DOI 10.1007/s00165-012-0237-4
BCS © 2012
Formal Aspects of Computing (2014) 26: 169–194

Formal Aspects of Computing

Abstractions of non-interference security: probabilistic versus possibilistic

T. S. Hoang[1], A. K. McIver[2], L. Meinicke[2], C. C. Morgan[3], A. Sloane[2] and E. Susatyo[2]

[1] Institute of Information Security, ETH Zurich, Zurich, Switzerland
[2] Department of Computing, Macquarie University, Sydney, Australia
[3] School of Computer Science and Engineering, UNSW, Sydney, Australia

Abstract. The Shadow Semantics (Morgan, Math Prog Construction, vol 4014, pp 359–378, 2006; Morgan, Sci Comput Program 74(8):629–653, 2009) is a possibilistic (qualitative) model for noninterference security. Subsequent work (McIver et al., Proceedings of the 37th international colloquium conference on Automata, languages and programming: Part II, 2010) presents a similar but more general *quantitative* model that treats probabilistic information flow. Whilst the latter provides a framework to reason about quantitative security risks, that extra detail entails a significant overhead in the verification effort needed to achieve it. Our first contribution in this paper is to study the relationship between those two models (qualitative and quantitative) in order to understand when qualitative Shadow proofs can be "promoted" to quantitative versions, i.e. in a probabilistic context. In particular we identify a subset of the Shadow's refinement theorems that, when interpreted in the quantitative model, still remain valid even in a context where a passive adversary may perform probabilistic analysis. To illustrate our technique we show how a semantic analysis together with a syntactic restriction on the protocol description, can be used so that purely qualitative reasoning can nevertheless verify probabilistic refinements for an important class of security protocols. We demonstrate the semantic analysis by implementing the Shadow semantics in Rodin, using its special-purpose refinement provers to generate (and discharge) the required proof obligations (Abrial et al., STTT 12(6):447–466, 2010). We apply the technique to some small examples based on secure multi-party computations.

Keywords: Non-interference security, Probabilistic non-interference, Program semantics, Program refinement

Correspondence and offprint requests to: A. K. McIver, E-mail: annabelle.mciver@mq.edu.au
We acknowledge the support of the Australian Research Council Grant DP0879529.

1. Introduction

The *Shadow Semantics* [Mor06] was developed to support reasoning about noninterference-style properties for sequential programs. It adopts the traditional *noninterference* separation [GM84] into high- and low-security variables, with the usual assumption that an adversary may observe directly the low-security (or visible) variables but can only infer the value of the high-security (or hidden) variables consistent with those observations. *Shadow Refinement* then ensures that implementations of specifications preserve (or improve upon) functionality without revealing any more about the hidden variables than the specification does. This permits stepwise development of secure programs in the same way that normal refinement methods, such as the B method [Abr96] or the Refinement Calculus [BvW98, Mor94, Mor87], permit the development of security-insensitive programs.

The quantitative extension of the Shadow Semantics [MMM10] is a generalisation that allows probabilistic assignment to variables; it models the attacker's knowledge using a probability distribution over the possible hidden states (rather than simply a set of those states) to capture any residual uncertainty consistent with observed events. This expands the attacker's armoury to include Bayesian-style learning for computing the most likely hidden value over the possibilities identified by the Shadow Semantics, thus increasing his chance of breaching the program's security defences. A compositional notion of "Entropy Refinement" was introduced [MMM10] and was shown to ensure that if "specification S is refined by I" then the chance that an attacker can guess the value of hidden variables after executing implementation I cannot exceed that chance after executing S.

Whilst the extended probabilistic semantics gives a more detailed profile of how well the hidden state is concealed, the original semantics is preferable from a practical viewpoint: this is largely because the simpler Boolean-based semantics is easier to automate.

Moreover, in working through a number of benchmark examples from the literature we discovered that, for a certain class of protocol, purely qualitative reasoning is actually sufficient to guarantee probabilistic results provided the initialisation of the probabilistic variables is uniform. This observation raised the attractive possibility of using (simpler) qualitative techniques to obtain (more robust) quantitative guarantees. One of the outcomes of this paper is to explore the extent to which this is possible. To this end we investigate the semantic relationship between a quantitative version of the Shadow semantics and its original qualitative version. Our results show that although the original Shadow is strictly weaker, we are able to impose an additional language restriction for which full probabilistic guarantees can be proven with only qualitative reasoning.

As a proof of concept we automate our proofs in the standard Rodin modelling environment [ABH+10]. This together with our language restriction implies that for an important class of protocols using our chosen cryptographic primitives, with only a little more work, the detailed probabilistic assessment of a program's security is within reach of automated verification.

Our first contribution is to show that for security *specifications* which declassify some information, the Shadow Semantics is no less informative than the quantitative probabilistic semantics under certain circumstances. Such specifications include a number of well-known cryptographic primitives, including the Oblivious Transfer [Rab81], the Dining Cryptographers [Cha88] and Multi-Party Computation [Yao82]. From a cryptographer's perspective, protocol implementations should be as robust against statistical attack as their specifications are. Our formal analysis reduces that problem to a consistency check determined by our semantic models. Complementing the simple consistency requirement is a requirement that the likelihood of the hidden value remains uniform over its set of possibilities, so that Bayesian learning cannot distinguish between them. For this we identify a subset of the language that is expressive enough for many protocols, and is guaranteed to preserve the uniformity requirement. We discuss the scope and limitations of our language restriction in Sect. 8.

Our second contribution is to provide automation for a part of the verification process that our analysis identifies, namely the consistency checking of the Shadow Semantics. We use the Rodin modelling and refinement engine to encode the Shadow Semantics, and we illustrate it on some non-trivial case studies. To our knowledge these are the first automated proofs of non-interference security that use a comparative style of verification supported by a secrecy-preserving refinement relation. The effort of automation presented a number of challenges which we report here. The first was in the encoding of the explicit updates of the "shadow variables" that track the possibilities over which the secret can range. This requirement creates a modelling burden within Rodin itself but it can be mitigated by a front-end translator that automatically creates the Shadow variables and the glueing invariants in a uniform way. The second challenge was to find a modelling style which accurately reflected the synchronous nature of our protocols within Rodin's asynchronous execution model to ensure that information is not released inadvertently.

We begin in Sect. 2 by recalling the probabilistic Shadow Semantics [MMM10], and we illustrate it in Sect. 2.7 with a probabilistic specification of a commitment protocol. Next in Sect. 3 we recall the original Shadow Semantics, and in Sect. 4 we study the relationship between the two, showing in Theorem 4.1 that a Shadow Semantics equivalence implies a probabilistic semantic equivalence in a context of a restricted programming language. Finally in Sect. 5 we show how to automate Shadow refinement, and we illustrate it on two case studies: *Perfect Information Retrieval* Sect. 6.1 and the *Dining Cryptographers* Sect. 6.2, for which we previously had performed only hand-verifications. Since both also comply with our language restrictions, we have therefore in fact given an automated probabilistic analysis for these protocols.

We discuss related work in Sect. 7 and propose some future research directions in Sect. 8.

2. A probabilistic, noninterference sequential semantics

In this section we briefly review our model for specifying quantitative noninterference security, together with a quantitative refinement relation between programs which ensures that implementations are at least as secure as their specifications, in a sense we define.

2.1. A brief motivation and informal summary

We focus on *visible* variables (low-security), typically named v and of some finite type $\mathcal{V}$, and *hidden* variables (high-security), typically h in finite $\mathcal{H}$. Variables within program texts are in $\mathsf{sans\ serif}$ to distinguish them from (perhaps decorated) values $v{:}\,\mathcal{V}$, $h{:}\,\mathcal{H}$ they might contain.[1]

A simple example is given by the following short program consisting of an assignment to a hidden variable h that is initialised uniformly over possible values $\{0, 1, 2\}$, followed by an assignment to a visible variable v that leaks the value of h's low-order bit:

$$\begin{array}{l} \textbf{hid}\ \mathsf{h};\ \ \textbf{vis}\ \mathsf{v}; \\ \quad \mathsf{h}{:}{\in}\ \{\!\{0, 1, 2\}\!\}; \\ \quad \mathsf{v}{:=}\ \mathsf{h}\ \mathbf{mod}\ 2. \end{array} \tag{1}$$

In the assignment to h we see an instance of the notation "$\mathsf{h}{:}{\in}\ \{\!\{\cdots\}\!\}$" where we mean a selection made uniformly at random from the elements between the double braces. An example of a security concern for this program is the extent to which information about the hidden value h is leaked by the assignment to the visible variable v.

Our attack model is based on a passive, but curious attacker whose role is to gather as much information about the values of the hidden variables as he can. We place the following limits on his observational abilities: he can observe the values of the visible variables directly as the program executes (perfect recall), and he may observe program branches taken (implicit flow), together with a knowledge of the static program text, to make deductions about the possible values of the hidden variables. He cannot however observe the hidden variables directly and, since he is working within a probabilistic environment, his knowledge will usually never be absolute but instead will be measured as a probability distribution over likely values. Although at some point he might know that h is uniformly distributed over three values, as after the first assignment above at (1) for example, that knowledge could change with his subsequent run-time monitoring of the visible state. In the case above, the attacker will revise his knowledge about h once he observes the value assigned to v. His reasoning would run as follows.

The original assignment to h is selected uniformly over its possible values—he knows this because he has a copy of the program text. However after the final assignment to v, if he observes that $\mathsf{v}=1$, then he deduces that therefore h must certainly be 1, because of the three possible values for h only "1" is consistent with $\mathsf{v} = \mathsf{h}\ \mathbf{mod}\ 2$. On the other hand if $\mathsf{v}=2$ finally then the attacker deduces that h is assigned either 0 or 2, with probability $1/2$ each way. These two observable outcomes (i.e. $\mathsf{v}=0$ or $\mathsf{v}=1$) themselves occur with probabilities $1/3$ and $2/3$

[1] We say hidden and visible, rather than high- and low security, because of the connection with data refinement where the same technical issues of visibility occur; but there they have no implications for security.

respectively, and they can be used to quantify the average amount of information leaked. We discuss how to do that below.

