet we *do not* have that $\langle\!\langle v{:}\in\{0, 1\};\ h{:}\in\{0, 1\}\rangle\!\rangle$ is refined by $\langle\!\langle h{:}\in\{0, 1\};\ v := h\rangle\!\rangle$. The former reveals nothing about h, but the latter reveals everything, and therefore is “less secure.”

[5] We omit semantic brackets here, to avoid clutter.

5.3. Reducing atomicity

With Lemma 5.2 we have access to standard reasoning, provided it is equality-based; however one of the reasons leading to the intricacy of the protocols is that we may not assume the level of atomicity that the standard reasoning alone would allow. The next lemma sets out a simple, but highly practical specialisation of Lemma 5.1 for increasing atomicity.

Lemma 5.3 Single assignment rule *If no visible variable is set* (*assigned-to*) *more than once in program* $P; Q$ *then* $\langle\!\langle P; Q \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle; \langle\!\langle Q \rangle\!\rangle$.

Proof There are three cases. If v is not set at all, then Lemma 5.1 applies immediately. If v is set exactly once, then either it is set in P or in Q, and in both cases we may deduce the intermediate value of v from its initial or final values, and so Lemma 5.1 applies. In detail: suppose v is set in P, but not in Q; in that case its final value (after Q) is the same as its intermediate value. Similarly if v is set in Q but not in P then its intermediate value is the same as its initial value. □

5.4. Scoping rules

Our examples frequently make use of local variables whose scope is indicated by $|[\cdots]|$. Refinement however is judged according to the persistent global variables only [Mor06, Mor09]. We use standard scoping rules like

$$|[\textbf{vis } v, v';\ \textbf{hid } h, h' \cdot\ P;\ \ Q]| = |[\textbf{vis } v\ \textbf{hid } h \cdot\ P;\ \ |[\textbf{vis } v';\ \textbf{hid } h' \cdot\ Q]|\ ]|, \tag{6}$$

which, as usual, apply whenever P contains no reference to the locally declared v', h', i.e. that they are not free in P.

When the local block contains only assignments to local hidden variables the result is always equivalent to **skip**:

$$|[\textbf{hid } h \cdot P]| = \textbf{skip}, \quad \text{if } P \text{ assigns only to } h \tag{7}$$

since (a) the assignments cannot release any information thus there is no effect on the shadow variables security, and (b) h is local, thus there is no effect on the global variables either.

Note that (7) applies only to hidden variables. For example $|[\textbf{vis } v \cdot v := h]|$ is not **skip**, when h is a global hidden variable, since the shadow describing h's uncertainty, is changed to indicate that the observer may learn the exact value of hidden h by reading visible v.

5.5. Summary of the algebra of revelations

Elsewhere [MM08, MM10] we showed how **reveal** x satisfies some useful algebraic laws. We summarise them here for easy reference further below.

5.5.1. Basic refinements

reveal statements specify an explicit information release with no other change of variables. Since the refinement order preserves ignorance any **reveal** statement which explicitly leaks no more information than another will be a refinement. For example let $h \ominus 1$ abbreviate $h - 1$ **max** 0; then we have

reveal h	$\sqsubseteq$	**reveal** $h \ominus 1$	Values 0, 1 are not distinguished on the right.
reveal $h \ominus 1$	$\not\sqsubseteq$	**reveal** h	Values 0, 1 are not distinguished on the left, but they are at right.

Informally, we can see that if the observer knows the result $h - 1$, then he can deduce h, but if he observes 0 from the expression $h - 1$ **max** 0 (as in the case that $h - 1$ is negative) then he knows only that $h \leq 1$. Thus an observer observing **reveal** $h \ominus 1$ remains strictly more ignorant than one observing **reveal** h.

5.5.2. *Revelations and deductions*

Shadow refinement accommodates the deductions that observers can make when they have several pieces of information to hand. For example if E and F are two expressions, then revealing them both, is the same as revealing one after another in a sequential composition, as in

$$\textbf{reveal}\ E;\ \ \textbf{reveal}\ F = \textbf{reveal}\ (E, F). \tag{8}$$

We will use this idea in our case studies, specifically for exclusive-or: we have

	$\textbf{reveal}\ x \oplus y;\ \ \textbf{reveal}\ y \oplus z$	"Write $\oplus$ for exclusive-or"
$=$	$\textbf{reveal}\ (x \oplus y, y \oplus z)$	"(8)"
$=$	$\textbf{reveal}\ (x \oplus y, x \oplus z)$	"See below"
$=$	$\textbf{reveal}\ x \oplus y;\ \ \textbf{reveal}\ x \oplus z.$	"(8) in both directions"

For the "see below," we observe that both $x \oplus y$ and $x \oplus z$ can be deduced from the facts $x \oplus y$ and $y \oplus z$ (e.g. $x \oplus z = x \oplus y \oplus y \oplus z$), and *vice versa*, and thus the semantics of the two reveals is the same.

Finally we note that control structures can leak information. For example, we have the equality

$$\textbf{if}\ E\ \textbf{then}\ \ \textbf{skip fi} = \textbf{reveal}\ E. \tag{9}$$

Thus if E is an expression involving hidden variables, the evaluation of E as a branching condition implies a potential update of the shadow H, in some contexts.

More generally, we can place a **reveal** statement in context, wherever the shadow variable will be unaffected, as in

$$\textbf{if}\ E\ \textbf{then}\ \ P\ \textbf{else}\ Q\ \textbf{fi} = \textbf{if}\ E\ \textbf{then}\ \ \textbf{reveal}\ E;\ P\ \textbf{else}\ Q\ \textbf{fi}. \tag{10}$$

(It would also be valid to place the **reveal** before Q, or both.)

We shall see examples of all of these equalities in our case studies to come.

6. Terminating loops have unique fixed points

In Sect. 4.6 we remarked that we should have the equality

$$h' := 0;\ \ \textbf{while}\ h' < h\ \textbf{do}\ h' := h' + 1\ \textbf{od} = h' := h;\ \ \textbf{reveal}\ h,$$

that is that the implicit information flow resulting from an observer's counting the loop iterations will reveal the value of h, since the observer also knows from the code that h' begins counting from 0. We now show how that is done.

When a loop terminates from all initial states, we can show—as in traditional program semantics—that the fixed point is unique. We give the proof here for finite state spaces only, as that is simpler and is sufficient for our examples. For the infinite case we can extend the arguments by imposing compactness on the result sets [AJ94].

First we have a technical lemma:

Lemma 6.1 Dead code *If a* (*semantic*) *context* C *built from our programming language of Fig. 2 is such that* $\bot \notin C.\amalg.t$ *for all initial states* t*, then in fact* C *is a constant function* (*i.e. in its first argument*).

Proof (Sketch: the rigour of a formal structural induction is unnecessary, since the idea is so simple.) The only way the presence of $\amalg$ in $C.\amalg.t$ can avoid any divergence at all is if for no initial state t does execution reach the always diverging $\amalg$: that means the substitution positions for $\amalg$ must be "dead code," sitting inside a loop that is never entered or on a conditional branch that is never taken. If that is so, then it does not matter what appears in those positions, since it is never executed, and so C must be a constant function (of programs). □

Lemma 6.2 Unique fixed points over finite state spaces *Let* **while** G **do** P **od** *be a loop, and let its related functional* L *be defined as in Fig. 2. If* $\mathcal{T}$ *is finite and* $\bot \notin (\mu\, L).t$ *for any triple* $t \in \mathcal{T}$ *then the least fixed point of* L *is in fact its only fixed point.*

Proof If $\mathcal{V}, \mathcal{H}$ are finite, so that $\mathcal{T}_\bot$ and hence $\mathbb{S}\mathcal{T}$ are finite, then the denotation type $\mathcal{T}_\bot \hookrightarrow \mathbb{S}\mathcal{T}$ will also be finite. That means that the chain defining $\mu\, L$ is finite in length (its iterates $L^n.\amalg$ being drawn from a finite set), and thus that $\mu\, L = L^N.\amalg$ for some (finite) N, where N is the effective length of the chain.

