Sums and Lovers: Case Studies in Security, Compositionality and Refinement

Annabelle K. McIver[1,⋆] and Carroll C. 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 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* [14,15] 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 enough to be subjected to rigorous algorithmic checking [19]. Then secure refinement

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

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 property is especially beneficial.

Our main contribution, in summary, is to argue by example that the secure-refinement paradigm [14,15], including its compositionality and layers of abstraction, can greatly extend the size and complexity of security applications that can be verified. The principal case study is a protocol for Yao's Millionaraires' Problem [23], especially suitable 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 §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 $(\mathsf{Q}x{:}\,T|R{\bullet}E)$ 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 "$(\mathsf{Q}$" 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.

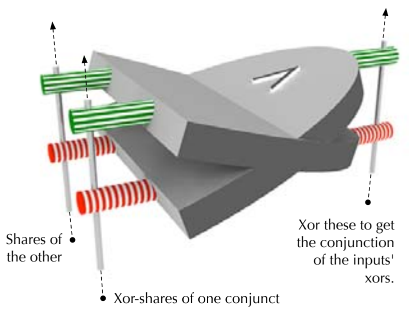
In the conclusions §8 we set out our **strategic goals** for the whole approach.

2 Secure Multi-party Computation: An Overview

In *Multi-party computations* (*MPC*'s) separate agents hold their own shares of a shared computation, as illustrated in Fig. 1. Only at the end are the shares combined; and the computation is *secure* if no information is released until that point. 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 separately, *somehow collaboratively calculate* the result $a{\otimes}b$ and publish it in the variable



∧
Shares of the other
Xor these to get the conjunction of the inputs' xors.
Xor-shares of one conjunct

Agent A sees the upper shares, the two inputs and one output; B sees the lower . 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: §6.2

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) 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. The *rTTP* can be avoided by appealing to the sub-protocol *Oblivious Transfer* [17,18] 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 millionaires A, B of §7 who compare their fortunes 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 (§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, 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 [12], and 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 [9,20], that is for any $v\colon \mathcal{V}$ we have $\left(\forall h_0, h_1\colon \mathcal{H} \cdot r.v.h_0 = r.v.h_1\right)$.

In our approach [14] 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 $v\colon \mathcal{V}$ and $h\colon \mathcal{H}$ we have $r_1.v.h \supseteq r_2.v.h$.
This is the classical "can reduce nondeterminism" form of refinement.

(ii) For all $v\colon \mathcal{V}, h\colon \mathcal{H}$, and $v'\colon \mathcal{V}$ satisfying $\left(\exists h_2'\colon \mathcal{H} \cdot (v', h_2') \in r_2.v.h\right)$, we have for all $h'\colon \mathcal{H}$ 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''s 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 §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 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

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

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 $r{:}\,\mathcal{V}\rightarrow\mathcal{H}\rightarrow\mathbb{P}(\mathcal{V}\times\mathcal{H})$ we define its *shadow enhancement* $\mathsf{addShadow}.r$ of type $\mathcal{V}\rightarrow\mathcal{H}\rightarrow\mathbb{P}\mathcal{H}\rightarrow\mathbb{P}(\mathcal{V}\times\mathcal{H}\times\mathbb{P}\mathcal{H})$ so that $\mathsf{addShadow}.r.v.h.H \ni (v',h',H')$ just when both

$$\begin{array}{lll} \text{(i)} & r.v.h \ni (v',h') & \text{— } \textit{classical} \\ \text{(ii) and} & H' = \{h'{:}\,\mathcal{H} \mid \left(\exists h''{:}\,H \cdot r.v.h'' \ni (v',h')\right)\}\ . & \text{— } \textit{shadow} \end{array}$$

□

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

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 (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 is, it is known that h has it because H is a singleton.

3.3 Step 2: The Shadow Semantics of Straight-Line Programs

General (non-atomic) programs gain their shadows by repeated application of §3.2 as implied 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

Our definition of refinement is based on scale-up experiments with program algebra [14,15]. 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.

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

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

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 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

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

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 $\{(v,0,\{0\}),(v,1,\{1\})\}$, 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 $\{0\} \subset \{0,1\}$.

	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	$\langle\!\langle P \rangle\!\rangle$	$\mathsf{addShadow}$.("classical semantics of P")	
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**	$[\![P_t]\!].v.h.\{h'\!: H \mid E.v.h' = \mathbf{true}\}$	
	We write *if* $\lhd$ *cond* $\rhd$ *else* [8] $\longrightarrow$	$\lhd\ E.v.h\ \rhd$	
		$[\![P_f]\!].v.h.\{h'\!: H \mid E.v.h' = \mathbf{false}\}$	

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 [14,15].