From our description above it becomes clear that each concrete observation of v can be paired with a residual uncertainty of h, one described by a probability distribution δ over the type $\mathcal{H}$; we call a pair (v, δ) a "split-state" because it splits the values of variables v from h and represents them differently. In fact the full probabilistic behaviour of executing program (1), i.e. its outcome, can now be summarised as a "probability distribution over split-states" assigned by

$$\begin{array}{ll} \mathrm{v} = 0 \wedge \mathrm{h} \in \{\!\{0, 2\}\!\} & @\, 2/3 \\ \mathrm{v} = 1 \wedge \mathrm{h} \in \{\!\{1\}\!\} & @\, 1/3\,, \end{array} \tag{2}$$

where we are writing "@" for "with probability."

In general, for a type X we write $\mathbb{D}X$ for the set of discrete distributions over X and, using this, we can formalise the information summarised at (2) as an element of type $\mathbb{D}(\mathcal{V}\times\mathbb{D}\mathcal{H})$, two-level structures that we will call *hyperdistributions*. Hyperdistributions form the basis of our programming language semantics, as set out below, so that if hyperdistribution $\Delta \in \mathbb{D}(\mathcal{V}\times\mathbb{D}\mathcal{H})$ is a result of executing a program, then observable (v, δ) in the support of Δ (thus of type $\mathcal{V} \times \mathbb{D}\mathcal{H}$) occurs with the probability that Δ assigns to (v, δ). When some (v, δ) is observed, the attacker's residual uncertainty of the possible values of h, consistent with his observations and the program text, is then given by δ. Indeed the distribution δ is the conditional distribution of *h given* the observations made by the attacker within the limits of our attack model, and a hyperdistribution expresses exactly the a posteriori distribution determined by such conditioning.

Of particular interest is how to compare hyperdistributions for their relative ability to hide the value of h from the attacker. A popular measure for analysing information leakage is "Bayes Vulnerability," [Smi07] defined to be the probability that the attacker will have guessed correctly if he chooses the most likely hidden value. For the hyperdistribution above at (2) the Bayes Vulnerability is 2/3: if $\mathsf{v} = 1$ then the attacker knows that h is 1 also, and so when he guesses 1 he is guaranteed to be right. If however v is 0 then he knows only that h is distributed uniformly over $\{0, 2\}$ and he will therefore guess 0 (or 2), being right only half the time. The overall Bayes Vulnerability is the weighted average of the two outcomes, giving overall $1/3 \times 1 + 2/3 \times 1/2 = 2/3$.

Thus if programs P, P' produce hyperdistributions Δ, Δ' respectively with corresponding Bayes Vulnerabilities $\mathsf{bv}(\Delta)$, $\mathsf{bv}(\Delta')$ say, then from $\mathsf{bv}(\Delta) \geq \mathsf{bv}(\Delta')$ we would conclude that P' is more secure than P, since the attacker's best guess succeeds for P' on average no more than it does for P.

However attacks are rarely based on a program run in isolation, but rather when embedded as a "component" within a larger project. Hence we will define our secure refinement relation between programs by insisting that P is said to be refined by P' only if *within all contexts* $\mathcal{C}(\cdot)$ defined by our programming language (given below) the Bayes Vulnerability of hyperdistributions produced by $\mathcal{C}(P)$ is no less than of those produced by $\mathcal{C}(P')$. This means we are interested in the *compositional closure* of bv-comparisons [MMM10].

Other candidates for measuring the "strength of security" include (Conditional) Shannon Entropy [Sha48], Guessing Entropy, or Marginal Guesswork [KB07], but it is not yet known whether or not they determine a well-defined partial order on $\mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})$ as Bayes Vulnerability does.[2]

In the following sections we fill in the technical details of this summary, giving the full probabilistic semantics of a programming language together with the definition of Entropy Refinement. We illustrate the definitions with a small case study based on an abstraction of a commitment scheme.

We begin by introducing the *distribution* notation, generalising the notations for naïve set theory: this is helpful because the usual notations for conditional/a-posteriori distributions (e.g. [GW86]) do not suit our denotational presentations very well.

[2] It is also known that the partial order based on Bayes Vulnerability is the weakest such entropy-based order [MMM10].

2.2. Notations for distributions: explicit, implicit and expected values

In what follows, operators without their operands are written between parentheses, as $(\preceq)$ for example. *Set* comprehensions are written as $\{s\colon S \mid G \bullet E\}$ meaning the set formed by instantiating bound variable s in the expression E over those elements of S satisfying formula G.[3]

By $\mathbb{D}S$ we mean the set of *discrete distributions* on S that sum to one.[4] The *support* $\lceil\delta\rceil$ of a distribution $\delta\colon\mathbb{D}S$ is then simply the subset of elements s in S with $\delta.s \neq 0$.

Here are our notations for *explicit* distributions (cf. set enumerations):

multiple We write $\{\!\{x^{@p}, y^{@q}, \cdots, z^{@r}\}\!\}$ for the distribution assigning probabilities $p, q, \cdots, r$ to elements $x, y, \cdots, z$ respectively, with $p{+}q{+}\cdots{+}r = 1$.

uniform When explicit probabilities are omitted they are uniform: thus $\{\!\{x\}\!\}$ is the point distribution $\{\!\{x^{@1}\}\!\}$, and $\{\!\{x, y, z\}\!\}$ is $\{\!\{x^{@\frac{1}{3}}, y^{@\frac{1}{3}}, z^{@\frac{1}{3}}\}\!\}$.

In general, we write $(\mathcal{E}\ d\colon\delta \bullet E)$ for the *expected value* $(\sum d\colon\lceil\delta\rceil \bullet \delta(d) \times E)$ of expression E interpreted as a random variable in d over distribution δ. If however E is Boolean, then it is taken to be 1 if E holds and 0 otherwise: thus in that case $(\mathcal{E}\ d\colon\delta \bullet E)$ is the combined probability in δ of all elements d that satisfy E. In some cases we write $[E]$ to indicate the Boolean-to-0/1 conversion explicitly.

We write *implicit* distributions (cf. set comprehensions) as $\{\!\{d\colon\delta \mid R \bullet E\}\!\}$, for distribution δ, real expression R and expression E, meaning

$$(\mathcal{E}\ d\colon\delta \bullet R \times \{\!\{E\}\!\}) \,/\, (\mathcal{E}\ d\colon\delta \bullet R) \tag{3}$$

where, first, an expected value is formed in the numerator by scaling and adding point-distribution $\{\!\{E\}\!\}$ as a real-valued function: this gives another distribution. The scalar denominator then normalises to give a distribution yet again. A missing E is implicitly d itself.

Thus $\{\!\{d\colon\delta \bullet E\}\!\}$ *maps* expression E in d over distribution δ to make a new distribution on E's type. When R is present, and Boolean, it is converted to 0,1; thus in that case $\{\!\{d\colon\delta \mid R\}\!\}$ is δ's *conditioning* over formula R as predicate on d.

Finally, for *Bayesian belief revision*—which is needed to compute updates to the hyperdistributions—we let δ be an a priori distribution over some support D, and we let expression R for each d in D be the probability of a certain subsequent result if that d is chosen. Then $\{\!\{d\colon\delta \mid R\}\!\}$ is the a-posteriori distribution over D when that result actually occurs.

To see these definitions at work, consider again the program at (1). The original assignment to h is uniform over the three possible values {0, 1, 2} forming the a-priori distribution over h; after the assignment to v we can calculate the a posteriori distributions based on the possible results. In the case that v's final value is 0, that gives

$$\begin{aligned} &= (\mathcal{E}\ h\colon\{\!\{0,1,2\}\!\} \bullet [0 = h \mathbin{\mathbf{mod}} 2] \times \{\!\{h\}\!\}) \,/\, (\mathcal{E}\ h\colon\{\!\{0,1,2\}\!\} \bullet [0 = h \mathbin{\mathbf{mod}} 2]) \\ &= \{\!\{0^{@\frac{1}{3}}, 2^{@\frac{1}{3}}\}\!\} \,/\, (2/3), \end{aligned}$$

that is $\{\!\{0^{@\frac{1}{2}}, 2^{@\frac{1}{2}}\}\!\}$, yielding again the uniform distribution $\{\!\{0, 2\}\!\}$ over h's values associated with this outcome as summarised at (2).

2.3. Program syntax and semantics

The programming language semantics is given in Fig. 1. In this presentation, for simplicity, we do not treat loops and therefore all our programs are terminating. (Loops are treated elsewhere [MM11].) In the conclusion we discuss this and other language restrictions.

[3] This is a different notational order from the usual notation $\{E \mid s{\in}S \wedge G\}$, but we have good reasons for the change: calculations involving both sets and quantifications are made more reliable by a careful treatment of bound variables and by arranging that the order $S/G/E$ is the same in both comprehensions and quantifications (as in $(\forall\, s\colon S \mid G \bullet E)$ and $(\exists s\colon S \mid G \bullet E)$).

[4] More generally we need subdistributions when modelling possible nontermination [MMM11] but here, for simplicity, we assume that all programs terminate.