But from our assumption we have $\bot \notin L^N.\Pi.t$ for any triple t, and from Lemma 6.1 just above that means that L^N must be a constant function. That being so, let l be *any* fixed point of L. We then have $l = L^N.l = L^N.\Pi = \mu\, L$. □

The significance of Lemma 6.2 is that is encourages the use of program algebra for reasoning about terminating loops: if for some convenient P we can manipulate **if** G **then** $body;\ P$ **fi** to become P again, and if in addition the loop **while** G **do** $body$ **od** terminates, then we have shown that the loop is actually equal to P.

We now return to our example. To use fixed-point reasoning, we must treat the loop on its own, i.e. without its initialisation (since, with its initialisation, it is no longer an obvious fixed point). Thus we define

$$loop := \textbf{while}\ h' < h\ \textbf{do}\ h' := h' + 1\ \textbf{od},$$

and we will show it to be equal to

$$spec := \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h).$$

This is necessarily more general than the actual result we're after, since we must capture the behaviour of *loop* from all initial values of h', not just from 0, in order to establish its fixed-point properties.

Since the loop terminates unconditionally, and there is thus a unique fixed point, we only need show that $spec = \textbf{if}\ h' < h\ \textbf{then}\ \ h' := h' + 1;\ spec\ \textbf{fi}$, for which we reason as follows:

$$\begin{array}{lll}
 & \textbf{if}\ h' < h\ \textbf{then}\ \ h' := h' + 1;\ spec\ \textbf{fi} & \\
= & \textbf{if}\ h' < h\ \textbf{then}\ \ h' := h' + 1;\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h)\ \ \textbf{fi} & \text{“Definition of } spec\text{”} \\
= & \textbf{if}\ h' < h\ \textbf{then}\ \ h' := h' + 1;\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h)\ \ \textbf{else skip fi} & \\
= & \textbf{if}\ h' < h & \text{“When } h' \geq h \text{ in the } \textbf{else}\text{-part we have} \\
 & \quad \textbf{then}\ \ h' := h' + 1;\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \quad \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) \\
 & \quad \textbf{else}\ \ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \quad \text{equals}\ \ \ \textbf{reveal}\ 0;\ \ h' := h' \\
 & \textbf{fi} & \quad \text{equals}\ \ \ \textbf{skip}\text{”} \\
= & \textbf{if}\ h' < h & \text{“Commute } h' := h' + 1 \text{ and } \textbf{reveal}\text{”} \\
 & \quad \textbf{then}\ \ \textbf{reveal}\ (h \ominus (h' + 1));\ \ h' := h' + 1;\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \quad \textbf{else}\ \ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \textbf{fi} & \\
= & \textbf{if}\ h' < h & \text{“Recall Sect. 5.5.2: simplify } \textbf{reveal} \text{ under condition } h' < h \text{ of } \textbf{then}\text{-part”} \\
 & \quad \textbf{then}\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := h' + 1;\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \quad \textbf{else}\ \ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \textbf{fi} & \\
= & \textbf{if}\ h' < h & \text{“Combine two assignments to } h'\text{”} \\
 & \quad \textbf{then}\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h' + 1)\ \textbf{max}\ h & \\
 & \quad \textbf{else}\ \ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \textbf{fi} & \\
= & \textbf{if}\ h' < h & \text{“If } h' < h \text{ then we have } (h' + 1)\ \textbf{max}\ h\ =\ h\ =\ h'\ \textbf{max}\ h\text{”} \\
 & \quad \textbf{then}\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \quad \textbf{else}\ \ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \\
 & \textbf{fi} & \\
= & \textbf{reveal}\ (h' < h); & \text{“Reveal } \textbf{if}\text{-condition via implicit flow,} \\
 & \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \quad \text{and collapse identical branches”} \\
= & \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \text{“Recall Sect. 5.5.2: knowing value of } h \ominus h' \text{ implies knowing whether } h' < h\text{”} \\
= & spec. &
\end{array}$$

That gives our straight line equivalent for *loop*; note that, in spite of that, our reasoning above did not involve anything other than loop-free program fragments. With that equivalence, we can now reason in context that

$$
\begin{array}{lll}
 & h' := 0;\ \ \textbf{while}\ h' < h\ \textbf{do}\ h' := h' + 1\ \textbf{od} & \\
= & h' := 0;\ \ \textit{loop} & \\
= & h' := 0;\ \ \textit{spec} & \text{“above”} \\
= & h' := 0;\ \ \textbf{reveal}\ (h \ominus h');\ \ h' := (h'\ \textbf{max}\ h) & \\
= & h' := 0;\ \ \textbf{reveal}\ (h \ominus 0);\ \ h' := (0\ \textbf{max}\ h) & \text{“Use that } h' = 0\text{”} \\
= & \textbf{reveal}\ h;\ \ h' := h, & \text{“Values are natural numbers”}
\end{array}
$$

which is just what we claimed: iteration through the loop has revealed h without any assignments to visible variables at all.

We emphasise that the algebraic reasoning above, over small nonlooping programs, has demonstrated rigorously and without operational or other informal arguments the effect of iterated implicit flow.

7. First case study: the encryption lemma

For Booleans x, y we write $(x \oplus y) := E$ to abbreviate the specification statement $x, y{:}[x \oplus y = E]$, thus an atomic command that sets x, y nondeterministically so that their exclusive-or equals E [Mor94]. By making the command atomic, we have $(x \oplus y := E) = \langle\!\langle x, y{:}[x \oplus y = E] \rangle\!\rangle$ by definition.

A very common pattern in noninterference-style protocols is the idiom $\textbf{|[vis}\ v;\ \textbf{hid}\ h' \cdot\ (v \oplus h') := h\ \textbf{]|}$ in the context of a declaration **hid** h; it is equivalent classically to **skip** because it assigns only to local variables, whose scope is indicated by the brackets $\textbf{|[} \cdots \textbf{]|}$. As our first example of secure refinement (actually equality) we show it is *security*-equivalent to **skip** also, in spite of its assigning a hidden *rhs* (variable h) to a partly visible *lhs* (includes v). We have via Shadow-secure program algebra the equalities

$$
\begin{array}{lll}
 & \textbf{|[vis}\ v;\ \textbf{hid}\ h' \cdot\ v \oplus h' := h\ \textbf{]|} & \\
= & \textbf{|[vis}\ v;\ \textbf{hid}\ h' \cdot\ \langle\!\langle v, h'{:}[v \oplus h' = h] \rangle\!\rangle\ \textbf{]|} & \text{“defined above”} \\
= & \textbf{|[vis}\ v;\ \textbf{hid}\ h' \cdot\ \langle\!\langle v :\in \{0,1\};\ h' := h \oplus v \rangle\!\rangle\ \textbf{]|} & \text{“classical reasoning within } \langle\!\langle\cdot\rangle\!\rangle\text{”} \\
= & \textbf{|[vis}\ v;\ \textbf{hid}\ h' \cdot\ \langle\!\langle v :\in \{0,1\} \rangle\!\rangle;\ \langle\!\langle h' := h \oplus v \rangle\!\rangle\ \textbf{]|} & \text{“Lemma 5.3”} \\
= & \textbf{|[vis}\ v;\ \textbf{hid}\ h' \cdot\ v :\in \{0,1\};\ h' := h \oplus v\ \textbf{]|} & \text{“syntactic atoms”} \\
= & \textbf{|[vis}\ v \cdot\ v :\in \{0,1\};\ \textbf{|[hid}\ h' \cdot\ h' := h \oplus v\ \textbf{]|]|} & \text{“(6)”} \\
= & \textbf{|[vis}\ v \cdot\ v :\in \{0,1\}\textbf{]|} & \text{“(7)”} \\
= & \textbf{skip}. & \text{“assignment of visibles to local visible is } \textbf{skip}\text{”}
\end{array}
$$

The overall chain of equalities establishes our EL as required.