Fig. 2. Semantics of non-looping commands

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, and thus classical *equality* (although not classical refinement) can be used within them. In simple cases atomicity is preserved by composition, but not in general:

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 as significant when its conditions are *not* met as 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 is perfect recall again. 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 equalities within $\langle\!\langle\cdot\rangle\!\rangle$.

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 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 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 [13]. 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 non-interference -style protocols is the idiom $[\![\ \mathbf{vis}\ v; \mathbf{hid}\ h' \cdot\ (v \oplus h') {:=} h\ ]\!]$ 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 $[\![\cdots]\!]$. 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

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

where at $\heartsuit$ we appeal to manipulations of scope, and more primitive **skip**-equivalences, that because of space we must justify elsewhere [14,15]. That is, 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 [17]; we use Rivest's specialisation of it [18]. 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. A complete derivation is published elsewhere [15], and it relies on the Encryption Lemma of §4.

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 [1, App. C].

6 Third Case Study: The Lovers' Protocols

The Lovers' Protocols (see for example "Dating without embarrassment" [21]) 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 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

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

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

$$\begin{array}{lll}
& \mathbf{vis}_A\ a', a_A, b_A; \quad \Leftarrow \text{These globals assumed below.} & \text{“specification”} \\
& \mathbf{vis}_B\ b', a_B, b_B; & \\
& \quad (a' \oplus b') := (a_A \oplus a_B) \wedge (b_A \oplus b_B) & \\[1em]
= & (a' \oplus b') := a_A \wedge b_A \ \oplus\ a_A \wedge b_B \ \oplus\ a_B \wedge b_A \ \oplus\ a_B \wedge b_B & \text{“Boolean algebra”} \\[1em]
= & [\![\ \mathbf{vis}_A\ r_A;\ \mathbf{vis}_B\ w_B; & \text{“}\mathit{EL}\text{ for } A\text{, and for } B \text{ (different visibilities),} \\
& \quad (r_A \oplus w_B) := a_A \wedge b_B; & \text{where } h \text{ is the expression } a_A \wedge b_B; \\
& \quad (a' \oplus b') := a_A \wedge b_A \ \oplus\ a_A \wedge b_B \ \oplus\ a_B \wedge b_A \ \oplus\ a_B \wedge b_B\ ]\!] & \text{then scope”} \\[1em]
= & [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B; & \text{“}\mathit{EL}\text{ for } A\text{, and for } B; \\
& \quad (r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A; & \text{then scope”} \\
& \quad (a' \oplus b') := a_A \wedge b_A \ \oplus\ a_A \wedge b_B \ \oplus\ a_B \wedge b_A \ \oplus\ a_B \wedge b_B\ ]\!] & \\[1em]
= & [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B; & \text{“Program- and Boolean algebra”} \\
& \quad (r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A; & \\
& \quad (a' \oplus b') := a_A \wedge b_A \oplus r_A \oplus w_A \quad \oplus \quad w_B \oplus r_B \oplus a_B \wedge b_B\ ]\!] & \\[1em]
\sqsubseteq & [\![\ \mathbf{vis}_A\ r_A, w_A;\ \mathbf{vis}_B\ r_B, w_B; & \text{“see below”} \\
& \quad (r_A \oplus w_B) := a_A \wedge b_B;\ (r_B \oplus w_A) := a_B \wedge b_A; \quad \Leftarrow \text{Implemented by } \mathit{LP1}. & \\
& \quad a' := a_A \wedge b_A \ \oplus\ r_A \oplus w_A; & \\
& \quad b' := w_B \oplus r_B \ \oplus\ a_B \wedge b_B\ ]\!]\ . &
\end{array}$$

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.[1]

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 — and then we generalise to “-aires” of arbitrary wealth.

[1] 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 $0 \leq a, b < 4$ 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 $a_1{<}b_1 \oplus (a_1{=}b_1 \wedge a_0{<}b_0)$.[2] 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 §6. Its derivation is given in [1, App. B]; the result is

```
  vis_A $a', a_{\{0,1\}}$;  vis_B $b', b_{\{0,1\}}$                      "specification"
     $(a' \oplus b') {:=} (2a_1{+}a_0 < 2b_1{+}b_0)$

$\sqsubseteq$  $[\![$ vis_A $a_A, b_A, w_A$;  vis_B $a_B, b_B, w_B$;              "from [1, App. B]"
        $(a_A \oplus a_B) {:=} \neg a_1 \wedge b_1$;    $\Leftarrow$ Lovers' Protocol I.            (2)
        $(w_A \oplus w_B) {:=} \neg a_0 \wedge b_0$;    $\Leftarrow$ Lovers' Protocol I.
        $(b_A \oplus b_B) {:=} (\neg a_1 \oplus b_1) \wedge (w_A \oplus w_B)$;    $\Leftarrow$ Lovers' Protocol II.
        $a', b' {:=} (a_A \oplus b_A), (a_B \oplus b_B)$  $]\!]$ .
```