Program type	Program text P	Semantics $\llbracket P \rrbracket(v,\delta)$
Identity	**skip**	$\{\!\{\ (v,\delta)\ \}\!\}$
Assign to visible	$\mathsf{v}{:=}\,E(\mathsf{v},\mathsf{h})$	$\{\!\{\ h{:}\,\delta \bullet (E(v,h), \{\!\{h'{:}\,\delta \mid E(v,h'){=}E(v,h)\}\!\})\ \}\!\}$
Assign to hidden	$\mathsf{h}{:=}\,E(\mathsf{v},\mathsf{h})$	$\{\!\{\ (v, \{\!\{h{:}\,\delta \bullet E(v,h)\}\!\})\ \}\!\}$
Choose prob. visible	$\mathsf{v}{:\in}\,D(\mathsf{v},\mathsf{h})$	$\{\!\{\ v'{:}\,(\mathcal{E}\,h{:}\,\delta \bullet D(v,h)) \bullet (v', \{\!\{h'{:}\,\delta \mid D(v,h')(v')\}\!\})\ \}\!\}$
Choose prob. hidden	$\mathsf{h}{:\in}\,D(\mathsf{v},\mathsf{h})$	$\{\!\{\ (v, (\mathcal{E}\,h{:}\,\delta \bullet D(v,h)))\ \}\!\}$
Composition	$P_1; P_2$	$(\mathcal{E}\,(v',\delta'){:}\,\llbracket P_1\rrbracket(v,\delta) \bullet\ \llbracket P_2\rrbracket(v',\delta'))$
Conditional choice	**if** $G(\mathsf{v},\mathsf{h})$ **then** P_t	$p{\times}\llbracket P_t\rrbracket(v, \{\!\{h{:}\,\delta \mid G(v,h)\}\!\})$
	else P_f **fi**	$+\ (1{-}p){\times}\llbracket P_f\rrbracket(v, \{\!\{h{:}\,\delta \mid \neg G(v,h)\}\!\})$
		where p is $(\mathcal{E}\,h{:}\,\delta \bullet G(v,h))$

For simplicity let $\mathcal{V}$ and $\mathcal{H}$ have the same type $\mathcal{X}$. Expression $E(\mathsf{v},\mathsf{h})$ is then of type $\mathcal{X}$, distribution $D(\mathsf{v},\mathsf{h})$ is of type $\mathbb{D}\mathcal{X}$ and expression $G(\mathsf{v},\mathsf{h})$ is Boolean. Expression p is of type $[0,1]$. Note that the extension to many variables $v_1, v_2, \cdots$ and $h_1, h_2, \cdots$, including local declarations, is straightforward [Mor06, Mor09].

The Assign-to semantics are special cases of the Choose-prob. semantics, obtained by making the distribution D equal to the point distribution $\{\!\{E\}\!\}$.

Fig. 1. Split-state semantics of commands

Programs take as input a split-state of type $\mathcal{V} \times \mathbb{D}\mathcal{H}$ and perform some computation; the output hyperdistribution gives a complete "digest" of the knowledge that an attacker has by then gained of the values of (v, h). As illustrated by our first example (1), this digest is a distribution over the split-states describing the probabilities with which particular behaviours can be observed, with the distribution δ in a split-state (v, δ) determined by Bayesian reasoning. In the semantics Fig. 1 both of these concepts are applied in the calculation of a program's output hyperdistribution.

Assignments to visible variables can release information if their results depend on the hidden state, as with $\mathsf{v}{:=}\,\mathsf{h}\ \mathbf{mod}\ 2$; similarly assignments to hidden variables can also release information, if their results can be correlated to facts already known, as in $\mathsf{h}{:=}\,2\mathsf{h}$ that establishes (and releases through knowledge of the source text) that h's low order bit is (now) zero. Branching at conditional choice is regarded as a visible event; however the behaviour of the program once that choice has been resolved is determined by the semantics of the subsequent execution. Thus the conditional

if $\mathsf{h} = 0$ **then** $\mathsf{v}{:=}\,0$ **else** $\mathsf{h}{:\in}\,\{\!\{2, 3\}\!\}$ **fi**

actually releases some information, in particular whether h is 0, as shown by the hyperdistribution that would result from $\{\!\{(0, \{\!\{0\}\!\})^{@\frac{1}{3}}, (v, \{\!\{2, 3\}\!\})^{@\frac{2}{3}}\}\!\}$ which is obtained after executing from initial hyper distribution $\{\!\{\ (v, \{\!\{0, 1, 2\}\!\})\ \}\!\}$.

The hidden and visible assignments are *atomic* actions in the sense that an attacker cannot observe intermediate values that might occur during the evaluations of their right-hand-sides.

Finally, we occasionally use local blocks: for example we write $|\![$ **hid** $\mathsf{h}';\ P(\ldots)\,]\!|$ for a locally declared hidden variable h' which is active within the scoping brackets $|\![\cdots]\!|$; the semantics within those brackets is defined by Fig. 1. Outside the scope we project onto the globally declared variables as follows. Let Δ' be the hyperdistribution result of $P(\ldots)$, so that it is composed of split-states in its support of the form (v, δ') where δ' is a joint distribution over the hidden state defined by global h and local h'. The hyperdistribution Δ obtained by projecting the local h' away is then given by $\Delta{:=}\,\{\!\{(v, \delta'){:}\ \Delta' \bullet (v, \{\!\{(h, h'){:}\ \delta' \bullet h\}\!\})\}\!\}$.

2.4. Entropy Refinement

Refinement between programs is defined so that programs' (average) ability to conceal the hidden state increases in the direction of the refinement order. In the following definition we show how to calculate the average information flow using Bayes Vulnerability of the output hyperdistribution.

Definition 2.1 *Bayes Vulnerability*

Let $\Delta{:}\ \mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})$ be a hyperdistribution; we define its *Bayes Vulnerability* $\mathsf{bv}(\Delta)$ to be the averaged maximal probability of guessing the value of its hidden value in $\mathcal{H}$:[5]

$$\mathsf{bv}(\Delta)\ :=\ (\mathcal{E}\,(v, \delta){:}\ \Delta \bullet \sqcup\delta) \qquad \square$$

[5] In conventional notation this would be written as $\sum_{(v,\delta):\mathcal{V}\times\mathbb{D}\mathcal{H}} \Delta(v, \delta){\times} \sqcup_{h:\mathcal{H}}\, \delta(h)$.

Entropy Refinement between programs is now defined to be the most permissive relation that preserves the order between Bayes Vulnerability in all contexts: a more refined implementation is able to withstand attacks defined by a program context at least as well as its less refined specification can.

Definition 2.2 *Entropy Refinement* Let P, P' be programs. We say that P is *Entropy refined* by P', writing $P \sqsubseteq P'$, if and only if for all initial split-states (v, δ), and for all contexts $\mathcal{C}(\cdot)$ expressed in the programming language defined in Fig. 1, we have

$$\mathsf{bv}(\,[\![\mathcal{C}(P)]\!](v,\delta)\,) \quad \geq \quad \mathsf{bv}(\,[\![\mathcal{C}(P')]\!]\,(v,\delta)\,). \qquad \square$$

Elsewhere [MMM10] we have shown that this combines both security and functional properties, and that it indeed defines a partial order between programs. In particular we can say that when $P \sqsubseteq P'$ then P' is *at least as secure* as P because Definition 2.2 ensures that for the same input $\mathcal{C}(P')$ outputs a hyperdistribution having Bayes Vulnerability no more than that output by $\mathcal{C}(P)$, for any context $\mathcal{C}$.

Functional properties are determined by the variables' values alone, ignoring their visibilities. For hyperdistribution Δ we can calculate the probability that the state is a specific value (v, h); it is

$$\mathsf{pr}(\Delta(v,h)) \;\; := \;\; (\mathcal{E}\,(v',\delta')\colon \Delta \bullet [v'{=}v] \times \delta'.h),$$

where we write $[v'{=}v]$ for the function that returns 1 if v is equal to v' and zero otherwise. We say that P' is *functionally consistent with* P if for all pairs (v, δ) we have $\mathsf{pr}([\![P]\!]\,(v,\delta)) = \mathsf{pr}([\![P']\!]\,(v,\delta))$.

The role of Entropy Refinement is summarised in the next theorem.

Theorem 2.1 *Partial order and functional consistency*

The relation between programs' outputs, defined at Definition 2.1, is a partial order. Thus it can be lifted pointwise to determine a partial order between programs of type $\mathcal{V} \times \mathbb{D}\mathcal{H} \to \mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})$.

We have that if $P \sqsubseteq P'$ then P' is at least as secure as P, and moreover is functionally consistent with P.

Proof That $(\sqsubseteq)$ yields a partial order and that P' is at least as secure as P was shown elsewhere [MMM10].

The proof that $(\sqsubseteq)$ implies functional consistency is a routine argument, albeit via a somewhat intricate context; we omit it here.

To see these ideas in action, consider the following small variations on the program defined at (1) above: we have

```
                       hid h;  vis v;
Program P:               h:= {{0, 1, 2}};
                         v:= h mod 2
                    ⋢
                         h:= {{0, 1, 2}};
Program Q:               v:= h mod 2;
                         v:= 0
                    ⊑
Program R:               h:= {{0, 1, 2}};
                         v:= 0.
```

The first two programs are not in the refinement order, i.e. $P \not\sqsubseteq Q$, because of an obvious functional inconsistency: variable v is finally 0 in Program Q but might not be in Program P.

To prove the refinement $Q \sqsubseteq R$ is more involved, however, since it in principle requires consideration of all contexts. But in fact it can be shown via an equivalent formulation of refinement in terms of a direct relation between the hyperdistributions. Details of that relation can be found elsewhere [MMM10] and are not required for the remainder of this paper. However a "sanity check" with respect to the context defined above shows that indeed the Bayes Vulnerability of the final hyperdistribution output by R is 2/3 and is no more than that of Q.