8. Second case study: Sect. 7 ⇒ oblivious transfer

The *oblivious transfer protocol* builds on Sect. 7. An agent A transfers to agent B one of two secrets, as B chooses: but A does not learn which secret B chose; and B does not learn the other secret. The protocol is originally due to Rabin [Rab81]; we use Rivest's specialisation of it [Riv99]. Its specification is

$$
\begin{array}{ll}
\textbf{vis}_A\ m_0, m_1; & \text{“Oblivious Transfer specification”} \\
\textbf{vis}_B\ c{:}\,\mathsf{Bool}, m; & \\
\quad m := (m_1 \lhd c \rhd m_0), & \Leftarrow \text{We write } (\textit{left} \lhd \textit{condition} \rhd \textit{right}) \text{ [Hoa85].}
\end{array}
$$

where the variables without scope brackets are global, and are assumed subsequently. It is implemented via a third, trusted party C who contributes *before* the protocol begins, and indeed before A, B need even have decided what their variables' values are to be. A complete derivation is published elsewhere [Mor09], and it relies on the EL of Sect. 7.

In brief (and approximately), agent C gives two secret keys $k_{\{x,y\}}$ to A; and as well C gives one of those keys to B, telling him which one it is; agent C then leaves. When the protocol proper begins, agent B instructs A to encrypt $m_{\{0,1\}}$ either with $k_{\{x,y\}}$ or $k_{\{y,x\}}$ resp. so as to ensure B holds the correct key for the value he wants to decode. Agent A sends both encrypted values to B. Because A sends both, he cannot tell which B really wants; because B holds only one key, he can decrypt only his choice. The derivation is also given in Appendix B.

9. Third case study: The Lovers' Protocols

The Lovers' Protocols (see for example "Dating without embarrassment" [Sch]) in this section provide our first examples of two-party computations, and form the backbone of the later derivation of the Millionaires' Protocol. Throughout we assume two agents A, B.

9.1. Section 8 ⇒ Lovers' Protocol I

In this simple protocol agent A knows a Boolean a and agent B knows a Boolean b; they construct two Boolean outcomes a', b' known by A, B resp. so that

1. neither agent learns anything more about $a \wedge b$ as a result of learning its own a' or b' (as well as knowing its own a, b); and
2. the exclusive-or $a' \oplus b'$ reveals $a \wedge b$ without revealing anything more about either of a, b to any agent, whether A, B or some third party.

Here is the derivation; remember that each step has to be valid from both A and B's point of view. We have

$\mathbf{vis}_A\ a, a';\ \mathbf{vis}_B\ b, b';$ ⇐ Global variables: assumed below. "specification"
$(a' \oplus b') := a \wedge b$

$=$ $\langle\!\langle\ a' :\in \{0,1\};\quad b' := (a \wedge b) \oplus a'\ \rangle\!\rangle$ "atomicity reasoning: compare EL"
$=$ $a' :\in \{0,1\};\quad b' := (a \wedge b) \oplus a'$ "Lemma 5.1: compare EL"
$=$ $a' :\in \{0,1\};\quad b' := (a \lhd b \rhd 0) \oplus a'$ "Boolean algebra: **true** is 1, **false** is 0"

$=$ $a' :\in \{0,1\};$ "Boolean algebra"
$b' := (a \oplus a' \lhd b \rhd a').$ ⇐ Implemented by *Oblivious Transfer.*

Our semantics Sect. 3 plays two roles here, in the background: it legitimises the manipulations immediately above that introduced OT into the implementation, which in Sect. 13 we call *horizontal* reasoning. And it assures us (compositionality/monotonicity) that when OT is in its turn replaced by a still lower-level implementation *derived elsewhere, but in the same semantics*, the validity will be preserved: that is *vertical* reasoning.

9.2. Sections 7 and 9.1 ⇒ Lovers' Protocol II from Fig. 1

The second Lovers' Protocol extends the first: here even the incoming values a, b are available only as "$\oplus$-shares" so that $a = a_A \oplus a_B$ and $b = b_A \oplus b_B$, just as they might have been constructed by an Lovers' Protocol I (LP1). That is agent A knows a_A and b_A; agent B knows a_B and b_B; but neither knows a or b. We want to construct a', b' known by A, B resp. so that $a' \oplus b' = (a_A \oplus a_B) \wedge (b_B \oplus b_A) = a \wedge b$. We have

$\mathbf{vis}_A\ a', a_A, b_A;$ ⇐ These globals assumed below. "specification"
$\mathbf{vis}_B\ b', a_B, b_B;$
$(a' \oplus b') := (a_A \oplus a_B) \wedge (b_A \oplus b_B)$

$=$ $(a' \oplus b') := a_A \wedge b_A \ \oplus\ a_A \wedge b_B \ \oplus\ a_B \wedge b_A \ \oplus\ a_B \wedge b_B$ "Boolean algebra"

$=$ $[\![\ \mathbf{vis}_A\ r_A;\ \mathbf{vis}_B\ w_B;$ "EL for A, and for B (different visibilities), where h is the expression $a_A \wedge b_B$; then scope"
$(r_A \oplus w_B) := a_A \wedge b_B;$
$(a' \oplus b') := a_A \wedge b_A \ \oplus\ a_A \wedge b_B \ \oplus\ a_B \wedge b_A \ \oplus\ a_B \wedge b_B\ ]\!]$

$=$ $[\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B;$ "EL for A, and for B; then scope"
$(r_A \oplus w_B) := a_A \wedge b_B;\ \ (r_B \oplus w_A) := a_B \wedge b_A;$
$(a' \oplus b') := a_A \wedge b_A \ \oplus\ a_A \wedge b_B \ \oplus\ a_B \wedge b_A \ \oplus\ a_B \wedge b_B\ ]\!]$

$$
\begin{array}{lll}
= & [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B; & \text{“Program- and Boolean algebra”} \\
& \quad (r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A; & \\
& \quad (a' \oplus b') := a_A \wedge b_A \oplus r_A \oplus w_A \quad \oplus \quad w_B \oplus r_B \oplus a_B \wedge b_B\ ]\!] & \\
\sqsubseteq & [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B; & \text{“see below”} \\
& \quad (r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A; \quad \Leftarrow \text{Implemented by LP1.} & \\
& \quad a' := a_A \wedge b_A \ \oplus\ r_A \oplus w_A; & \\
& \quad b' := w_B \oplus r_B \ \oplus\ a_B \wedge b_B\ ]\!]. &
\end{array}
$$

The last step is clearly a classical refinement; it is secure (as well) because A, B already know the values revealed to them by the individual assignments to a', b'. Note that it is a proper refinement, not an equality.[6]

10. Fourth case study: a refinement that is not

It is one thing to show that we *can* derive putatively security-respecting implementations of our specifications; but positive examples alone are not enough. What does the cost of rigour actually buy?

