Sums and Lovers:
Case studies in security, compositionality and refinement

AK McIver[1]⋆ and CC Morgan[2]⋆

[1] Dept. Computer Science, Macquarie University, NSW 2109 Australia
[2] School of Comp. Sci. and Eng., Univ. New South Wales, NSW 2052 Australia

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 techniques: proof or model-checking. Informally, they are difficult or impossible to achieve.
Our formal technique is *refinement*, until recently not much applied to security. We argue its benefits by giving rigorous formal developments, in refinement-based program algebra, of several security case studies.
A conspicuous feature of our studies is their layers of abstraction and –for the main study, in particular– that the protocol is unbounded in state, placing its verification beyond the reach of model checkers.
Correctness in all contexts is crucial for our goal of layered, refinement-based developments. This is ensured by our semantics in which the program constructors are monotonic with respect to "security-aware" refinement, which is in turn a generalisation of compositionality.
Keywords: Refinement of security; formalised secrecy; hierarchical security reasoning; compositional semantics.

1 Introduction

This paper is about verifying computer programs that have security- as well as functional requirements; in particular it is about developing them in a layered, refinement-oriented way. To do that we use the novel *Shadow Semantics* [16, 17] that supports security refinement.

Security refinement is a variation of (classical) refinement that preserves non-interference properties (as well as classical, functional ones), and features compositionality and hierarchical proof with 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

⋆ We acknowledge the support of the Australian Research Council Grant DP0879529.

enough to be subjected to rigorous algorithmic checking [20]. Then secure refinement ensures that non-interference -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 is especially beneficial.

Our main contribution, in summary, is to argue that the secure-refinement paradigm [16, 17], including its compositionality and layers of abstraction, can greatly extend the size and complexity of security applications that can be verified. The principal example is Yao's Millionaraires' Protocol [24], especially suitable because it includes four (sub-) protocols nested like dolls within it: our paradigm allows them to be treated separately, so that they 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 §3 we set out the semantics for secure refinement; and in §4 we begin our series of case studies, in increasing order of complexity; but before any of that, in §2 we introduce multi-party computations. 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 $z, z{+}1, z{+}4, \cdots$ that exceed z by a perfect square exactly.

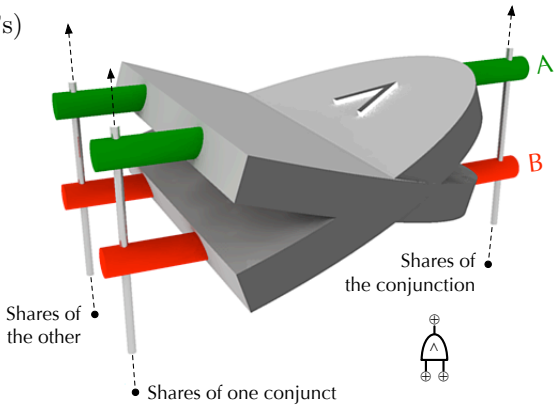
In the conclusions §8 we set out our **strategic goals** for the whole approach.

2 Secure multi-party computation: an overview

Multi-party computations (*MPC*'s) are protocols in which separate agents hold separate subsets of values in a shared computation [24]. The computation is *secure* if it reveals only what can be deduced about the agents' separate values once they are combined at the very end. We specify a typical two-party *MPC* as

$$\mathbf{vis}_A\ a; \mathbf{vis}_B\ b; \mathbf{vis}\ x; \quad x{:=}\, a \otimes b\ , \tag{1}$$

in which two agents A and B, with their respective variables a and b visible only to them



Agent A sees the upper (green) shares; Agent B sees the lower (red). Unless they combine their shares with exclusive-or ⊕, neither agent infer anything about the other's values.

Fig. 1. *Two-party ⊕-shared conjunction: §6.2*

separately, somehow collaboratively calculate $a \otimes b$ and publish it in a variable x; but they reveal nothing (more) about a, b in the process, either to each other, or 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) 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 indeed 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 §5ff; and it is not self-evident at all.

An *unsatisfactory* implementation of (1) involves 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. Yao [24] avoids the $rTTP$ by appealing to the sub-protocol *Oblivious Transfer* [18, 19] 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.

Our main case study is Yao's spectacular illustration of his own technique, the millionaires A, B of §7 who compare their fortunes a, b without revealing either: only the Boolean $a<b$ is published. For us too it is a showcase exemplar, because it uses the Lovers' II protocol (§6.2), using Lovers' I (§6.1), using Oblivious Transfer (§5), using the Encryption Lemma (§4); moreover our treatment of the main loop (§7.3, unbounded riches) abstracts from the loop body (§7.1, 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 non-interference -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, paramount 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. And it is only the second example (to our knowledge) of proof of an unbounded protocol [5, 13, 14], differing from other approaches which typically impose resource restrictions. Finally, the proofs are lengthy but, crucially, they are boring — they comprise many tiny steps similar to those already automated in probabilistic program algebra [12], and are thus easily checked.