7.2 §7.1, §7.3 (to Come)⇒ The Unbounded Millionaires (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 millionaries seek; but the shares are not directly combined until then. Thus the specification is

```
  vis_A $a(N..0], l_a$;                                              "specification"
  vis_B $b(N..0], l_b$;                                                               (3)
     $(l_a \oplus l_b) {:=}\ a_{(N..0]} < b_{(N..0]}$
```

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

```
   $[\![$ vis $n$;                                              "implementation guess"
      $n {:=} 0$;
      $(l_a \oplus l_b) {:=} 0$;
      while $n{<}N$ do
         $(l_a \oplus l_b) {:=}\ a_n{<}b_n \oplus (a_n{=}b_n \wedge l_a{\oplus}l_b)$;    $\Leftarrow$ $MP_2$ modified:      (4)
         $n {:=} n + 1$                                              maintains the invariant.
      od
   $]\!]$ .
```

[2] 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 leads 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 other program fragment 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 fragment P as follows:

```
if $n{<}N$ then                                                              "postulated effect
    $(l_a \oplus l_b) := a_{(N..n]} {<} b_{(N..n]} \oplus (a_{(N..n]} {=} b_{(N..n]} \wedge l_a \oplus l_b)$;     of loop"
    $n := N$                                                                     (5)
fi .
```

We check this program-algebraically 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

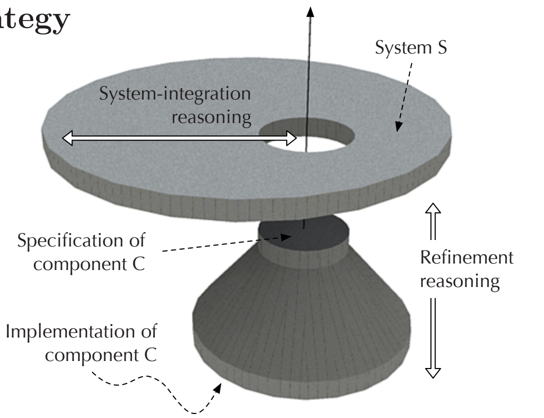
```
    ⟦ vis $n$;                                                  "loop within (4) replaced by
      $n := 0$;                                                  equivalent straight-line code (5)"
      $(l_a \oplus l_b) := 0$;
      if $n{<}N$ then
         $(l_a \oplus l_b) := a_{(N..n]} {<} b_{(N..n]} \oplus (a_{(N..n]} {=} b_{(N..n]} \wedge l_a \oplus l_b)$;
         $n := N$
      fi ⟧

=   ⟦ vis $n$;                                                            "program algebra"
      $n := 0$;
      $(l_a \oplus l_b) := 0$;
      if $0{<}N$ then
         $(l_a \oplus l_b) := a_{(N..0]} {<} b_{(N..0]} \oplus (a_{(N..0]} {=} b_{(N..0]} \wedge 0)$;
         $n := N$
      fi ⟧

=   $(l_a \oplus l_b) := 0$;                                                 "Eliminate local $n$
    if $0{<}N$ then $(l_a \oplus l_b) := a_{(N..0]} < b_{(N..0]}$ fi              and simplify $\wedge 0$"

=   $(l_a \oplus l_b) := a_{(N..0]} < b_{(N..0]}$ ,          "$0 \leq N$ assumed, and $a_{(0..0]} {<} b_{(0..0]} = 0$"
```

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

The interior assignment of the loop (4, 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 §7.1 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


System S
System-integration reasoning
Specification of component C
Refinement reasoning
Implementation of component C

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) 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 (1,2,3) 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 [14,15] for the non-interference -style security domain, and thus to make the rigorous development of security applications more accessible to our (refinement) community.

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 [4], and the current interest in the use of the oblivious transfer as a cryptographic primitive [10].) 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 [16] and Coble [5] also have general proofs relating to specific security properties over computations with unbounded resources.