In this section we first give an example of a refinement that looks like it *should* go through, but does not; then we investigate why it does not; and finally we comment on the nature of the disaster avoided. What the rigour buys is *not* having disasters like that.

10.1. A small error here: a catastrophe over there

The simple two-party-propositional scheme above is of the form $(a' \odot b') := a \otimes b$ with a, a' visible only to agent A and b, b' visible only to B; it is a generalisation of what we saw at (1) in Sect. 2. In principle the operators $\odot$, $\otimes$ could even be the same, so that $(a' \wedge b') := a \wedge b$ would reveal the conjunction of original a, b by choosing (other) a', b' with the same conjunction, but whose separate values still could not be used to infer the originals. In practice, however, exclusive-or tends to be used on the left, as indeed is the case in Protocols I, II of Sects. 9.1 and 9.2 above.

An interesting case however is when *both* operators are $\oplus$, i.e. we specify $(a' \oplus b') := a \oplus b$. It seems obvious that this can be implemented $a', b' := a, b$ from A, B's point of view, and indeed it is easily shown:

$$
\begin{array}{lll}
 & \mathbf{vis}_A\ a, a';\ \mathbf{vis}_B\ b, b'; & \\
 & \quad (a' \oplus b') := a \oplus b & \\
= & \quad a' :\in \{0, 1\}; & \text{“for all agents”} \\
 & \quad b' := a' \oplus (a \oplus b) & \\
\sqsubset & \quad a' := a; & \text{“for agent } A \text{ an instance of classical refinement (but not for } B\text{)”}\ \dagger \\
 & \quad b' := a' \oplus (a \oplus b) & \\
= & \quad a' := a; & \text{“program- and Boolean algebra”} \\
 & \quad b' := b. &
\end{array}
$$

We handle the “but not for B” by decomposing the specification to $b' := \{0, 1\}; \ldots$ instead, and we reach the same implementation. Since it *is* the same for both, we have

$$(a' \oplus b') := a \oplus b \sqsubset_{A,B} a', b' := a, b$$

for both A, B, where we have decorated the refinement symbol to indicate its applicability. Note the refinement is strict, since some nondeterminism has been removed. This reasoning is in fact what backs up the informal justification of the last step of Sect. 9.2.

We have not proved $(a' \oplus b') := a \oplus b \ \sqsubseteq_X\ a', b' := a, b$ however for some third observer X, say, from whom all variables are hidden: the step † fails since the hidden nondeterminism (as $a :\in \{0, 1\}$ is, i.e. hidden from X) may not be removed by security-enabled refinement. But it is not obvious why it should not go through: even though a', b' reveal the inputs a, b directly to A, B (which they knew anyway), agent X cannot see a', b' and so it still learns nothing about a, b.

[6] Other proper classical refinements of $(a' \oplus b') := E_A \oplus E_B$ include $a', b' := \neg E_A, \neg E_B$ and $a', b' := E_B, E_A$. In the former case the extra $\neg$'s are pointless; and the latter case would not be a *secure* refinement, since, e.g. it would reveal E_B to A.

Here thus is a "red flag" from our method: it signals an error even though we cannot see what that error might be. Should we heed the warning?

Yes, we should heed the warning. As with most catastrophes, the effect of an error can be manifested far from the error itself: indeed the "Well, that looks correct to me…" decision has probably been long forgotten, and no doubt was not documented in any case. Consider this program fragment:

$$
\begin{array}{l}
\mathbf{vis}_X\ c;\ \mathbf{vis}_A\ a'';\\
\quad [\![\ \mathbf{vis}_B\ b, b';\ \mathbf{vis}_A\ a, a';\\
\qquad a, b := 0, 0;\\
\qquad (a' \oplus b') := a \oplus b; \qquad \Leftarrow \text{Our specification of interest, from above.} \quad \ddagger\\
\qquad c := a'' \oplus a'\\
\quad]\!].
\end{array}
$$

By elementary reasoning we calculate further for this fragment:

$$
\begin{array}{lll}
= & [\![\ \mathbf{vis}_B\ b, b';\ \mathbf{vis}_A\ a, a'; & \text{"program algebra"}\\
& \quad a, b := 0, 0;\\
& \quad (a' \oplus b') := 0;\\
& \quad c := a'' \oplus a'\\
&]\!] & \\
= & [\![\ \mathbf{vis}_B\ b';\ \mathbf{vis}_A\ a'; & \text{"remove superfluous locals } a, b\text{"}\\
& \quad (a' \oplus b') := 0;\\
& \quad c := a'' \oplus a'\\
&]\!] & \\
= & [\![\ \mathbf{vis}_A\ a'; & \text{"remove superfluous local } b'\text{"}\\
& \quad a' :\in \{0, 1\};\\
& \quad c := a'' \oplus a'\\
&]\!], &
\end{array}
$$

which last is an instance of the EL: the code reveals nothing to agent X about variable a'' in spite of the assignment to c.

But now return to the start of those equalities, and replace statement ‡ by its purported refinement $a', b' := a, b$, which is the one that drew the red flag above. We then re-reason

$$
\begin{array}{lll}
& \mathbf{vis}_X\ c;\ \mathbf{vis}_A\ a'' & \\
& \quad [\![\ \mathbf{vis}_B\ b, b';\ \mathbf{vis}_A\ a, a'; & \\
& \qquad a, b := 0, 0; & \\
& \qquad a', b' := a, b; \qquad \Leftarrow \text{Looks like a refinement… but is it?} \quad \ddagger & \\
& \qquad c := a'' \oplus a' & \\
& \quad]\!] & \\
= & [\![\ c := a'' \oplus 0\ ]\!] & \text{"program algebra"}\\
= & c := a''. & \text{"program algebra"}
\end{array}
$$

This is hardly code that "reveals nothing to agent X about variable a'' in spite of the assignment to c".

The error we have just encountered has subtle- and difficult-to-explain causes, but its effect is not subtle at all: variable a'' is supposed to be A's little secret—but here we have handed it to X on a platter. That's a catastrophe for A.

The error would not have arisen had we noted the warning that our semantics gave us earlier. Such warnings are, after all, precisely what the semantics is for: it says "there is invalidating context, although you might not think of it, in which this replacement fails."[7]

A correct implementation is this one:

$$
\begin{array}{ll}
& \mathbf{vis}_A\ a, a';\ \mathbf{vis}_B\ b, b';\\
& \quad (a' \oplus b') := a \oplus b\\
\\
= & \quad a' :\in \{0,1\}; \qquad \text{"for all agents"}\\
& \quad [\![\ \mathbf{vis}_B\ b'';\ b'' := (a' \oplus a);\ b' := b'' \oplus b\ ]\!],
\end{array}
$$

so that agent A chooses its output value a' at random and sends the value $a' \oplus a$ to B, who then $\oplus$'s it with its own b to produce the output value b'. This is a valid refinement for A, B and the generic X as well.

11. Main case study: the millionaires do their sums

This, our main example, sets us apart from validation of straight-line protocols over finite state-spaces: we develop a (secure) loop; and the state-space can be arbitrarily large. Two millionaires want to find which has the bigger fortune without either revealing to the other how big their fortunes actually are. Since two-bit millionaires expose the main issues of the protocol, we will start with them—and then we generalise to "-aires" of arbitrary wealth.