3 The Shadow Model of security and refinement

The Shadow Model extends the *non-interference model* of security [7] to determine an attacker's inferred knowledge of hidden (high-security) variables at each point in the computation; we then require that the inferred knowledge *is not increased* by secure refinement.

In its original form. non-interference partitions variables into high-security- and low-security classes: we call them *hidden* and *visible*. A "non-interference -secure" program is then one where our 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 r thus takes initial states (v, h) to sets of final visible states v' and is of type $\mathcal{V} \to \mathcal{H} \to \mathbb{P}\mathcal{V}$, where $\mathcal{V}, \mathcal{H}$ are the value sets corresponding to the types of v, h. Such a program r is *non-interference -secure* just when for any initial visible the set of possible final visibles is independent of the initial hidden [9, 21], that is for any $v{:}\,\mathcal{V}$ we have $\big(\forall h_0, h_1{:}\,\mathcal{H} \cdot r.v.h_0 = r.v.h_1\big)$.

In our approach [16] we extend this view, in several stages. The first is to concentrate on final- (rather than initial) hidden values and therefore to model programs as $\mathcal{V}{\to}\mathcal{H}{\to}\mathbb{P}(\mathcal{V}{\times}\mathcal{H})$. For two such programs $r_{\{1,2\}}$ we say that $r_1 \sqsubseteq r_2$, that r_1 "is securely refined by" r_2, whenever both the following hold:

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

(ii) For all $v{:}\,\mathcal{V}, h{:}\,\mathcal{H}$ and $v'{:}\,\mathcal{V}$ satisfying $\big(\exists h'_2{:}\,\mathcal{H} \cdot (v', h'_2) \in r_2.v.h\big)$, we have that $(v', h') \in r_1.v.h$ implies $(v', h') \in r_2.v.h$ for all $h'{:}\,\mathcal{H}$.
This second condition says that for any observed visibles v, v' and any initial h the attacker's "deductive powers" w.r.t. final h''s cannot be improved by refinement: there can only be more possibilities, never fewer.

In this simple setting the two conditions together do not yet allow an attacker's ignorance of h strictly to increase: refinement boils down to allowing decrease of nondeterminism in v but not in h. But strict increase of hidden nondeterminism is illustrated later, in §3.3.

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.

3.1 The Shadow H of h records h's inferred values

In r_1 above the set of possible final values of h' was $\{0, 1\}$ for each v' separately. This set is called "The Shadow," and represents explicitly an attacker's ignorance

of h': it is the smallest set of possibilities he can infer. In r_2 that shadow was the same; but in r_3 the shadow was smaller, just $\{v'\}$ for each v', and that is why r_3 was not a secure refinement of r_1.

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

$$\begin{array}{ll} r_1 — & \{(0,0,\{0,1\}),\ (0,1,\{0,1\}),\ (1,0,\{0,1\}),\ (1,1,\{0,1\})\} \\ r_2 — & \{(0,0,\{0,1\}),\ (0,1,\{0,1\})\} \\ r_3 — & \{(0,0,\{0\}),\ (1,1,\{1\})\}\ , \end{array}$$

and we have $r_1 \sqsubseteq r_2$ because r_1's set of outcomes includes all of r_2's. But for r_3 we find that its outcome $(0,0,\{0\})$ does not occur among r_1's outcomes, nor is there even an r_1-outcome $(0,0,H')$ with $H' \subseteq \{0\}$ that would satisfy (ii). That, again, is why $r_1 \not\sqsubseteq r_3$.

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

3.2 Step 1: The Shadow Semantics of atomic programs

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

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

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

□

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 standard program that chooses variable x's value from a non-empty set S. Assume here only that S is constant, not depending on v, h. Then from Def. 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) but choosing h affects both h *and* H, introducing ignorance because
$$\mathsf{addShadow}.(h :\in S).v.h.H = \{h' : S \cdot (v, h', S)\}$$

(iii) and an assignment of hidden to visible "collapses" ignorance because

$$\mathsf{addShadow}.(v{:=}h).v.h.H = \{(h,h,\{h\})\} \quad .$$

From (ii) and (iii) the composition $\mathsf{addShadow}.(h{:\in}S);\mathsf{addShadow}.(v{:=}h)$ 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 as a whole is nondeterministic, yielding $\{x{:}\,S \cdot (x,x,\{x\})\}$ with v,h's common final value x drawn arbitrarily from S; but whatever that value is, it is known that h has it because H is a singleton.