2.5. Comparative security with Entropy Refinement

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. Crucially however we will have suffered a failure of calculation, not of guesswork: only the former can be audited.

Using refinement as a basis for verification supports a comparative approach to security analysis, putting the onus on the specifier to summarise precisely the level of security that his application requires. Rather than determining whether a specific security property holds of a protocol by direct scrutiny, instead we only guarantee that it is "as secure as" the specification it refines. Thus 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. This approach is related to universal composability [Can01], which we discuss further in our conclusions.

In the remainder of this section we show how to specify part of a commitment scheme; for that we describe how the simple framework above (with a single hidden variable) applies even in a multi-agent system where the security requirements of each agent differ from the others.

2.6. Agent-based points of view

In a system consisting of a number of agents, each agent has a limited knowledge of the system state, determined by his *point of view*; and different agents probably have different views. The above simple semantics assumes that the state is divided into a hidden part and a visible part, reflecting only a single agent's viewpoint. We extend this by introducing a visibility modifier, attached to variable declarations, so that we can give an agent-centered interpretation to a program. Variables declared to be $\mathbf{vis}_{list}$ are interpreted as being visible (v) variables if a named agent is in $list$ and as hidden (h) variables otherwise. More precisely,

- **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).

For example, suppose Agents A, B each have a hidden Boolean variable and want to reveal the conjunction of the two variables but nothing more; we can specify those functional and security requirements as follows:

$$\begin{array}{l} \mathbf{vis}_A\ \mathsf{a}\colon\mathbb{B};\quad \mathbf{vis}_B\ \mathsf{b}\colon\mathbb{B};\quad \mathbf{vis}\ \mathsf{v}\colon\mathbb{B}; \\ \quad \mathsf{v}:=\mathsf{a}\wedge\mathsf{b}. \end{array} \tag{4}$$

Now from A's viewpoint the specification would be interpreted with a and v 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 v is still visible.

For each viewpoint we use Fig. 1 to calculate the semantics based on the simple **vis**/**hid** separation; we say that Entropy Refinement is valid for an agent-based system only if Definition 2.2 is valid for each individual viewpoint.

2.7. Specification of an ideal commitment scheme

A commitment scheme between two agents S, R allows S to choose a secret value s, and then to give a token r to R that does not reveal that value s but nevertheless inhibits S from trying later to change his mind. For example R might claim to be a mind reader, and to test this S secretly chooses a number s, then giving a token r to R. When R tries to guess s he is not to gain any advantage from knowing r—this is called the *hiding* property of the scheme; correspondingly, when R correctly announces the number s, he can use r to make it difficult for S to claim that in fact he was thinking of another, different number—this complementary property is called *binding*.

Those two properties, hiding and binding, are idealistic; and many commitment schemes use computational complexity in order to approximate the ideal, that successful cheating is not possible. We use our probabilistic model to look further at them.

We use a trusted third-party T to describe a simple version of the scheme as follows, in which $\mathbb{B}$ denotes the type Boolean or equivalently {0, 1} as convenient:

```
vis$_{ST}$ s: $\mathbb{B}$;  vis$_T$ t: $\mathbb{B}$;  vis$_{TR}$ r: $\mathbb{B}$;

  s:∈ {{0, 1}};        S selects a value s, visible to S and T only
  t:∈ {{0, 1}};        T selects a random bit, visible only to him
  r:= t⊕s              R receives s but encrypted with t, visible to T and R only
```

We write $t \oplus s$ for the "exclusive-or" between Booleans t and s.

To see this from R's point of view, we convert the annotations **vis**$_{ST}$ and **vis**$_T$ both to **hid**, and **vis**$_{TR}$ to **vis**, giving the declarations **hid** s, t; **vis** r followed by the same code as above. (From S's point of view we would have had instead **vis** s; **hid** t, r.)

The output hyperdistribution of the above program (from R's point of view) is thus

$$\{\!\{ (0, \rceil\mathbb{B}^2\lceil),\ (1, \rceil\mathbb{B}^2\lceil) \}\!\},$$

where the first component (0 or 1) is the visible r and the shadow component $\rceil\mathbb{B}^2\lceil$ denotes the uniform distribution over the set $\mathbb{B}^2$ that assigns (in this case) probability $1/4$ to each of the four possibilities for the hidden s, t. The probability of R guessing s is then calculated directly as $1/2$, no better than a purely random guess would have been.

Observe that the role played by the trusted third party is absolutely crucial here, allowing us to escape the apparent paradox in respect of the well-known impossibility of a commitment scheme, if it involves only direct exchange of information between S and R, to be simultaneously perfectly binding and hiding. In our terms we phrase this impossibility result as saying that the scheme sketched above *cannot be implemented* without the trusted third party T.

We now elaborate the simple scheme above, introducing features which (necessarily) weaken it, and by doing so allow it subsequently to be implemented without the third party T.

2.8. A more elaborate abstract-commitment scheme

Instead of restricting ourselves to Booleans, we allow more general values s, r of type $\mathcal{S}$, $\mathcal{R}$ resp. And our more elaborate third party T now controls a partial function $f\colon \mathcal{S} \nrightarrow \mathcal{R}$ instead of a simple Boolean; the function is initially everywhere undefined. The third party is thus

```
vis$_{ST}$ $s\colon\mathcal{S}$;  vis$_{TR}$ $r\colon\mathcal{R}$;
vis$_T$ $f\colon\mathcal{S}\nrightarrow\mathcal{R}$;          |Initially everywhere undefined

  if $s \in$ dom $f$ then $r{:=}f.s$ else   | Make a commitment to s
    r:∈ $\rceil\mathcal{R}\lceil$;                   |
    $f.s{:=}r$                        |
  fi ,                              |
```

where by $f.s := r$ we mean an assignment to variable f that extends it precisely so that it becomes defined at value s and is given value r there. Then S makes a commitment to value s by executing the code so-labelled and R receives (as before) a token, namely r.

The effectiveness of this scheme depends crucially on the distribution used to choose new values for r, and in the above we have used the uniform distribution $\rceil\mathcal{R}\lceil$ over $\mathcal{R}$. If—at one extreme—that distribution were instead $\{\!\{r_0\}\!\}$ say, for some fixed r_0, then the scheme would provide perfect hiding but no binding: every s would return the same token r_0 and S could claim to have chosen any s he wishes. If on the other hand that distribution were $\{\!\{s\}\!\}$, then it would provide perfect binding but no hiding since R would know from the token r what value s generated it.

For a uniform assignment to r, the Bayes Vulnerability varies depending on the viewpoint. From R's viewpoint the vulnerability for each (r, s) pair is $1/\#\mathcal{R}$; however from S's point of view, the larger the domain of f, the greater his chance of cheating successfully. For example if dom f has cardinality 2, then his chance of being able to change his mind (and not be caught) is $1/\#\mathcal{R}$, but increases to $2/\#\mathcal{R} - 1/\#\mathcal{R}^2$ if dom f has cardinality 3.

Now the advantage of this less-than-ideal third party is that in practice it can be replaced by a one-way function whose characteristics lie somewhere between the two extremes above: the possible tokens do occupy most of $\mathcal{R}$ for any given s, and yet do not much overlap for different values of s. If the one-way function cannot be inverted in polynomial time, then it can be visible to all so need not be hidden by a third party.[6]

In fact the specification we have given is essentially the "random-oracle model" [BR93], since it uses randomisation to abstract from details of the precise encryption method. We note however that the random-oracle model has been shown to be a technically unsafe abstraction for some notions of security [RCH04]. One way to phrase the observed problems is that there are programs which use the random oracle which have no secure refinements in terms of known cryptographic hashes. In spite of this the random oracle still plays a useful role in specification; it is a topic of future research to understand under what circumstances the random oracle can be usefully implemented.

In this section we have summarised a probabilistic model for non-interference properties, together with Entropy Refinement to enable programs' relative security properties to be compared. In the next section we summarise an abstraction of this model with a view to automating a class of Entropy-Refinement proofs.

3. The Shadow Semantics: a possibilistic abstraction of hyperdistributions

Proofs of refinement in the probabilistic model explained above currently need careful manipulations involving real numbers; and both manual and automated reasoning would be far simpler in a qualitative model in which concrete probabilities have been replaced with abstract possibilities. The cost of such an abstraction is however a potential loss of precision: such qualitative models do not provide statistical guarantees of either program security or functionality. Nevertheless there are situations where the possibilistic and probabilistic semantics coincide, and indeed many security protocols are carefully designed so that this is the case; in Sect. 4 we consider the circumstances under which the relationship is valid. Before we study the relationship we first recall the Shadow model for qualitative noninterference [Mor06].

As for the quantitative model, we use (qualitative) split-states, representing possible states of the program from the view of an attacker; these are now more abstractly of type $\mathcal{V} \times \mathbb{P}\mathcal{H}$. Thus (v, H) represents a state in which the visible variables are known to have value v and the set of possible values that the hidden state might take is known to be H. The Shadow Semantics does not quantify the likelihood that the program terminates in a given split-state; instead it represents the possible program outcomes as a set, so that Shadow programs have type:

$$\mathcal{V} \times \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathbb{P}\mathcal{H}).$$

In Fig. 2 we set out the Shadow interpretation for a small programming language; it is similar to but simpler than that defined in Fig. 1. As above we assume that there is only one visible variable $\mathsf{v}{:}\,\mathcal{V}$ and one hidden variable $\mathsf{h}{:}\,\mathcal{H}$. Programs behave essentially as they do in the quantitative interpretation, but wherever probabilities were observed, now only possibility is recorded.