11.1. Section 9 ⇒ the two-bit millionaires ($\mathbf{MP}_2$)

We compare two-bit numbers without revealing either: two integers $0 \le a, b < 4$ with $a = \langle a_1, a_0\rangle$ and $b = \langle b_1, b_0\rangle$ are given in binary, and we reveal $(2a_1 + a_0 < 2b_1 + b_0)$ by calculating $a_1 < b_1 \oplus (a_1 = b_1 \wedge a_0 < b_0)$.[8] Thus we have a formula in which only conjunctions, negations and exclusive-or appear, and the implementation is simply a stitching together of what we did earlier in Sect. 9. Its derivation is given in Appendix Appendix A; the result is

$$
\begin{array}{ll}
& \mathbf{vis}_A\ a', a_{\{0,1\}};\ \mathbf{vis}_B\ b', b_{\{0,1\}} \qquad \text{"specification"}\\
& \quad (a' \oplus b') := (2a_1 + a_0 < 2b_1 + b_0)\\
\\
\sqsubseteq & [\![\ \mathbf{vis}_A\ a_A, b_A, w_A;\ \mathbf{vis}_B\ a_B, b_B, w_B; \qquad \text{"from Appendix A"}\\
& \quad (a_A \oplus a_B) := \neg a_1 \wedge b_1; \quad \Leftarrow \text{Lovers' Protocol I.}\\
& \quad (w_A \oplus w_B) := \neg a_0 \wedge b_0; \quad \Leftarrow \text{Lovers' Protocol I.}\\
& \quad (b_A \oplus b_B) := (\neg a_1 \oplus b_1) \wedge (w_A \oplus w_B); \quad \Leftarrow \text{Lovers' Protocol II.}\\
& \quad a', b' := (a_A \oplus b_A), (a_B \oplus b_B)\ ]\!].
\end{array}
\tag{11}
$$

11.2. Sections 11.1 and Sect. 11.3 (to come) ⇒ the unbounded millionaires ($\mathbf{MP}_N$)

Now we imagine more generally that we have two N-bit numbers $a(N..0]$ and $b(N..0]$ and we want to compare them in the same oblivious way as in the two-bit case. There we moved from least- to most-significant bit: that suggests as the "effect so far" invariant that some Boolean l always indicates whether $a(n..0]$ is strictly less than $b(n..0]$ as n increases from 0 to N; obviously for security we split that l into two shares $l_{\{a,b\}}$. At the end the shares' exclusive-or gives the Boolean $a < b$ the millionaires seek; but the shares are not directly combined until then. Thus the specification is

$$
\begin{array}{ll}
& \mathbf{vis}_A\ a(N..0], l_a; \qquad \text{"specification"}\\
& \mathbf{vis}_B\ b(N..0], l_b;\\
& \quad (l_a \oplus l_b) :=\ a_{(N..0]} < b_{(N..0]}
\end{array}
\tag{12}
$$

[7] In spite of this, our (similar) step in Sect. 9.2 is not one of those invalidating contexts, and the reduction of nondeterminism is in fact valid there. It relies on the fact that the variables $r_{\{A,B\}}$ and $w_{\{A,B\}}$ occurring the *rhs* of the assignment are themselves nondeterministically set in the statement just before, and so the reduction in nondeterminism is only apparent. A detailed discussion of validities like this is given elsewhere [MMM10, Sec. 8.6].

[8] We thank Berry Schoenmakers for this suggestion of using $\oplus$ rather than $\vee$ here.

and, because of our comments above, we aim at the implementation

$$
\begin{array}{l}
\textbf{|[vis } n; \qquad \text{“implementation guess”} \\
\quad n := 0; \\
\quad (l_a \oplus l_b) := 0; \\
\quad \textbf{while } n < N \textbf{ do} \\
\quad\quad (l_a \oplus l_b) := \ a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b); \quad \Leftarrow \text{MP}_2 \text{ modified: maintains the invariant.} \\
\quad\quad n := n + 1 \\
\quad \textbf{od} \\
\textbf{]|}.
\end{array} \tag{13}
$$

11.3. Arbitrarily rich millionaires

Moving to an arbitrarily large (but still finite) state-space leads consequentially away from straight-line programs: for arbitrarily rich millionaires our comparison requires a loop. For our case we hypothesise that our **while**-loop at (13) implements the straight-line code fragment P as follows:

$$
\begin{array}{l}
\textbf{if } n < N \textbf{ then} \qquad \text{“postulated effect of loop”} \\
\quad (l_a \oplus l_b) := \ a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b); \\
\quad n := N \\
\textbf{fi}.
\end{array} \tag{14}
$$

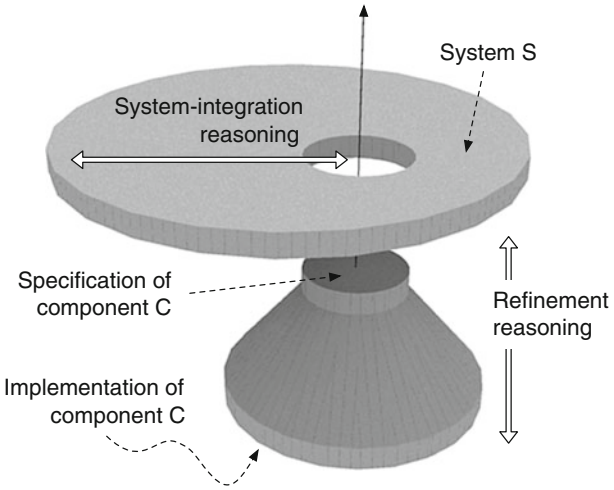
We check this program-algebraically in Appendix C. Most of the manipulations are routine (i.e. would be the same steps even if one were reasoning carefully with only functional properties in mind); but a crucial step (marked $\star$ in Appendix) uses EL to establish that the individual calculations within each iteration do not leak any information as the loop proceeds.

Thus in our proposed implementation (13) we can again rely on compositional semantics to replace the loop by its equivalent straight-line code (14). That gives

$$
\begin{array}{ll}
 & \textbf{|[vis } n; \qquad \text{“loop within (13) replaced by equivalent straight-line code (14)”} \\
 & \quad n := 0; \\
 & \quad (l_a \oplus l_b) := 0; \\
 & \quad \textbf{if } n < N \textbf{ then} \\
 & \quad\quad (l_a \oplus l_b) := \ a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b); \\
 & \quad\quad n := N \\
 & \quad \textbf{fi]|} \\
= & \textbf{|[vis } n; \qquad \text{“program algebra”} \\
 & \quad n := 0; \\
 & \quad (l_a \oplus l_b) := 0; \\
 & \quad \textbf{if } 0 < N \textbf{ then} \\
 & \quad\quad (l_a \oplus l_b) := \ a_{(N..0]} < b_{(N..0]} \oplus (a_{(N..0]} = b_{(N..0]} \wedge 0); \\
 & \quad\quad n := N \\
 & \quad \textbf{fi]|} \\
= & (l_a \oplus l_b) := 0; \qquad \text{“Eliminate local } n \text{ and simplify } \wedge 0\text{”} \\
 & \textbf{if } 0 < N \textbf{ then } \ (l_a \oplus l_b) := \ a_{(N..0]} < b_{(N..0]} \textbf{ fi} \\
= & (l_a \oplus l_b) := \ a_{(N..0]} < b_{(N..0]}, \qquad \text{“0} \le N \text{ assumed, and } a_{(0..0]} < b_{(0..0]} = 0\text{”}
\end{array}
$$

thus establishing that (12) is indeed implemented by (13).