3.3 Step 2: The Shadow Semantics of straight-line programs

Shadow enhancement is extended to general programs by induction over their syntax, as shown in Fig. 2. The only non-traditional command is **reveal** that publishes an expression but changes no program variables; note it does change the shadow.

3.4 Step 3: Refinement's properties via Gedanken Experiments

We choose our definition of refinement based on scale-up experiments with program algebra. Our first observation is that the semantics enforces *perfect recall*, that visible variables reveal information even if subsequently overwritten. 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 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 –imperfect recall– is false.

There is a similar gedanken experiment for conditionals: because (A,B) makes

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

true, 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 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. With those ideas in place, we define refinement as follows:

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

$$\begin{array}{l}(\forall\,(v',h',H_2'){:}\,[\![P_2]\!].v.h.H \cdot \\ \qquad \big(\exists H_1'{:}\,\mathbb{P}\mathcal{H} \mid H_1' \subseteq H_2' \cdot \quad (v',h',H_1') \in [\![P_1]\!].v.h.H\big)\ )\ ,\end{array}$$

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

The syntactically atomic commands A marked $\star$ have the property that $A = \langle\!\langle A \rangle\!\rangle$. This is deliberate: syntactic atoms execute atomically. The function $\mathsf{lift}.[\![P_2]\!]$ applies $[\![P_2]\!]$ to all triples in its set-valued argument, un-Currying each time, and then takes the union of all results.
The extension to many variables $v_1, v_2, \cdots$ and $h_1, h_2, \cdots$, including local declarations, is straightforward [16, 17].

Fig. 2. Semantics of non-looping commands

with $[\![\cdot]\!]$ as defined in Fig. 2.

This means that for each initial triple (v, h, H) every final triple (v', h', H_2') produced by P_2 must be “justified” by the existence of a triple (v', h', H_1'), with equal or smaller 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 $\{(v,0,\{0\}),(v,1,\{1\})\}$ whereas $[\![h{:\in}\,\{0,1\}]\!].v.h.H$ is $\{(v,0,\{0,1\}),(v,1,\{0,1\})\}$. This is thus an example of a refinement where the two commands differ only by a strict 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 $\{0\} \subset \{0,1\}$.

3.5 Step 4: 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. Atomicity is not generally preserved by composition; but in simple cases it is.

Lemma 1. *atomicity and composition* Given two programs $P_{\{1,2\}}$ over v, h we have $\langle\!\langle P_1; P_2 \rangle\!\rangle = \langle\!\langle P_1 \rangle\!\rangle; \langle\!\langle P_2 \rangle\!\rangle$ just when v's *intermediate* value, i.e. “at the semicolon,” can be deduced from its *endpoint* values, i.e. initial and final, possibly in combination. The semicolon is interpreted classically on the left, and as in Fig. 2 on the right.

Proof: Given in [1, App. A]. □

This lemma is more significant when its conditions are *not* met than when they are. It means for example that we cannot conclude from Lem. 1 that $\langle\!\langle v{:=}\,h; v{:=}\,0\rangle\!\rangle = \langle\!\langle v{:=}\,h\rangle\!\rangle; \langle\!\langle v{:=}\,0\rangle\!\rangle$, 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 Def. 1

(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 again is a case of perfect recall. More interesting is the utility of introducing atomicity temporarily in a derivation, as illustrated in §4 below: when applicable, we can infer security properties via (simpler) classical reasoning.

3.6 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 simple semantics reflects A's viewpoint, say, by interpreting variables declared to be $\mathbf{vis}_{list}$ 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.
- $\mathbf{vis}_{list}$ means the associated variable is visible to all agents in the (non-empty) list, and is hidden from all others (including third parties).
- $\mathbf{hid}_{list}$ means the associated variable is hidden from all agents in the list, and is visible to all others (including third parties).

For example 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 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 of successful attack is a special case of ours: if P reveals nothing, then a successful attack is one in which Q is forced to reveal anything at all — because $P \sqsubseteq Q$ then means that also Q must reveal nothing.

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 First case study: the Encryption Lemma (*EL*)