Consider again the program (1), this time with a Shadow interpretation:

```
hid h;  vis v;
   h:∈ {0, 1, 2};                                        (5)
   v:= h mod 2.
```

The result set of this program is $\{(0, \{0, 2\}), (1, \{1\})\}$, which has two possible outcomes, each one bearing a different uncertainty as to the hidden state. In particular if the visible result is 0 then we deduce that the hidden state is contained in the set $\{0, 2\}$, but that if the visible state is 1 then we deduce exactly the value of h.

[6] This is intended to capture the spirit of the standard cryptographical notion, since our formalism is unable to express complexity properties. For example the standard definition includes the assumption that a one-way function cannot be inverted (with a non-negligible chance of success) in (probabilistic) polynomial time.

Program type	Program text P	Semantics $[\![P]\!](v,H)$
Identity	**skip**	$\{\ (v,H)\ \}$
Assign to visible	$\mathsf{v}{:=}\,E(\mathsf{v},\mathsf{h})$	$\{h{:}\,H \bullet\ (E(v,h),\ \{h'{:}\,H \mid E(v,h') = E(v,h)\})\ \}$
Assign to hidden	$\mathsf{h}{:=}\,E(\mathsf{v},\mathsf{h})$	$\{\ (v,\{h'{:}\,H \bullet E(v,h')\})\ \}$
Choose visible	$\mathsf{v}{:\in}\,S(\mathsf{v},\mathsf{h})$	$\{\ h{:}\,H; v'{:}\,S(v,h) \bullet\ (v',\ \{h''{:}\,H \mid v' \in S(v,h'')\})\ \}$
Choose hidden	$\mathsf{h}{:\in}\,S(\mathsf{v},\mathsf{h})$	$\{\ (v,\{h'{:}\,H; h''{:}\,S(v,h) \bullet h''\}\ \}$
Composition	$P_1; P_2$	$\mathsf{lift}.[\![P_2]\!]([\![P_1]\!](v,H))$
Conditional Choice	**if** $E(\mathsf{v},\mathsf{h})$ **then** P_t **else** P_f **fi**	$[\![P_t]\!].v.H_t \cup [\![P_f]\!].v.H_f$ where $H_t{:=}\,\{h'{:}\,H \mid E(v,h')\}$ and $H_f{:=}\,\{h'{:}\,H \mid \neg E(v,h')\}$

For simplicity we assume that the state of the visible and hidden variables is $\mathcal{X}$; that $E(\mathsf{v},\mathsf{h})$ is an expression in $\mathcal{X}$ possibly involving free variables v and h, and $S(\mathsf{v},\mathsf{h})$ is an expression in $\mathbb{P}\mathcal{X}$. The function $\mathsf{lift}.[\![P_2]\!]$ applies $[\![P_2]\!]$ to all pairs in its set-valued argument, un-Currying each time, and then takes the union of all results.

The Assign-to semantics are special cases of the Choose semantics, obtained by making the set S equal to the singleton set $\{E\}$.

Fig. 2. Shadow Semantics of commands [Mor06]

As in the quantitative model, we assume that the attacker has knowledge of the program text, can observe the visible variables directly as the program executes, and can make deductions, although not actual observations, to determine the possibilities for the hidden state. Unlike the quantitative semantics, however, the (Shadow) attacker is unable to compute whether any hidden value in a split state (v, H) is more likely than any other, nor is it possible to determine how likely is each observed split state. For example, the split state $(1, \{1\})$ as a possible outcome of (5) denotes an observable event in which the hidden state is revealed in its entirety. In some applications this might only be acceptable if its occurrence was extremely rare, something that only the quantitative semantics can express.

Qualitative Shadow Refinement (written $\sqsubseteq_S$) between programs now preserves possibility; it is defined so that $P \sqsubseteq_S P'$ holds if and only if for all contexts $\mathcal{C}(-)$ expressed in the programming language, whenever the attacker can guess h correctly after executing $\mathcal{C}(P')$, then he can also do so for $\mathcal{C}(P)$. (Here an attacker is able to guess the value of the hidden state if the result set contains a pair of the form $(v, \{h\})$ for some h, since then a guess $\mathsf{h}{=}h$ is guaranteed to succeed.) Elsewhere [Mor09] it was shown that this formulation of refinement is equivalent to a more direct definition based on the result sets given next.

Definition 3.1 *Shadow Refinement*

For programs P, P' we say that P *is Shadow refined by* P' and write $P \sqsubseteq_S P'$ just when for all incoming split-states (v, H) and final split states (v', H') in $[\![P']\!](v, H)$ we can find some $H_1 \cdots H_N$ such that

1. each (v', H_n) is in $[\![P]\!].(v, H)$ for each $1{\le}n{\le}N$, and
2. $H_1 \cup \cdots \cup H_N = H'$,

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

This means that for each initial (v, H) every final result (v', H') produced by P' must be "justified" as the union of a set of results (v, H_n) produced by P and with $v{=}v'$. Informally, refinement in this model is effectively joining shadows with union. □

Thus a program which produces results which record a greater degree of uncertainty in their split-states (v, H) are regarded as "more secure" than those producing results with a lesser degree of uncertainty. Only in the case that H is a singleton can the attacker know the hidden state for sure.

4. Quantitative guarantees in a qualitative framework

In earlier work [McI09, MM09], we have given hand-verified derivations in the qualitative Shadow Semantics of a number of cryptographic protocols, including the Dining Cryptographers [Cha88], Perfect Information Retrieval [CGKS99], the Oblivious Transfer [Rab81], and multiparty computations [Yao82]. In this section we describe the conditions under which such proofs are also valid in the quantitative model presented in Sect. 2. For this we will abstract from actual probabilities by using the Shadow model described in Sect. 3, based on sets rather than the distributions of Sect. 2, but with a new convention of how those sets are to be interpreted. Rather than denoting complete ignorance (as resulting from demonic nondeterminism), we will take the inner sets, the shadows, to be denoting "pre-uniform distributions," which are defined next.

4.1. Pre-uniform programs

The characteristic feature of quantitative programs that can nevertheless be analysed qualitatively is that they make use of probability only in a restricted way. Using only uniform distributions is an obvious restriction; but it turns out to be too strong. Instead, we will concentrate on programs that deal only on distributions on $\mathcal{H}$ that are "pre-uniform," which is to say that they are uniform on their supports, but can be zero elsewhere on $\mathcal{H}$. Here we set out the details.

- Say that a distribution δ is *pre-uniform* on some set D just when it is uniform over its support, that is when δ is equal to $\rfloor\lceil\delta\rceil\lfloor$. Note that although there is only one uniform distribution $\rfloor D\lfloor$ in $\mathbb{D}D$, there are many pre-uniform distributions: in fact, there is one for each non-empty subset of D.[7]
- Write $\mathbb{U}D$ for the set of pre-uniform distributions on D.
- Say that a program in $\mathcal{V}\times\mathbb{D}\mathcal{H}\to\mathbb{D}(\mathcal{V}\times\mathbb{D}\mathcal{H})$ is pre-uniform if when restricted to input split-states whose $\mathcal{H}$-portions are pre-uniform, i.e. in $\mathcal{V}\times\mathbb{U}\mathcal{H}$, it gives output hyperdistributions whose inners are also pre-uniform, i.e. are in $\mathbb{D}(\mathcal{V}\times\mathbb{U}\mathcal{H})$. Note that the outer distribution of the hyper need not be pre-uniform.

It turns out that many cryptographic protocols are pre-uniform in their specifications. This is true, for instance, of the specification of the so-called *Lovers' Protocol* given in (4),

$$\mathsf{v} := \mathsf{a} \wedge \mathsf{b},$$

in which two people A and B wish to know whether their interest in pursuing a relationship, modelled by a and b respectively, is mutual; however they are both unwilling to make a public declaration for fear of embarrassment in the case that their feelings are not returned. The protocol specification at (4) achieves that, setting visible variable v to true only when the interest is mutual. We can see, for example, that when we quantitatively model the program from the perspective of A (i.e. with v and a considered as visible variables and b as a hidden variable), in a context where a and b are uniformly initialised the final hyperdistribution of the program with triples ((v, a), $\mathbb{DB}$), is

$$\{\!\{((\mathsf{true},\mathsf{true}),\{\!\{\mathsf{true}\}\!\})^{@\frac{1}{4}},((\mathsf{false},\mathsf{true}),\{\!\{\mathsf{false}\}\!\})^{@\frac{1}{4}},((\mathsf{false},\mathsf{false}),\{\!\{\mathsf{true},\mathsf{false}\}\!\})^{@\frac{1}{2}}\}\!\}$$

and that although the program declassifies information—the mutual interest of A and B in this case–the hidden distribution of b remains uniform in each of the final split-states that the attacker A may find herself in. In the case that A is interested in pursuing a relationship with B, then regardless of the outcome of v her knowledge of b is a point distribution on the exact value of b, whereas if A has no interest, then her final knowledge of b is uniformly distributed over both possibilities for b.

This raises the attractive possibility that pre-uniform programs can be represented, and reasoned with, using the simpler qualitative model. This is an issue we now explore.

4.2. Pre-uniformity by construction

Pre-uniformity is determined entirely by the individual assignments and the control structure of the program, because sequential composition preserves it. To see that, consider $P;\ Q$ and assume that both P and Q are pre-uniform. Observe that the definition of composition at Fig. 1 applies $[\![Q]\!]$ to the inner split-states in the output $[\![P]\!](v,\rfloor H\lfloor)$. Since P is pre-uniform then each of those output split-states must be pre-uniform; the result now follows by pre-uniformity of Q.

With this observation the next lemma sets out some syntactic conditions on program texts that are sufficient to make them pre-uniform: any program constructed by observing all these conditions must therefore be pre-uniform. Some of these conditions imply a syntactic restriction on program construction, and some are conditions placed on the functional updates.

[7] Note that D must be finite for the existence of a uniform (discrete) distribution with support D.