The interior assignment of the loop (13, MP_2 modified) is based on the two-bit protocol MP_2, a small difference being that the final sub-expression is $l_a \oplus l_b$ rather than a comparison of two data-bits $a_0 < b_0$ as it was in Sect. 11.1 above. By analogy with the derivation of (11) in Appendix A, we complete our verified implementation as shown in Fig. 4, where the appeals to LP1, 2 have been expanded.



The compositionality of the security semantics is necessary for the correctness of the two types of reasoning separately, and for their mutual consistency.

Fig. 3. Horizontal- and vertical reasoning

12. Related work

The literature for security is extensive, as it is separately for refinement; however the overlap is surprisingly small.

As for many classic formulations of security, ours borrows ideas from "noninterference" [GM84] and the Logic of Knowledge [EvdMM01]. The latter is very general, allowing the system to consist of multiple interacting agents; its logical language introduces "knowledge modalities" for expressing properties such as "agent A knows a thing." Our approach optimises those desirable features to allow expressivity of security, together with the availability of source-level reasoning, a crucial component for the creation of automated proof support. Our approach is comparable to Leino [LJ00] and Sabelfeld [SS01], but differs in details (we focus on variables' final values).

Within the refinement community the methods for analysing systems are highly advanced, and have been demonstrated on numerous case studies and examples, many of them impressive for their size and complexity. However classical refinement is not sound for secrecy-style properties and therefore, without special-purpose extra formulae, developments of protocols which rely on noninterference is not possible. A selection of recent work linking refinement and security is listed below.

- Pironti and Sisto [PS08]: the Spi calculus [GA97] extends the π-calculus with extra features for designing secure protocols; in this work the refinement issue is couched in terms of a translation to Java.
- Naumann at al. [NBR07]: Naumann at al. is similar to Amtoft below, and just as relevant: here an extra logical operator encodes comparisons of several runs of the program, again at the Hoare-logic level. Again we expect to benefit by our increased emphasis on refinement.
- Amtoft and Banerjee [AB04]: Amtoft and Banerjee base their approach in program (Hoare) logic, as we do implicitly, and capture security properties via an extra judgement of *independence* between program variables. This, like our approach, is based on variables' values (unlike Mantel below), and should integrate well with classical approaches.
  Amtoft however does not emphasise refinement directly, in spite of working in the Hoare-logic setting.
- Bossi et al. [BFPR03]: this is a general approach in which criteria are developed with respect to which various relations can be classified as being refinements (or not); the framework is traces and bisimulations. It is sufficiently detailed to illustrate the interplay between nondeterminism as ignorance and nondeterminism as unpredictable behaviour (the latter being the classical view).
- Mantel [Man01]: Mantel considers the preservation of *information-flow* properties under refinement, which latter he interprets in terms of events and traces, with security then represented as possible deductions about the occurrences (or not) of high-level events. The basic theory is couched in terms of absolute security (complete absence of illegal flows) and then speculatively relaxed.

$(l_a \oplus l_b) := (a_{(N..0]} < b_{(N..0]})$ $\Leftarrow$ Exclusive-or $l_{\{a,b\}}$ finally, for the outcome $a<b$.

$\sqsubseteq$ $[\![$ **vis** n;
$n := 0$;
$(l_a \oplus l_b) := 0$;
while $n<N$ **do**
$\mathbf{vis}_A\ a_A, b_A, w_A, x_A, r_A$; $\mathbf{vis}_B\ a_B, b_B, w_B, x_B, r_B$;
$a_A :\in \{0,1\}$; $a_B := (a_n \equiv a_A \lhd b_n \rhd a_A)$;
$w_A :\in \{0,1\}$; $w_B := (l_a \equiv w_A \lhd l_b \rhd w_A)$;
$r_A :\in \{0,1\}$; $x_B := (r_A \equiv a_n \lhd w_B \rhd r_A)$;
$r_B :\in \{0,1\}$; $x_A := (r_B \oplus b_n \lhd w_A \rhd r_B)$;
} Each of these expands to six statements and four further pre-distributed bits.
$b_A, b_B := (\neg a_n \wedge w_A \oplus r_A \oplus x_A), (x_B \oplus r_B \oplus b_n \wedge w_B)$;
$l_a, l_b := (a_A \oplus b_A), (a_B \oplus b_B)$;
$n := n + 1$
od $]\!]$.

Each of the four transfers abstracts from six elementary statements, making over thirty elementary statements in all. Ten local variables are declared in the loop body, at this level. The *TTP* acts within the Oblivious Transfers, supplying four random bits for each: thus $24N$ further random bits are used in total.

Fig. 4. Millionaires: the complete code at the level of oblivious transfers

Our approach concentrates not on events (at all) but rather on variables' values, though naturally it would be possible in principle to encode one in terms of the other. We hope our advantage will be an easier integration with existing source-text-based refinement tools.

Finally we mention model checking which has been hugely successful in locating problems in published security protocols, and it continues to have a crucial input in the design process. But model checking is limited by computational time and memory, and when applied to software normally only screens for certain properties, rather than whole designs. Although new work on refinement can deal with specific properties [ML09]; our theory can provide a sound relationship between general system-wide designs (which often can be tested by model checkers) and their implementations (which sometimes cannot).

13. Conclusions

"Horizontal" reasoning across the disc of Fig. 3 (recall Sect. 9.1) uses the specification of component C to establish that it plays its proper role in the context of system S; this is done (1) without referring to the implementation of C at all. "Vertical" reasoning, down the cone, establishes that C's implementation has properties no worse than its specification; this is done (2) in isolation, without referring to any contextual system S at all. Then compositionality (3) ensures that these two separate activities (1,2) are consistent when combined. These basic features (1–3) of refinement are well known, but in each case require a semantics appropriate to the application domain: **our overall strategy** is to formulate such a semantics [Mor06, Mor09] for the noninterference-style security domain, and thus to make the rigorous development of security applications more accessible to our (refinement) community (Fig. 4).

Our specific aim in this paper, for which we chose the millionaires' problem, was to demonstrate scalability within a topical application domain (see for example the recent practical application of two-party secure computation [BCD+], and the current interest in the use of the OT as a cryptographic primitive [MNPS04].) We used both vertical reasoning (from specification to implementation of components) and horizontal reasoning (use of components' specifications only) in doing so. To our knowledge our proof here is the first (formally) for the full millionaires' problem. More generally our goal is to verify security-critical software, hence our particular focus on source-level reasoning and proofs which apply in all contexts; within those specific confines we are amongst the first to prove a (randomised) security protocol with unbounded state. Paulson [Pau98] and Coble [Cob08] also have general proofs relating to specific security properties over computations with unbounded resources.

The Shadow has been extended to deal semantically with *loops* Sect. 11.3 and syntactically with labelled *views* Sect. 4.7, the latter to enable the uniform treatment of the complementary security goals of multiple agents.

We believe that three prominent features of our approach make it suitable for practical verification: (a) secure refinement preserves (noninterference) security properties; (b) refinement is monotonic (implying compositionality); and (c) we exploit a simple source-level program algebra.

Features (a,b) allow layering of design; and (c) allows proofs to be constructed from many small (algebraic) steps, of the kind suited to automation [MCMG08]. This distinguishes us from other refinement-oriented approaches that do not so much emphasise code-level algebraic reasoning [LJ00, SS01, Man01, EvdMM01], on the one hand, or appear not to be compositional [AČZ06, Cer09], on the other.

Future work will include investigating the alternatives for automating the proofs, extending the protocols to which this style of development will apply, and expanding the semantics to allow quantitative judgements. The latter extension will enable us to move towards bridging the gap between the formal proofs required for program verification, and the primitives and their assumptions commonly used in cryptographic protocols.