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

$$
\begin{array}{lll}
 & [\![\ \mathbf{vis}\ v; \mathbf{hid}\ h'{\cdot}\ v{\oplus}h'{:=}\,h\ ]\!] & \\
= & [\![\ \mathbf{vis}\ v; \mathbf{hid}\ h'{\cdot}\ \langle\!\langle v, h'{:}[v{\oplus}h' = h]\rangle\!\rangle\ ]\!] & \text{“defined above”} \\
= & [\![\ \mathbf{vis}\ v; \mathbf{hid}\ h'{\cdot}\ \langle\!\langle v{:}\in\{0,1\}; h'{:=}\,h{\oplus}v\rangle\!\rangle\ ]\!] & \text{“classical reasoning within } \langle\!\langle\cdot\rangle\!\rangle\text{”} \\
= & [\![\ \mathbf{vis}\ v; \mathbf{hid}\ h'{\cdot}\ \langle\!\langle v{:}\in\{0,1\}\rangle\!\rangle; \langle\!\langle h'{:=}\,h{\oplus}v\rangle\!\rangle\ ]\!] & \text{“Lem. 1”} \\
= & [\![\ \mathbf{vis}\ v; \mathbf{hid}\ h'{\cdot}\ v{:}\in\{0,1\}; h'{:=}\,h{\oplus}v\ ]\!] & \text{“syntactic atoms”} \\
= & [\![\ \mathbf{vis}\ v{\cdot}\ v{:}\in\{0,1\}; [\![\ \mathbf{hid}\ h'{\cdot}\ h'{:=}\,h{\oplus}v\ ]\!]\ ]\!] & \text{“}h'\text{ not free”}\ \heartsuit \\
= & [\![\ \mathbf{vis}\ v{\cdot}\ v{:}\in\{0,1\}\ ]\!] & \text{“assignment to local hidden is \textbf{skip}”}\ \heartsuit \\
= & \mathbf{skip}\ , & \text{“assignment of visibles to local visible is \textbf{skip}”}\ \heartsuit
\end{array}
$$

where at $\heartsuit$ we appeal to manipulations of scope, and more primitive **skip**-equivalences, that because of space we must justify elsewhere [16, 17].

Each step can be justified by the semantics of §3, and the overall chain of equalities establishes our Encryption Lemma: we will see it often.

5 Second case study: §4⇒ Oblivious Transfer (*OT*)

The *Oblivious Transfer Protocol* builds on §4: 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 [18]; we use Rivest's specialisation of it [19]. Its specification is

$$
\begin{array}{ll}
\mathbf{vis}_A\ m_0, m_1; & \text{“Oblivious Transfer specification”} \\
\mathbf{vis}_B\ c{:}\,\mathsf{Bool}, m; & \\
\quad m := (m_1 \lhd c \rhd m_0)\ , & \Leftarrow \text{We write } (\textit{left} \text{ if } \textit{condition} \text{ else } \textit{right}) \text{ [8]}
\end{array}
$$

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. We give a complete derivation elsewhere [17], and it relies on the Encryption Lemma of §4.

In brief (and only 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 given in [1, App. C].

6 Third case study: The Lovers' Protocols

The Lovers' Protocols (see for example "Dating without embarrassment" [22]) in this section are 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.

6.1 §5⇒ Lovers' Protocol I (*LP1*)

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 knows anything more about $a \wedge b$ as a result of knowing its own a' or 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 elide the justifications for third parties.) We have

$$\begin{array}{lll}
& \mathbf{vis}_A\ a, a'; \mathbf{vis}_B\ b, b'; \quad \Leftarrow \text{Global variables: assumed below.} & \text{"specification"} \\
& \quad (a' \oplus b') := \ a \wedge b & \\
& & \\
= & \quad \langle\!\langle\ a' :\in \{0,1\};\ \ b' := (a \wedge b) \oplus a'\ \rangle\!\rangle & \text{"atomicity reasoning: compare } \mathit{EL}\text{"} \\
= & \quad a' :\in \{0,1\};\ \ b' := (a \wedge b) \oplus a' & \text{"Lem. 1: compare } \mathit{EL}\text{"} \\
= & \quad a' :\in \{0,1\};\ \ b' := (a \lhd b \rhd 0) \oplus a' & \text{"Boolean algebra: } \mathbf{true} \text{ is 1, } \mathbf{false} \text{ is 0"} \\
& & \\
= & \quad a' :\in \{0,1\}; & \text{"Boolean algebra"} \\
& \quad b' := (a \oplus a' \lhd b \rhd a')\ . \quad \Leftarrow \text{Implemented by } \textit{Oblivious Transfer.} &
\end{array}$$

Our semantics §3 plays two roles here, in the background: it legitimises the manipulations immediately above that introduced OT into the implementation, which in §8 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.

6.2 §4, §6.1⇒ Lovers' Protocol II (*LP2*) 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 *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

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

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

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

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

$=$ $\quad [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B;$ "Program- and Boolean algebra"
$(r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A;$
$(a' \oplus b') := a_A \wedge b_A \oplus r_A \oplus w_A \quad \oplus \quad w_B \oplus r_B \oplus a_B \wedge b_B\ ]\!]$

$\sqsubseteq$ $\quad [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B;$ "see below"
$(r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A;$ $\quad \Leftarrow$ Implemented by *LP1*.
$a' := a_A \wedge b_A \ \oplus\ r_A \oplus w_A;$
$b' := w_B \oplus r_B \ \oplus\ a_B \wedge b_B\ ]\!]$.