Lemma 4.1 *Sufficient conditions for pre-uniformity*
Given a program P written in the language of Fig. 2, we have that P is uniform-preserving within a given local-variable declaration scope (without further embedded local scopes) if the following constraints hold on its conditional updates:

1. Hidden variables are assigned (i.e. initialised) only once.
2. Given Constraint (1), hidden variables may be initialised deterministically, as in $\mathsf{h}{:=}\, E(\mathsf{v},\mathsf{h})$, or probabilistically, as in $\mathsf{h}{:\in}\, S(\mathsf{v},\mathsf{h})$, as long as for any given visible-state value v, the size of the set $S(v,h)$ is independent of the value h of the (entire) hidden state h.
   For example, the initialisation $\mathsf{h}{:\in}\,\mathbb{B}$ is valid, as the set of Booleans $\mathbb{B}$ is independent of all variables in the hidden state.
   However $\mathsf{h}_1{:\in}\,\{0,1,2\};\;\; \mathsf{h}_2{:=}\,\{0,\ldots,\mathsf{h}_1\}$ is not pre-uniform: the probability that the hidden state $(\mathsf{h}_1,\mathsf{h}_2)$ equals $(0,0)$ is $1/3$, but the probability that it equals $(1,0)$ is $1/6$.
3. Any deterministic assignments $\mathsf{v}{:=}\, E(\mathsf{v},\mathsf{h})$ are permissible. Non-deterministic assignments $\mathsf{v}{:\in}\, S(\mathsf{v},\mathsf{h})$ to the visible state, for example $\mathsf{v}{:\in}\, S(\mathsf{v},\mathsf{h})$, are allowed if whenever $S(v,h)$ and $S(v,h')$ have non-empty intersection, then they have the same size.
   For example, the assignment $\mathsf{v}{:\in}\,\mathbb{B}$ is acceptable as is $\mathcal{V}{:\in}\,\{\mathsf{h},\ (\mathsf{h}{+}1)\,\mathbf{mod}\,N\}$ for hidden variable $\mathsf{h}\in 0\ldots N{-}1$ where N is greater than one.
   However the assignment $\mathsf{v}{:=}\,\{0,\mathsf{h}\}$ is not allowed: for if v is set to 0 then by conditional reasoning the attacker interpreting the probabilistic semantics deduces that h is strictly more likely to be 0 than any other value. Intuitively, this is because all the choice-sets intersect (since each one contains 0), but one of them has size 1 (when $\mathsf{h}=0$) whereas all of the others have size at least 2.

Finally, we need a condition for dealing with local variables. When the local variables of programs satisfying the above three conditions are removed from scope, the resulting hidden distribution is also uniform if the following constraint holds in the Shadow-Semantic interpretation of these programs:

4. Let H be the Shadow set of the program with local and global hidden variables, representing a uniform distribution δ over the elements in the set. The projection of δ onto the global variables only is uniform if there exists a constant k such that for each $h{\in}H$ we have
   $$k = \#\{h'{:}\,H \bullet \mathit{global}.h' = \mathit{global}.h\},$$
   where we write $\mathit{global}.h$ for the projection of the hidden state h onto the global variables.

Proof (Sketch) Conditions (1 and 2) imply pre-uniformity as follows. The problematic case occurs when some hidden variable h_1 has already been initialised; a second initialisation of a fresh h_2 might disturb the overall uniform distribution (as the example illustrates). Let $\rfloor H_1\lceil$ be the (uniform) distribution over values taken by h_1, and let $\delta'(h_1,h_2)$ be the resulting distribution after executing $\mathsf{h}_2{:\in}\, S(\mathsf{v},\mathsf{h}_1)$. Then

$$\delta'(h_1,h_2) \quad = \quad \rfloor H_1\lceil(h_1)\times\rfloor S(v,h_1)\lceil(h_2) \quad = \quad \rfloor H_1\times S(v,h_1)\lceil(h_1,h_2),$$

where the final equality holds if $\#S(v,h_1)=k$ for some constant k independent of h_1.
Condition (3) follows similarly.
Finally condition (4) also follows similarly by observing that, if $\delta(h_1,h_2)$ is a distribution over variables $\mathsf{h}_1,\mathsf{h}_2$ with h_1 global and h_2 local, then projecting onto the h_1 component yields a distribution $\delta'(h_1)=\sum_{h_2}\delta(h_1,h_2)/\#\,\mathit{global}^{-1}(h_1)$. This yields a uniform distribution if $k=\#\mathit{global}^{-1}(h_1)$ for any h_1, and $\delta(h_1,h_2)$ is uniform over its support. □

Consequently, we have that for any two programs S and I that satisfy the conditions stipulated in Lemma 4.1 and are qualitatively equal, that is $[\![S]\!]{=}[\![I]\!]$, then they are quantitatively equal as well, except for external probability provided they receive uniformly distributed internal inputs.

Lemma 4.2 (*Uniform consistency*)
Let S, I be programs expressed in the language of Fig. 2. If they are both syntactically pre-uniform in the sense of Lemma 4.1, and further are Shadow equivalent, namely $[\![S]\!]{=}[\![I]\!]$, then

$$\lceil (\!\lbrack S \rbrack\!)(v,\rfloor H\lceil)\rceil \quad = \quad \lceil (\!\lbrack I \rbrack\!)(v,\rfloor H\lceil)\rceil,$$

for any initial pre-uniform split-state $(v,\rfloor H\lceil)$.

Proof Follows from Sect. 4.1, Definition 3.1 and the observation that pre-uniformity is maintained by composition. □

In the following section we explore extra conditions that are sufficient to guarantee that the external probabilities of Shadow-equivalent programs also coincide, for uniformly distributed inputs.

4.3. Recovering the full probabilistic semantics

Lemma 4.2 gives some guidance about when equality between programs—as measured by the Shadow Semantics– can safely be used to deduce that the attacker's knowledge of the hidden state is uniformly distributed over the possibility set. However, as the next example shows, it can only be regarded as a very approximate estimate of the *frequency* with which those possibility sets are observed. Consider two programs, P_1 defined by (1), and P_2 below which similarly sets global v and h, but uses a slightly perplexing local block to do so:

```
P_2  :=   hid h;  vis v
          |[ hid h_1 :∈ {0, 1};  h_2 :∈ {0, 1};
             if h_1 + h_2 = 1 then h := 1;  v := 1 else h :∈ {0, 2};  v := 0 fi
          ]|
```

Note that both P_1, P_2 conform to the conditions set out in Lemma 4.1 and indeed, each is pre-uniform. Moreover $[\![P_1]\!]=[\![P_2]\!]$, thus we deduce by Lemma 4.2 that

$$\lceil (\!|P_1|\!)(v, \rfloor H \lceil) \rceil \quad = \quad \lceil (\!|P_2|\!)(v, \rfloor H \lceil) \rceil \, .$$

However an analysis of their corresponding full probabilistic interpretation—both internal and external—reveals that the probability that $\mathsf{h}=1$ is only 1/3 in P_1 but it is 1/2 in P_2 (because there are two ways that the condition ($\mathsf{h}_1+\mathsf{h}_2 = 1$) holds for independent initialisations of $\mathsf{h}_{1,2}$). The problem here is that Lemma 4.2 gives no information about the *external probability*, and this is exactly where P_1 and P_2 differ.

The quantitative semantics allows thresholds to be placed on the probability of an insecurity, so that in cases where absolute secrecy cannot be achieved its severity can at least be contained. Here the precise measurement of the threshold might be that an observable leak occurs only with a very small probability; in this case an overall measurement of security within a larger context can then be calculated. For programs which are Shadow equivalent to a specification, but do not respect its thresholds on external probabilities, the use of those programs could result in an unsafe implementation. This would be a failure of standard Shadow equivalence to be compositional in the quantitative context; in what follows we study how to repair it.

We observe first that recovering external probabilities from the Shadow is in general not possible—after all, the Shadow intentionally abstracts many of the complications involved in keeping track of detailed probabilities. However there is a class of protocols where the observed probabilities are correlated with the size of the Shadow sets; in these cases the Shadow interpretation actually summarises the full probabilistic behaviour, with no loss of information.

A simple example of this phenomenon is illustrated by a pre-uniform program P which, on input (v, H), produces a single output result (v', H'): in this case the only possible output distribution in the probabilistic interpretation is $(v', \rfloor H' \lceil)$, and there are no external probabilities to worry about. In protocols which simply declassify some information, whilst keeping other parts of the state secret—a typical requirement in many security protocols—we have a slightly more complicated, but still similar situation where the "probabilistic attacker" using the quantitative semantics yields no more information than does a "Shadow attacker" using the qualitative semantics. For example if *all* the information is released after a single run of the program, then the "probabilistic attacker's" statistical analyses grant him no more distinguishing power than the Shadow attacker's straightforward possibilistic reasoning. Amongst the pre-uniform programs there is a subclass of programs which declassify information in such a way that the full probabilistic semantics can be calculated directly from an examination of the Shadow Semantics. We end this section by highlighting some important situations where this can be done.

The next definition formalises when the split-state observables can be distinguished by their supports rather than the details of their (inner) probability distributions.

Definition 4.1 *h-disjoint* Program P is *h-disjoint* if for any pair (v', δ') and (v'', δ'') resulting from the same input split-state we have either $\lceil \delta' \rceil \cap \lceil \delta'' \rceil = \varnothing$ or in fact $(v', \delta') = (v'', \delta'')$. □

For example the program (4) is h-disjoint, but v:= {0, h} is not, and as explained above, it does not give pre-uniform results. The h-disjoint condition allows us to set out a situation where the Shadow Semantics actually determines the probabilistic semantics.