Appendix A. Proof for the two-bit millionaires (Sect. 11.1)

$\mathbf{vis}_A\ a', a_{\{0,1\}};\ \mathbf{vis}_B\ b', b_{\{0,1\}}$ "specification"
$(a' \oplus b') := (2a_1 + a_0 < 2b_1 + b_0)$

$= \quad (a' \oplus b') := \quad \neg a_1 \wedge b_1 \ \oplus\ (\neg a_1 \oplus b_1 \wedge \neg a_0 \wedge b_0)$ "arithmetic"

$= \quad$ ⟦ $\mathbf{vis}_A\ a_A, b_A;\ \mathbf{vis}_B\ a_B, b_B;$ "separate *rhs* into stages using encryption lemma"
$(a_A \oplus a_B) := \neg a_1 \wedge b_1;$
$(b_A \oplus b_B) := \neg a_1 \oplus b_1 \wedge \neg a_0 \wedge b_0;$
$(a' \oplus b') := (a_A \oplus a_B) \oplus (b_A \oplus b_B)$ ⟧

$\sqsubseteq \quad$ ⟦ $\mathbf{vis}_A\ a_A, b_A;\ \mathbf{vis}_B\ a_B, b_B;$ "operands visible to A, B already"
$(a_A \oplus a_B) := \neg a_1 \wedge b_1;$
$(b_A \oplus b_B) := (\neg a_1) \oplus b_1 \wedge (\neg a_0) \wedge b_0;$
$a', b' := (a_A \oplus b_A), (a_B \oplus b_B)$ ⟧

$= \quad$ ⟦ $\mathbf{vis}_A\ a_A, b_A, w_A;\ \mathbf{vis}_B\ a_B, b_B, w_B;$ "separate further using EL"
$(a_A \oplus a_B) := \neg a_1 \wedge b_1;$ ⇐ Lovers' Protocol I.
$(w_A \oplus w_B) := \neg a_0 \wedge b_0;$ ⇐ Lovers' Protocol I.
$(b_A \oplus b_B) := (\neg a_1 \oplus b_1) \wedge (w_A \oplus w_B);$ ⇐ Lovers' Protocol II.
$a', b' := (a_A \oplus b_A), (a_B \oplus b_B)$ ⟧.

Appendix B. Derivation of oblivious transfer (Sect. 8)

The full OT can be seen as an OT, in its first phase, of random values chosen by a trusted third party. In the second phase, the protocol proper, the EL is superposed three times to yield the actual transfer.

$\mathbf{vis}_A\ m_0, m_1;$ "Oblivious Transfer specification"
$\mathbf{vis}_B\ c{:}\,\mathsf{Bool}, m;$
$m := (m_1 \lhd c \rhd m0)$

$= \quad$ ⟦ $\mathbf{vis}\ x;\ \mathbf{vis}_B\ c';$ "Encryption lemma"
$c' :\in \{0, 1\};$
$x := c \oplus c'$⟧; ⇐ Publish encrypted c as x.
$m := (m_1 \lhd c \rhd m0)$

$=$ ‖[**vis** x; **vis**$_B$ c'; “Encryption lemma twice more”
$c' :\in \{0,1\}$;
$x := c \oplus c'$]‖;
‖[**vis** y, z; **vis**$_B$ m'_0, m'_1;
$m'_0 :\in \{0,1\}$; $m'_1 :\in \{0,1\}$;
$y := m_0 \oplus m'_{\neg x}$; ⇐ Publish encrypted m_0 as y.
$z := m_1 \oplus m'_x$]‖; ⇐ Publish encrypted m_1 as z.
$m := (m_1 \lhd c \rhd m0)$

$=$ ‖[**vis** x; **vis**$_B$ c'; “Program- and Boolean algebra ; scoping”
vis y, z; **vis**$_A$ m'_0, m'_1;
$c' :\in \{0,1\}$; $x := c \oplus c'$;
$m'_0 :\in \{0,1\}$; $m'_1 :\in \{0,1\}$;
$y := m_0 \oplus m'_{\neg x}$; $z := m_1 \oplus m'_x$;
$m := (y \lhd c \rhd z) \oplus m'_{c'}$]‖ ⇐ Boolean algebra here based on assignments above.

$=$ ‖[**vis** x; **vis**$_B$ c'; “Program algebra”
vis y, z; **vis**$_A$ m'_0, m'_1;
$c' :\in \{0,1\}$; $x := c \oplus c'$;
$m'_0 :\in \{0,1\}$; $m'_1 :\in \{0,1\}$;
$y := m_0 \oplus m'_{\neg x}$; $z := m_1 \oplus m'_x$;
‖[**vis**$_B$ m';
$m' := (m'_1 \lhd c' \rhd m'_0)$; ⇐ Oblivious Transfer of random values into m'...
$m := (y \lhd c \rhd z) \oplus m'$]‖]‖ ⇐ ... used here.

$=$ ‖[**vis**$_B$ c', m'; **vis**$_A$ m'_0, m'_1; “Reorder statements; scoping.”
$c' :\in \{0,1\}$;
$m'_0 :\in \{0,1\}$; $m'_1 :\in \{0,1\}$;
$m' := (m'_1 \lhd c' \rhd m'_0)$; ⇐ This and above done in advance by TTP.

vis x, y, z; ⇐ This and below done during protocol proper.
$x := c \oplus c'$; ⇐ Published by B.
$y := m_0 \oplus m'_{\neg x}$; ⇐ Published by A.
$z := m_1 \oplus m'_x$; ⇐ Published by A.
$m := (y \lhd c \rhd z) \oplus m'$ ⇐ Decoded by B.
]‖.

Appendix C. Proof of loop equivalence (Sect. 11.3)

if $n < N$ **then** “this is **if** B **then** $body$; P **fi** ”
$(l_a \oplus l_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b)$;
$n := n + 1$;
if $n < N$ **then**
$(l_a \oplus l_b) := a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b)$;
$n := N$
fi
fi

$=$ **if** $n < N - 1$ **then** “Manipulate n and the inner **if** ”
$(l_a \oplus l_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b)$;
$(l_a \oplus l_b) := a_{(N..n+1]} < b_{(N..n+1]} \oplus (a_{(N..n+1]} = b_{(N..n+1]} \wedge l_a \oplus l_b)$;
$n := N$
else if $n = N - 1$ **then**
$(l_a \oplus l_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b)$;
$n := N$
fi

$=$ |[$\mathbf{vis}_A\ l'_a;\ \mathbf{vis}_B\ l'_b;$ "Introduce temporary variables"
  if $n < N-1$ **then**
    $(l'_a \oplus l'_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b);$
    $(l_a \oplus l_b) := a_{(N..n+1]} < b_{(N..n+1]} \oplus (a_{(N..n+1]} = b_{(N..n+1]} \wedge l'_a \oplus l'_b);$
    $n := N$
  else if $n = N-1$ **then**
    $(l_a \oplus l_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b);$
    $n := N$
  fi
]|

$=$ |[$\mathbf{vis}_A\ l'_a;\ \mathbf{vis}_B\ l'_b;$ "Binary arithmetic"
  if $n < N-1$ **then**
    $(l'_a \oplus l'_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b);$
    $(l_a \oplus l_b) := a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b);$
    $n := N$
  else if $n = N-1$ **then**
    $(l_a \oplus l_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b);$
    $n := N$
  fi
]|