The Shadow has been extended to deal semantically with *loops* §7.3 and syntactically with labelled *views* §3.6, the latter to enable the uniform treatment of the complementary security goals of multiple agents. The relationship to other

```
          (l_a ⊕ l_b):= (a_(N..0] < b_(N..0])   ⇐ Exclusive-or l_{a,b} finally, for the outcome a<b.

⊑    [[ vis n;
        n:= 0;
        (l_a ⊕ 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:∈ {0,1};  a_B:= (a_n≡a_A ◁ b_n ▷ a_A);    }
            w_A:∈ {0,1};  w_B:= (l_a≡w_A ◁ l_b ▷ w_A);    }  Each of these expands to six statements
            r_A:∈ {0,1};  x_B:= (r_A≡a_n ◁ w_B ▷ r_A);    }  and four further pre-distributed bits.
            r_B:∈ {0,1};  x_A:= (r_B⊕b_n ◁ w_A ▷ r_B);    }
            b_A, b_B:= (¬a_n∧w_A ⊕ r_A ⊕ x_A), (x_B ⊕ r_B ⊕ b_n∧w_B);
            l_a, l_b:= (a_A ⊕ b_A), (a_B ⊕ 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

formal semantics of non-intereference has been summarised in detail elsewhere [14,15]; it is comparable to Leino [9] and Sabelfeld [20], but differs in details; and it shares the goals of the pioneering work of Mantel [11] and Engelhardt [6].

We believe that three prominent features of our approach make it suitable for practical verification: (a) secure refinement preserves (non-interference) 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 [12]. This distinguishes us from other refinement-oriented approaches that do not so much emphasise code-level algebraic reasoning [9,20,11,6], 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 [22].

References

1. Appendices are available at, `www.cse.unsw.edu.au/~carrollm/probs/bibliographyBody.html#McIver:09`
2. Černý, P.: Private communication (February 2009)

3. Alur, R., Černý, P., Zdancewic, S.: Preserving secrecy under refinement. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 107–118. Springer, Heidelberg (2006)
4. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Secure multiparty computation goes live, http://eprint.iacr.org/2008/068
5. Coble, A.: Formalized information-theoretic proofs of privacy using the HOL-4 theorem-prover. In: Borisov, N., Goldberg, I. (eds.) PETS 2008. LNCS, vol. 5134, pp. 77–98. Springer, Heidelberg (2008)
6. Engelhardt, K., van der Meyden, R., Moses, Y.: A refinement theory that supports reasoning about knowledge and time. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 125–141. Springer, Heidelberg (2001)
7. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proc. IEEE Symp. on Security and Privacy, pp. 75–86 (1984)
8. Hoare, C.A.R.: A couple of novelties in the propositional calculus. Zeitschr für Math. Logik und Grundlagen der Math. 31(2), 173–178 (1985)
9. Leino, K.R.M., Joshi, R.: A semantic approach to secure information flow. Science of Computer Programming 37(1–3), 113–138 (2000)
10. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — A secure two-party computation system. In: Proc. 13th Conf. on USENIX Security Symposium. USENIX Association (2004)
11. Mantel, H.: Preserving information flow properties under refinement. In: Proc. IEEE Symp. Security and Privacy, pp. 78–91 (2001)
12. McIver, A.K., Cohen, E., Morgan, C., Gonzalia, C.: Using probabilistic Kleene algebra pKA for protocol verification. Journal of Logic and Algebraic Programming 76(1), 90–111 (2008)
13. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall, Englewood Cliffs (1994), web.comlab.ox.ac.uk/oucl/publications/books/PfS/
14. Morgan, C.C.: The Shadow Knows: Refinement of ignorance in sequential programs. In: Uustalu, T. (ed.) Math. Prog. Construction. LNCS, vol. 4014, pp. 359–378. Springer, Heidelberg (2006) Treats Dining Cryptographers
15. Morgan, C.C.: The Shadow Knows: Refinement of ignorance in sequential programs. Science of Computer Programming 74(8) (2009) Treats Oblivious Transfer
16. Paulson, L.: Proving properties of security protocols by induction, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-409.pdf
17. Rabin, M.O.: How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University (1981), http://eprint.iacr.org/2005/187
18. Rivest, R.: Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser. Technical report, M.I.T (1999), http://theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf
19. Ryan, P., Schneider, S., Goldsmith, M., Lowe, G., Roscoe, B.: Modelling and Analysis of Security Protocols. Addison-Wesley, Reading (2000)
20. Sabelfeld, A., Sands, D.: A PER model of secure information flow. Higher-Order and Symbolic Computation 14(1), 59–91 (2001)
21. Schoenmakers, B.: Cryptography lecture notes, http://www.win.tue.nl/~berry/2WC13/LectureNotes.pdf
22. http://www.deploy-project.eu
23. Yao, A.C.-C.: Protocols for secure computations (extended abstract). In: Annual Symposium on Foundations of Computer Science (FOCS 1982), pp. 160–164 (1982)