In the last step we have broken the *rhs* into two operands; and revealing them consequentially to A, B separately is justified by those recipients' knowing the operands' values already. Note that it is a proper refinement. [3]

7 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 — then we generalise to "-aires" of arbitrary wealth.

[3] 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.

7.1 §6⇒ The two-bit Millionaires (MP_2)

We compare a pair of 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 $a_1{<}b_1 \oplus (a_1{=}b_1 \wedge a_0{<}b_0)$.[4] 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 have done already in §6. Its derivation is given in [1, App. B]; the result is

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

7.2 §7.1⇒The unbounded Millionaires (MP_N): overview

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, and that suggests as an invariant that some Boolean l 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 millionaries seek; but they are not directly combined until then.

Thus the specification is

$$
\begin{array}{ll}
& \mathbf{vis}_A\ a(N..0], l_a; \qquad \text{“specification”} \\
& \mathbf{vis}_B\ b(N..0], l_b; \\
& \quad (l_a \oplus l_b) := \ a_{(N..0]} < b_{(N..0]}
\end{array}
\tag{3}
$$

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

$$
\begin{array}{ll}
& [\![\ \mathbf{vis}\ n; \qquad \text{“implementation guess”} \\
& \quad n := 0; \\
& \quad (l_a \oplus l_b) := 0; \\
& \quad \mathbf{while}\ n{<}N\ \mathbf{do} \\
& \quad\quad (l_a \oplus l_b) := \ a_n{<}b_n \oplus (a_n{=}b_n \wedge l_a{\oplus}l_b); \quad \Leftarrow MP_2 \text{ modified.} \\
& \quad\quad n := n + 1 \\
& \quad \mathbf{od} \\
&]\!] \ .
\end{array}
\tag{4}
$$

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

7.3 How do we deal with loops?

Moving to an unbounded state-space takes us consequentially away from straight-line programs: for arbitrarily rich millionaires our comparison requires a loop. We extend our semantics with fixed points in the usual way: thus a terminating loop **while** B **do** *body* **od** equals some P just when via secure program algebra we can manipulate **if** B **then** *body*;P **fi** to become P again. For our case we hypothesise that our **while**-loop at (4) implements the straight-line code P as follows:

$$\begin{array}{l}\textbf{if } n{<}N \textbf{ then} \qquad\qquad \text{“postulated effect}\\ \quad (l_a \oplus l_b){:=}\ a_{(N..n]}{<}b_{(N..n]} \oplus (a_{(N..n]}{=}b_{(N..n]} \wedge l_a \oplus l_b); \quad \text{of loop”}\\ \quad n{:=}N \qquad\qquad (5)\\ \textbf{fi}\ .\end{array}$$

We check this in [1, App. D]. 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 the 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 (4) we can again rely on compositional semantics to replace the loop by its equivalent straight-line code (5). That gives

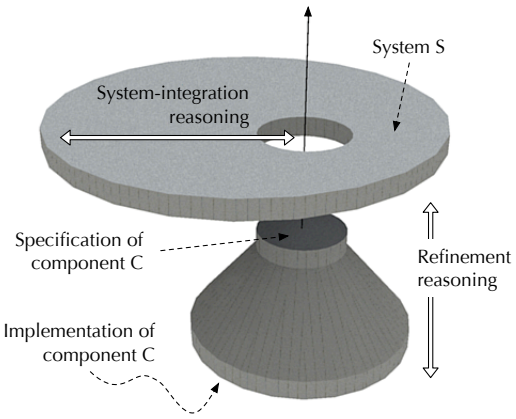
$$\begin{array}{ll}& [\![\ \textbf{vis } n; \qquad\qquad \text{“loop within (4) replaced by}\\ & \quad n{:=}0; \qquad\qquad \text{equivalent straight-line code (5)”}\\ & \quad (l_a \oplus l_b){:=}0;\\ & \quad \textbf{if } n{<}N \textbf{ then}\\ & \qquad (l_a \oplus l_b){:=}\ a_{(N..n]}{<}b_{(N..n]} \oplus (a_{(N..n]}{=}b_{(N..n]} \wedge l_a \oplus l_b);\\ & \qquad n{:=}N\\ & \quad \textbf{fi}\ ]\!]\\ =& [\![\ \textbf{vis } n; \qquad\qquad \text{“program algebra”}\\ & \quad n{:=}0;\\ & \quad (l_a \oplus l_b){:=}0;\\ & \quad \textbf{if } 0{<}N \textbf{ then}\\ & \qquad (l_a \oplus l_b){:=}\ a_{(N..0]}{<}b_{(N..0]} \oplus (a_{(N..0]}{=}b_{(N..0]} \wedge 0);\\ & \qquad n{:=}N\\ & \quad \textbf{fi}\ ]\!]\\ =& (l_a \oplus l_b){:=}0; \qquad\qquad \text{“Eliminate local } n\\ & \quad \textbf{if } 0{<}N \textbf{ then } (l_a \oplus l_b){:=}\ a_{(N..0]} < b_{(N..0]} \textbf{ fi} \qquad\qquad \text{and simplify } \wedge 0\text{”}\\ =& (l_a \oplus l_b){:=}\ a_{(N..0]} < b_{(N..0]}\ , \qquad\qquad \text{“}0{\le}N \text{ assumed, and } a_{(0..0]}{<}b_{(0..0]} = 0\text{”}\end{array}$$

thus establishing that (3) is indeed implemented by (4).