$=$ |[$\mathbf{vis}_A\ l'_a;\ \mathbf{vis}_B\ l'_b;$ "**if** structure"
  if $n < N$ **then**
    if $n < N-1$ **then** $(l'_a \oplus l'_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b)$ **fi**;
    $(l_a \oplus l_b) := a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b);$
    $n := N$
  fi
]|

$=$ **if** $n < N$ **then** "manipulate scope"
  if $n < N-1$ **then**
    |[$\mathbf{vis}_A\ l'_a;\ \mathbf{vis}_B\ l'_b;$
      $(l'_a \oplus l'_b) := a_n < b_n \oplus (a_n = b_n \wedge l_a \oplus l_b)$
    ]|
  fi;
  $(l_a \oplus l_b) := a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b);$
  $n := N$
fi

$=$ **if** $n < N$ **then** "$\star$ Encryption lemma $\star$"
  if $n < N-1$ **then skip fi**;
  $(l_a \oplus l_b) := a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b);$
  $n := N$
fi

$=$ **if** $n < N$ **then** "Remove **if** $\cdots$ **skip**, noting that n is visible"
  $(l_a \oplus l_b) := a_{(N..n]} < b_{(N..n]} \oplus (a_{(N..n]} = b_{(N..n]} \wedge l_a \oplus l_b);$
  $n := N$
fi,

which establishes our hypothesis about the effect of the loop, since the last line is just P again.

References

[AB04] Amtoft T, Banerjee A (2004) Information flow analysis in logical form. In: Proceedings of the static analysis symposium. LNCS, vol 3148. Springer, Berlin (awarded *Best Paper*)

[AČZ06] Alur R, Černý P, Zdancewic S (2006) Preserving secrecy under refinement. In: ICALP '06: proceedings (part II) of the 33rd international colloquium on automata, languages and programming. Springer, Berlin, pp 107–118

[AJ94] Abramsky S, Jung A (1994) Domain theory. In: Abramsky S, Gabbay DM, Maibaum TSE (eds) Handbook of logic and computer science, vol 3. Oxford Science Publications, Oxford, pp 1–168

[BCD+] Bogetoft P, Christensen DL, Damgård I, Geisler M, Jakobsen T, Krøigaard M, Nielsen JD, Nielsen JB, Nielsen K, Pagter J, Schwartzbach M, Toft T (2010) Secure multiparty computation goes live. http://eprint.iacr.org/2008/068

[BFPR03] Bossi A, Focardi R, Piazza C, Rossi S (2003) Refinement operators and information flow security. In: SEFM, IEEE, pp 44–53

[Cer09] Cerny P (2009) Private communication

[Cob08] Coble A (2008) Formalized information-theoretic proofs of privacy using the HOL-4 theorem-prover. In: Privacy enhancing technologies symposium (to appear)

[Dij76] Dijkstra EW (1976) A discipline of programming. Prentice-Hall, Englewood Cliffs

[EvdMM01] Engelhardt K, van der Meyden R, Moses Y (2001) A refinement theory that supports reasoning about knowledge and time. In: Nieuwenhuis R, Voronkov A (eds) LPAR. Lecture notes in computer science, vol 2250. Springer, Berlin, pp 125–141

[GA97] Gordon AD, Abadi M (1997) A calculus for cryptographic protocols: the Spi calculus. In: Proceedings of the ACM conference on computer and communications security

[GM84] Goguen JA, Meseguer J (1984) Unwinding and inference control. In: Proceedings of the IEEE symposium on security and privacy. IEEE Computer Society, pp 75–86

[Hoa85] Hoare CAR (1985) A couple of novelties in the propositional calculus. Z Math Logik Grundlagen Math 31(2):173–178

[LJ00] Leino KRM, Joshi R (2000) A semantic approach to secure information flow. Sci Comput Program 37(1–3):113–138

[Man01] Mantel H (2001) Preserving information flow properties under refinement. In: Proceedings of the IEEE symposium on security and privacy, pp 78–91

[MCMG08] McIver AK, Cohen E, Morgan C, Gonzalia C (2008) Using probabilistic Kleene algebra pKA for protocol verification. J Logic Algebr Program 76(1):90–111

[ML09] Murray T, Lowe G (2009) On refinement-closed security properties and nondeterminism. In: Proceedings of the AVoCS. ENTCS 250(2):49–68

[MM08] McIver AK, Morgan CC (2008) A calculus of revelations. Presented at VSTTE theories workshop. http://www.cs.york.ac.uk/vstte08/

[MM09] McIver AK, Morgan CC (2009) Sums and lovers: case studies in security, compositionality and refinement. In: Cavalcanti A, Dams D (eds) Proceedings of the FM '09. LNCS, vol 5850. Springer, Berlin

[MM10] McIver AK, Morgan CC (2010) The thousand-and-one cryptographers. In: Jones CB, Roscoe AW, Wood KR (eds) Reflections on the work of C.A.R. Hoare. Springer, Berlin

[MMM10] McIver AK, Meinicke LA, Morgan CC (2010) Compositional closure for Bayes risk in probabilistic noninterference. In: Abramsky S, Gavoille C, Kirchner C, auf der Heide FM, Spiraki PG (eds) Proceedings of the ICALP 2010. LNCS, vol 6199, pp 223–235 (extended abstract)

[MNPS04] Malkhi D, Nisan N, Pinkas B, Sella Y (2004) Fairplay—a secure two-party computation system. In: Proceedings of the 13th conference on USENIX security symposium. USENIX Association

[Mor94] Morgan CC (1994) Programming from specifications, 2nd edn. Prentice-Hall, Englewood Cliffs. http://web.comlab.ox.ac.uk/oucl/publications/books/PfS/

[Mor06] Morgan CC (2006) The shadow knows: refinement of ignorance in sequential programs. In: Uustalu T (ed) Mathematics of program construction, vol 4014. Springer, Berlin, pp 359–378

[Mor09] Morgan CC (2009) The shadow knows: refinement of ignorance in sequential programs. Sci Comput Program 74(8):629–653

[NBR07] Naumann DA, Banerjee A, Rosenberg S (2007) Towards a logical account of declassification. In: Proceedings of the workshop on programming languages and analysis for security, pp 61–66

[Pau98] Paulson LC (1998) The inductive approach to verifying cryptographic protocols. J Comput Secur 6:85–128

[PS08] Pironti A, Sisto R (2008) Formally sound refinement of Spi calculus protocol specifications into Java code. In: IEEE high assurance systems engineering symposium

[Rab81] Rabin MO (1981) How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University. eprint.iacr.org/2005/187

[Riv99] Rivest R (1999) Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser. Technical report, M.I.T. http://theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf

[RSG+00] Ryan P, Schneider S, Goldsmith M, Lowe G, Roscoe B (2000) Modelling and analysis of security protocols. Addison-Wesley, Reading

[Sch] Schoenmakers B (2010) Cryptography lecture notes. http://www.win.tue.nl/~berry/2WC13/LectureNotes.pdf

[Smy78] Smyth MB (1978) Power domains. J Comput Syst Sci 16:23–36

[SS01] Sabelfeld A, Sands D (2001) A PER model of secure information flow. High Order Symb Comput 14(1):59–91

[Yao82] Yao AC-C (1982) Protocols for secure computations (extended abstract). In: Annual symposium on foundations of computer science (FOCS 1982). IEEE Computer Society, pp 160–164

Received 27 February 2010
Revised 10 August 2010
Accepted 22 September 2010 by Ana Cavalcanti, Dennis Dams and Marie-Claude Gaudel
Published online 10 November 2010