Lemma 4.3 Let program P be h-disjoint. If in addition P does not modify the input hidden state, then

$$(\!|P|\!)(v, \rfloor H \lceil) \quad = \quad \{\!\{(v_1, \rfloor H_1 \lceil)^{@\frac{\#H_1}{\#H}}, \ldots, (v_N, \rfloor H_N \lceil)^{@\frac{\#H_N}{\#H}}\}\!\} \tag{6}$$

for some $v_1, \ldots, v_N$ and disjoint partitioning $H_1, \ldots, H_N$ of H.
Proof (Sketch) The h-disjoint assumption implies that the output hyperdistribution has the form

$$\{\!\{(v_1, \delta_1)^{@p_1}, \ldots, (v_N, \delta_N)^{@p_N}\}\!\},$$

where the support of δ_i is disjoint from that of δ_j for $i \neq j$. Since P does not modify the value of the input hidden state, we deduce that

$$\rfloor H \lceil \quad = \quad p_1 \times \delta_1 + \cdots + p_N \times \delta_N. \tag{7}$$

Now since the supports are pairwise disjoint (7) implies that $\delta_i = \rfloor \lceil \delta_i \rceil \lceil$, and therefore that $p_i = 1/\#\lceil \delta_i \rceil$. □

Finally we can determine when a Shadow analysis is enough for a probabilistic analysis.

Theorem 4.1 Let programs S and I be expressed in the language of Fig. 2, satisfying the conditions of Lemma 4.2. Suppose also that neither modify the global hidden state. If now either S or I is h-disjoint then

$$[\![S]\!](v, H) = [\![I]\!](v, H) \quad \Rightarrow \quad (\!|S|\!)(v, \rfloor H \lceil) = (\!|I|\!)(v, \rfloor H \lceil),$$

for all split-states (v, H).

Proof Suppose that S is h-disjoint. Lemma 4.3 now implies that its output hyperdistribution is of the form (6), and Lemma 4.2 implies that the output hyperdistribution $(\!|I|\!)(v, \rfloor H \lceil) = p_1 \times \rfloor H_1 \lceil + \cdots + p_N \times \rfloor H_N \lceil$, for the partition $H_1, \ldots, H_N$ appearing in $(\!|S|\!)$'s result hyperdistribution. The result now follows, since $\rfloor H \lceil = p_1 \times \rfloor H_1 \lceil + \cdots + p_N \times \rfloor H_N \lceil$. □

Observe that Lemma 4.1 allows the Bayes Vulnerability to be calculated as $N/\#H$ for the programs that satisfy its assumptions, where N is the number of partitions in the Shadow Semantics.

Although Lemma 4.1 seems to be quite restrictive, surprisingly it applies in many situations, such as in the examples treated here and elsewhere. These include the Oblivious Transfer [Rab81], the Dining Cryptographers [Cha88], Private Information Retrieval [CGKS99], as well as general multi-party computations [Yao82] and the specification of the commitment at Sect. 2.8. Thus for this class of program a qualitative Shadow analysis also provides a detailed probabilistic analysis, implying that automated proofs of probabilistic correctness is a viable goal.

More generally, although we have not explored the completeness aspects of our restricted language, it might be that any pre-uniform program is expressible using it; and that might explain the range of examples that we have been able to model and verify. If that turns out to be the case, then one possible research direction would be to use the language as a guideline for structuring specifications.

5. Automated analysis of Shadow refinement using Rodin

In this section we show how our semantic constructions can be used to automate Shadow refinement proofs using Event-B and its associated supporting Rodin platform [ABH+10]. Event-B is a modelling method for formalising and developing systems whose components can be modeled as discrete transition systems. An evolution of the (classical) B-method [Abr96], Event-B is now centered around the general notion of *guarded events*. The semantics of Event-B based on transition systems and simulation between such systems, is described in [ABH+10]. Event-B models consist of contexts and machines. Contexts define static parts of the model, e.g. carrier sets (types), constants, and axioms constraining them. Machines specify dynamic parts of the model, represented as state transition systems. The state of each machine is captured by variables v and is constrained by invariant $I(v)$. Transitions correspond to guarded events of the form e $\widehat{=}$ **when** $G(v)$ **then** $S(v)$ **end** ,[8] where $G(v)$ is the

[8] In general, events can have parameters, which we leave out for clarity.

guard stating the necessary condition for invoking the event, and $S(v)$ is the action describing how variables v are updated. Event-B supports (standard) stepwise refinement for developing systems. In particular, an abstract machine containing variables v is refined by a concrete machine containing variables w with some gluing invariant $J(v, w)$ if it establishes a simulation of the concrete machine by its abstract counterpart. Typically, simulation is proved on a per-event basis: a concrete event is indicated as the refinement of an abstract event. The supporting Rodin platform of Event-B includes tools for generating proof obligations and provers for discharging them, either automatically or interactively.

We chose Event-B/Rodin because (1) it is straightforward to make the Event-B refinement sensitive to security refinement by explicitly including a "ghost" variable to track the Shadow, and (2) Rodin automatically generates the proof obligations associated with security refinement, and assists in discharging those proof obligations. Our formalisation in Rodin raised a number of technical challenges—the first was how to manage the Shadow variables within a modelling framework, and the second was in Rodin's treatment of concurrent execution.

To overcome the complications associated with the explicit Shadow, we developed a translator which takes high-level protocol descriptions written in a custom *security language* into Event-B suitable for importing into Rodin. This allows a user to abstract from creating and updating the Shadow in an ad hoc manner, leaving the translator to generate and install the invariants for Shadow refinement. Once the generated specification has been imported by Rodin, its automatic provers take over to discharge the proof obligations sufficient to prove Shadow refinement. In our experiments we found that most of the obligations were automatically discharged by Rodin, while the others can be proven with limited user assistance. We also found that an alternative equivalent form of the Shadow Semantics based on a state as a triple (v, h, H) was more convenient for encoding the consistency checking between refinements, where the h component represents a specific hidden value contained in the set H.

To deal with the second challenge—that Rodin's concurrent execution model prevented some refinements to be proved—we developed a modelling style to force a particular sequence of sequential execution which allowed Rodin's model to reflect more accurately the sequential execution in our security model.

5.1. The security language

Figure 3 shows a development of the lovers' protocol (4) written in our language as a specification and three refinements, separated by the refinement symbol `[=`. Each protocol is described by (a) variable and set declarations, (b) function declarations, and (c) a sequence of imperative commands.

The specification of the lovers' protocol has three variables `a`, `b`, representing the respective interest of the two participants in each other and `v`, the result $a \wedge b$. `a` and `v` are declared to be visible (keyword `VIS`) since their values are known to the participant from whose perspective the protocol is defined. `b` is declared to be hidden (keyword `HID`) since its value is unknown to this participant. The variables take on values from an abstract set `X` which is implicitly declared by use; in this protocol, `X` models Boolean.

An abstract function `AND` is also declared. Our language allows arbitrary Rodin axioms to be included in a protocol description (not shown in Fig. 3). For example, we specify commutativity for `AND` with the following declaration:

```
AXIOM "∀ xA, xB · AND (xA ↦ xB) = AND (xB ↦ xA)"
```

In general, axioms define the properties of the abstract functions, including their relationships to each other.

Finally, each command is labelled. The sole command `result` of the specification defines the variable `v` to be the `AND` of the two separate lovers' variables. Functions can be called using either infix or prefix syntax.

Protocol refinements are specified in the same way as the specification, except that variables, functions and axioms from an earlier refinement are assumed and do not have to be repeated. The command labels are used to indicate that a command is intended to be a refinement of the command that has the same label in the previous machine. The three refinements of the initial lovers' protocol specification also illustrate a selection command (`resultA`) which non-deterministically selects a value from the given set, instead of assigning a particular value.

The tool also supports other Rodin operators, but we omit discussion of them here since they are not needed for the example.

```
VIS a : X, v : X
HID b : X
FUN AND : X x X -> X

result: v = a AND b;

[=

VIS aP : X
HID bP : X
FUN XXOR : X x X -> X

resultA: aP :∈ X;
resultB: bP = XXOR ((a AND b), aP);
result: v = XXOR (aP, bP);

[=

FUN ZERO : X
FUN IF : X x X x X -> X

resultA: aP :∈ X;
resultB: bP = XXOR (IF (b, a, ZERO), aP);
result: v = XXOR (aP, bP);

[=

resultA: aP :∈ X;
resultB: bP = IF (b, XXOR (a, aP), aP);
result: v = XXOR (aP, bP);
```

Fig. 3. A development of the lovers' protocol expressed in our security language as a specification, followed by three refinements. Axioms have been omitted

5.2. Shadow set representation

The tool performs two main functions: given a protocol development expressed in the security language, it (i) carries out basic semantic checking and (ii) translates the protocols into Event-B specifications such that *if* the translation into Event-B is consistent,[9] *then* for each step "$P1 \sqsubseteq P2$" in the development, protocol $P2$, with newly introduced variable declarations treated as local declarations, is a Shadow-refinement of $P1$.

The basic semantic checking of the protocols includes analysis and type-checking. These steps are quite standard. After semantic checking, the tool translates the protocols into Event-B specifications in the following way. Each initial protocol specification and subsequent refinement is translated into an Event-B machine; all but the first machine are declared to refine the previous one. Sets, functions and axioms are declared in a context specific to the relevant machine.

Each initial protocol specification/refinement with visible variables `v`, hidden variables `h` are translated to an Event-B machine with variables `v`, `h` and an additional variable `H` representing the shadow of `h` (i.e., the inferred values of `h`). For example, the initial specification in Fig. 3) has a single hidden variable `b`, hence its shadow variable is $H_1 \in \mathbb{P}(X)$. Similarly, the shadow variable for the first refinement in Fig. 3 is $H_2 \in \mathbb{P}(X \times X)$ since there are two hidden variables: `b` (from the initial specification) and `bP`. Typing invariants are generated constraining variables and functions to be elements of the appropriate sets.