The interior assignment of the loop (4) is based on the two-bit protocol MP_2, with a small difference being that the final sub-expression is $l_a{\oplus}l_b$ rather than a

comparison of two data-bits as it was above. By analogy with the derivation of (2) in [1, App. B], we complete our verified implementation as shown in Fig. 4, where the appeals to *LP1,2* have been expanded.

8 Conclusions and strategy



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*

"Horizontal" reasoning across the disc of Fig. 3 (recall §6.1) refers to 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. Compositionality (3) ensures that these two separate activities (1,2) are consistent when combined. These basic features (1,2,3) 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 [16, 17] for the non-interference -style security domain, and thus to make the rigorous development of security applications more accessible to our (refinement) community.

Our aim in this paper, for which we chose the Millionaires' problem, was to demonstrate the scalability of our approach within a topical application domain. (See for example the recent practical application of two-party secure computation [4], and the current interest in the use of the oblivious transfer as a cryptographic primitive [10].) We demonstrated both vertical reasoning (within sections) and horizontal reasoning (use of previous sections) in doing so. To our knowledge our proof here is the first (formally) for the full Millionaires' problem, and it is only the second for *any* (randomised) security protocol with unbounded state. (The first was the generalised Dining Cryptographers, proved by Coble [5] and independently by us [13, 14].)

The Shadow Semantics [16, 17] is our basis, but here it has been extended to deal with *loops* and *views*, the latter to enable the uniform treatment of the complementary security goals in a multiparty system. The relationship to other formal semantics of non-intereference has been summarised in detail elsewhere [*loc. cit.*]; it is comparable to Leino [9] and Sabelfeld [21], but differs in details; and it shares the goals of the pioneering work of Mantel [11] and Engelhardt [6].

$(l_a \oplus l_b) := (a_{(N..0]} < b_{(N..0]})$ $\Leftarrow$ 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**
vis$_A$ a_A, b_A, w_A, x_A, r_A; **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 needed 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.

We believe that three prominent features of our approach make it suitable for practical verification: (a) refinement preserves (non-interference) security properties; (b) refinement is compositional; 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 [12]. This distinguishes us from other refinement-oriented approaches that do not emphasise code-level algebraic reasoning [9, 21, 11, 6], on the one hand, or appear not to be compositional [3, 2], on the other.

Our plans include constructing/extending computer-based tools to prove the small algebraic steps, based on theorem-proving over the Shadow semantics, and thus to form a library of allowed transformations. At the same time we hope to integrate Shadow-style reasoning, based on such a library, into industrial-strength refinement-based developments [23].

References

1. Appendices are available at `www.cse.unsw.edu.au/~carrollm/probs/bibliographyBody.html#McIver:09`.
2. Pavol Černý: private communication, February 2009.
3. Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *Proceedings of Automata, Languages and Programming*, number 4052 in LNCS, pages 107–118. Springer, 2006.

4. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. Available at `eprint.iacr.org/2008/068`.
5. Aaron Coble. Formalized information-theoretic proofs of privacy using the HOL-4 theorem-prover. In *Privacy Enhancing Technologies Symp.*, 2008. To appear.
6. K. Engelhardt, R. van der Meyden, and Y. Moses. A refinement theory that supports reasoning about knowledge and time. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 125–41. Springer, 2001.
7. J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc IEEE Symp on Security and Privacy*, pages 75–86, 1984.
8. C.A.R. Hoare. A couple of novelties in the propositional calculus. *Zeitschr für Math Logik und Grundlagen der Math.*, 31(2):173–8, 1985.
9. K.R.M. Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–38, 2000.
10. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — A secure two-party computation system. In *Proc. 13th Conf. on USENIX Security Symposium*. USENIX Association, 2004.
11. Heiko Mantel. Preserving information flow properties under refinement. In *Proc IEEE Symp Security and Privacy*, pages 78–91, 2001.
12. A.K. McIver, E. Cohen, C. Morgan, and C. Gonzalia. Using probabilistic Kleene algebra pKA for protocol verification. *Journal of Logic and Algebraic Programming*, 76(1):90–111, 2008.
13. A.K. McIver and C.C. Morgan. A calculus of revelations, October 2008. Presented at `//www.cs.york.ac.uk/vstte08/`.
14. Annabelle McIver and Carroll Morgan. The thousand-and-one cryptographers. In *Festschrift in Honour of Tony Hoare*, 2009. To appear. *Note to reviewers: This overlaps only in §3, our summary of previous work [16, 17].*
15. C.C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994. `web.comlab.ox.ac.uk/oucl/publications/books/PfS/`.
16. C.C. Morgan. *The Shadow Knows:* Refinement of ignorance in sequential programs. In T. Uustalu, editor, *Math Prog Construction*, volume 4014 of *Springer*, pages 359–78. Springer, 2006.
17. C.C. Morgan. *The Shadow Knows:* Refinement of ignorance in sequential programs. *Science of Computer Programming*, 74(8), 2009.
18. Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University, 1981. Available at `eprint.iacr.org`.
19. R. Rivest. Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser. Technical report, M.I.T., 1999. `//theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf`.
20. Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2000.
21. A. Sabelfeld and D. Sands. A PER model of secure information flow. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
22. Berry Schoenmakers. Crytography lecture notes.
Available at `www.win.tue.nl/~berry/2WC13/LectureNotes.pdf`.
23. `//www.deploy-project.eu`.
24. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Annual Symposium on Foundations of Computer Science (FOCS 1982)*, pages 160–164, 1982.

A Proof of Lem. 1 from §3.5

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

$$\langle\!\langle r_1; r_2 \rangle\!\rangle.v.h.H \ni (v', h', H')$$

iff
$$\begin{aligned} & (r_1; r_2).v.h \ni (v', h') && \text{“Def. 1”} \\ \text{and}\quad & H' = \{h'' \colon H, h' \colon \mathcal{H} \mid (r_1; r_2).v.h'' \ni (v', h') \cdot h'\} \end{aligned}$$

iff
$$\begin{aligned} & \big(\exists v^\flat \colon \mathcal{V}, h^\flat \colon \mathcal{H} \mid r_1.v.h \ni (v^\flat, h^\flat) \wedge r_2.v^\flat.h^\flat \ni (v', h')\big) && \text{“forward relational composition”} \\ \text{and}\quad & H' = \{h'' \colon H, h' \colon \mathcal{H}, v^\sharp \colon \mathcal{V}, h^\sharp \colon \mathcal{H} \\ & \qquad\qquad \mid r_1.v.h'' \ni (v^\sharp, h^\sharp) \wedge r_2.v^\sharp.h^\sharp \ni (v', h') \\ & \qquad\qquad \cdot h'\} \end{aligned}$$

On the other hand, with piecewise atomicity we have

$$(\langle\!\langle r_1 \rangle\!\rangle; \langle\!\langle r_2 \rangle\!\rangle).v.h.H \ni (v', h', H')$$

iff “forward relational composition”
$$\begin{aligned} & (\exists v^\natural \colon \mathcal{V}, h^\natural \colon \mathcal{H}, H^\natural \colon \mathbb{P}\mathcal{H} \mid \\ & \qquad \langle\!\langle r_1 \rangle\!\rangle.v.h.H \ni (v^\natural, h^\natural, H^\natural) \wedge \langle\!\langle r_2 \rangle\!\rangle.v^\natural.h^\natural.H^\natural \ni (v', h', H')) \end{aligned}$$

iff “Def. 1”
$$\begin{aligned} & (\exists v^\natural \colon \mathcal{V}, h^\natural \colon \mathcal{H}, H^\natural \colon \mathbb{P}\mathcal{H} \mid \\ & \qquad r_1.v.h \ni (v^\natural, h^\natural) \\ & \qquad \wedge H^\natural = \{h^+ \colon \mathcal{H}, h^- \colon H \mid r_1.v.h^- \ni (v^\natural, h^+) \cdot h^+\} \\ & \qquad \wedge r_2.v^\natural.h^\natural \ni (v', h') \\ & \qquad \wedge H' = \{h^+ \colon \mathcal{H}, h^- \colon H^\natural \mid r_2.v^\natural.h^- \ni (v', h^+) \cdot h^+\}\,) \end{aligned}$$

iff “propositional calculus; eliminate $H^\natural$”
$$\begin{aligned} & (\exists v^\natural \colon \mathcal{V}, h^\natural \colon \mathcal{H} \mid \\ & \qquad r_1.v.h \ni (v^\natural, h^\natural) \wedge r_2.v^\natural.h^\natural \ni (v', h') \\ & \qquad \wedge H' = \{h^+ \colon \mathcal{H}, h^- \colon \{h^+ \colon \mathcal{H}, h^- \colon H \mid r_1.v.h^- \ni (v^\natural, h^+) \cdot h^+\} \quad) \\ & \qquad\qquad\qquad\qquad \mid r_2.v^\natural.h^- \ni (v', h^+) \\ & \qquad\qquad\qquad\qquad \cdot h^+\} \end{aligned}$$

iff “rename bound variables”
$$\begin{aligned} & (\exists v^\natural \colon \mathcal{V}, h^\natural \colon \mathcal{H} \mid \\ & \qquad r_1.v.h \ni (v^\natural, h^\natural) \wedge r_2.v^\natural.h^\natural \ni (v', h') \\ & \qquad \wedge H' = \{h^+ \colon \mathcal{H}, h^- \colon \{h^\pm \colon \mathcal{H}, h^\mp \colon H \mid r_1.v.h^\mp \ni (v^\natural, h^\pm) \cdot h^\pm\} \quad) \\ & \qquad\qquad\qquad\qquad \mid r_2.v^\natural.h^- \ni (v', h^+) \\ & \qquad\qquad\qquad\qquad \cdot h^+\} \end{aligned}$$

iff “collapse comprehensions”

$$
\begin{array}{l}
(\exists v^\natural\colon \mathcal{V}, h^\natural\colon \mathcal{H} \mid \\
\qquad r_1.v.h \ni (v^\natural, h^\natural) \wedge r_2.v^\natural.h^\natural \ni (v', h') \\
\quad \wedge\ H' = \{h^+\colon \mathcal{H}, h^\pm\colon \mathcal{H}, h^\mp\colon H \\
\qquad\qquad\qquad \mid r_1.v.h^\mp \ni (v^\natural, h^\pm) \wedge r_2.v^\natural.h^\pm \ni (v', h^+) \\
\qquad\qquad\qquad \cdot\ h^+\}
\end{array}
\qquad)
$$

iff “rename bound variables”

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

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

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

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

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

iff “introduce D”

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

And for the second case we can continue

iff “introduce D”

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

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

That completes the proof of Lem. 1.

B Proof for the two-bit millionaires (§7.1)

$\mathbf{vis}_A\ a', a_{\{0,1\}};\ \mathbf{vis}_B\ b', b_{\{0,1\}}$ “specification”

$(a' \oplus b') := (2a_1 + a_0 < 2b_1 + b_0)$

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

$=$ $[\![$ $\mathbf{vis}_A\ a_A, b_A;\ \mathbf{vis}_B\ a_B, b_B;$ "separate *rhs* into stages
$(a_A \oplus a_B) := \neg a_1 \wedge b_1;$ using Encryption Lemma"
$(b_A \oplus b_B) := \neg a_1 \oplus b_1 \wedge \neg a_0 \wedge b_0;$
$(a' \oplus b') := (a_A \oplus a_B) \oplus (b_A \oplus b_B)$ $]\!]$

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

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

C Derivation of Oblivious Transfer (§5)

The full Oblivious Transfer can be seen as an oblivious transfer, in an advanced first phase, of random values chosen by a Trusted Third Party. In the second phase, the protocol proper, the Encryption Lemma is superposed three times to yield the actual transfer.

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

$=$ $[\![$ $\mathbf{vis}\ x; \mathbf{vis}_B\ c';$ "Encryption Lemma"
$c' :\in \{0,1\};$
$x := c \oplus c'$ $]\!];$ $\Leftarrow$ Publish encrypted c as x.
$m := (m_1 \lhd c \rhd m0)$

$=$ $[\![$ $\mathbf{vis}\ x; \mathbf{vis}_B\ c';$ "Encryption Lemma twice more"
$c' :\in \{0,1\};$
$x := c \oplus c'$ $]\!];$
$[\![$ $\mathbf{vis}\ y, z; \mathbf{vis}_B\ m'_0, m'_1;$
$m'_0 :\in \{0,1\};\ m'_1 :\in \{0,1\};$
$y := m_0 \oplus m'_{\neg x};$ $\Leftarrow$ Publish encrypted m_0 as y.
$z := m_1 \oplus m'_x$ $]\!];$ $\Leftarrow$ 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;
$c'{:\in}\{0,1\}$; $x{:=}\, c \oplus c'$;
$m'_0{:\in}\{0,1\}$; $m'_1{:\in}\{0,1\}$;
$y{:=}\, m_0 \oplus m'_{\neg x}$; $z{:=}\, m_1 \oplus m'_x$;
$m := (y \lhd c \rhd z) \oplus m'_{c'}$ $]\!]$ $\Leftarrow$ 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';
$m'{:=}(m'_1 \lhd c' \rhd m'_0)$; $\Leftarrow$ Oblivious Transfer of random values into m'...
$m := (y \lhd c \rhd z) \oplus m'$ $]\!]\!]$ $\Leftarrow$ ... used here.

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

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

D Proof of loop equivalence (§7.3)

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.