Each labelled protocol command becomes two events in the relevant machine. The first event, called shadow event, describes the update (if any) of the shadow according to the Shadow Semantics of Fig. 2. The second event, called functional event, describes the effect of the command on the protocol variables. Sequencing of events is achieved using a *program counter* variable pc (one for each Event-B machine) identifying the previously executed command, and a *synchronisation* flag $insync$ indicating if the state of the hidden protocol variables h is accurately reflected in the current value of the shadow variable H. As an example, command result of the initial specification in Fig. 3 results in the following events.

[9] That is, it satisfies the Event-B proof obligations.

```
resultShadow
  when
    $pc1 = INIT1$
    $insync1 = \mathsf{true}$
  then
    $insync1 = \mathsf{false}$
    $H1 = \{vb \in H1 \mid \mathsf{AND}(a \mapsto vb) = \mathsf{AND}(a \mapsto b)\}$
  end
```

```
result
  when
    $pc1 = INIT1$
    $insync1 = \mathsf{false}$
  then
    $pc1 = RESULT1$
    $insync1 = \mathsf{true}$
    $v := \mathsf{AND}(a \mapsto b)$
  end
```

In general, update to the Shadow variable is required in two cases: (a) when there is an assignment updating a visible variable based on values of hidden variables (e.g. the above `result` command of the initial specification), and (b) when there is an assignment to a hidden variable. Examples of (b) is the `resultB` commands in the lovers' protocol where the hidden variable `bP` is updated. Consider the `resultB` command of the first refinement machine, the corresponding shadow event is as follows.

```
resultBshadow
  when
    $pc2 = RESULTA2$
    $insync2 = \mathsf{true}$
  then
    $insync2 = \mathsf{false}$
    $H2 := \left\{ vb, vbP, new \cdot \begin{matrix} vb \mapsto vbP \in H2 \ \land \\ new = \mathsf{XXOR}(\mathsf{AND}(a \mapsto vb) \mapsto aP) \end{matrix} \mid vb \mapsto new \right\}$
  end
```

The action of resultBshadow updates the `bP` component of the Shadow set (represented by `vbP`) to reflect the effect of the first action. Note that the update computes the new value of each `bP` shadow value from `a` and `aP` (since they are visible) and from the relevant value of `b` in the Shadow set (since it is hidden). The latter is represented by the `vb` component.

An earlier version of the tool combined the shadow and functional events, resulted in a simpler Event-B machine without the need for *insync* variable. However, that approach is only sufficient for some simple protocols. Section 6.2 provides an in-depth discussion of the necessity for separation of the events for more complex protocols.

The refinement relationship of the functional events (those update which protocol variables) is the same as specified by the label of the protocol command. For example, translated event result of the initial specification in Fig. 3 is refined by translated event with the same label of the first refinement.

By default, the refinement relationship of shadow events (those which update the shadow variables) is also consistent with the label of the command. In general, it could be (and should be) more elaborated, as illustrated later in Sect. 6.2, allowing the refinement of shadow events to be decoupled from the refinement of the functional events.

We also generate strongest post-condition invariants that express the effects of the events.[10] For example, in the specification of the lovers' protocol, the `result` command gives rise the following invariant.[11]

```
pc1 = RESULT1 ⇒
      (∃ old1v · (v = AND(a ↦ b)) ∧
      ((a ∈ X) ∧ (old1v ∈ X) ∧ (b ∈ X)))
```

[10] The current tool assumes single-assignment programs, where each protocol variable is assigned at most once.
[11] To keep the examples simple, we omit the parts of the invariant that specify that other variables are unchanged.

5.3. Shadow invariants and refinement

Apart from a basic typing invariant, each Shadow set gets (a) an invariant stating that when the model is synchronised the actual values of the hidden variables are represented in the set—i.e. that each state is *valid*, and (b) a coupling invariant to relate the Shadow set to that of the machine that it is refining. For example, the invariants for the Shadow set of the first lovers' protocol refinement machine are:

```
insync2 = TRUE ⇒ b ↦ bP ∈ H2

∀ vb · vb ∈ H1 ⇒ (∃ vbP · vb ↦ vbP ∈ H2)
```

where the second condition states that the projection of H2 onto the global variables (i.e. those variables which have not been newly introduced in the proposed implementation) is a superset of H1 (the shadow variable of the initial specification). Essentially, invariants of the form (b) relating the shadow variables of two different machine encode the requirement for shadow refinement, i.e., the shadow cannot be reduced (with respect to projection to the common hidden variables).

The tool also generates post-condition invariants that express the effects of the shadow set updates at each step, and upper and lower bounds for the shadow. We omit the details since these invariants are similar to the post-conditions generated for the other events, as described in the previous section.

6. Case studies

In this section we describe two cases studies using our Rodin-based implementation of the Shadow Semantics. We briefly describe the purpose of each case study and highlight the challenges we encountered during the automation exercise.

Both examples satisfy the restrictions set out in Sect. 4.2 which means that this automation represents a full probabilistic analysis.

6.1. Private Information Retrieval

Private Information Retrieval (PIR) concerns a user who wishes to privately consult a public database to retrieve the value F(x) of her request x, i.e. she requires that her request remains private to herself.

Chor et al. [CGKS99] proposed a scheme which used $n \geq 2$ copies of the database that may not collude with each other. A user decomposes her request x into n secret shares, x1,...,xn, so that the request may be reconstructed from the shares, i.e. x = XOR(x1,...,xn) for some function XOR; but each share on its own reveals no information about the actual request. Assuming that the function F distributes over the recomposition of the shares so that F(XOR (x1,...,xn)) equals XOR(F(x0),...,F(xn)), the user then recovers her actual request by combining the lookups of the individual shares. Chor et al. [CGKS99] showed that the user's security requirements are met for $n \geq 2$, and proved results about message complexity. Since we are only interested in the security aspects, we only develop the case for $n = 2$.

McIver [McI09] produced a hand proof of the security of this scheme using the Shadow Semantics. In Fig. 4 we illustrate a derivation of PIR for $n = 2$, that has now been fully automated.

The initial specification for PIR (Fig. 4) contains one command

```
result: y = F(x)
```

in which the result of the database request x is calculated in secret and stored in hidden variable y.

Our refinement below introduces multiple (copies of) databases, together with a protocol for accessing them. What the refinement guarantees is that any information flowing as a result of the protocol *is already accounted for* in this single-command specification. This relationship is preserved in all contexts. This specification is then refined (in a stepwise fashion) so that we first split x into two separate requests xA and xB:

```
splitX: (xA XXOR xB) = x
```

Request xA is then sent to database A, which calculates the result of the request and returns it:

```
transferRequestA: a = xA;
computeA:         zA = F(a);
resultA:          yA = zA;
```

```
HID x : X, y: Y
FUN F : X -> Y

result: y = F(x);  "Consults the database F without revealing x"

[=

HID xB : X
VIS xA : X
FUN XXOR : X x X -> X
FUN YXOR : Y x Y -> Y

splitX: (xA XXOR xB) = x;         "Splits x into two shares, using the database
result: y = F(xA) YXOR F(xB);       property:   F(xA) YXOR F(xB) = F(xA XXOR xB)"

[=

HID yB : Y
VIS yA : Y

splitX:  (xA XXOR xB) = x;
resultA: yA = F(xA);
resultB: yB = F(xB);
result:  y = yA YXOR yB;

[=

HID zB : Y
VIS zA : Y

splitX:     (xA XXOR xB) = x;
computeA:   zA = F(xA);
resultA:    yA = zA;
computeB:   zB = F(xB);
resultB:    yB = zB;
result:     y = yA YXOR yB;

[=

HID b : X
VIS a : X

splitX:              (xA XXOR xB) = x;       "Introduces separate commands:
transferRequestA:    a = xA;
computeA:            zA = F(a);                      ...consults database A...
resultA:             yA = zA;
transferRequestB:    b = xB;
computeB:            zB = F(xB);                     ...consults database B...
resultB:             yB = zB;
result:              y = yA YXOR yB;                 ...combines the results."
```

Fig. 4. A derivation of Private Information Retrieval expressed in our security language, consisting of a protocol specification and four refinements. Axioms have been omitted

and the other request is sent to database B for calculation in a similar fashion. Finally, the database responses are composed to form an answer to the original request:

`result: y = yA YXOR yB`

The protocol as a whole is modelled from database A's perspective, and so variables `a`, `zA` and `yA` are declared to be visible, while the others are hidden.

Of course in order to prove this we need to add the following axiom regarding distributivity of the function `F`:

$\forall$ `xA, xB` $\cdot$ `YXOR(F(xA)` $\mapsto$ `F(xB)) = F(XXOR(xA`$\mapsto$ `xB))`

The tool generates an Event-B model our of the program in Fig. 4 written in our security language. The resulting model is then imported and analysed by Rodin. All generated proof obligations are successfully discharged (automatically and manually).

6.2. The Dining Cryptographers

The *Dining Cryptographers* (*DC*) is a protocol proposed by Chaum [Cha88] concerning a group of agents who want to perform certain computation together, while preserving their anonymity. The original formulation of the problem is as follows. Three cryptographers dining together call the waiter over to ask for the bill only to be informed that it has already been settled. It transpires that the payer can only be one of the cryptographers themselves (who wishes to remain anonymous) or the NSA. Being cryptographers they agree on a protocol which will reveal *only* whether a cryptographer, or the NSA paid, and in the former case *will not* reveal the identity of the payer to the other two (non-paying) cryptographers.

The protocol they agree to carry out is as follows. Each pair of cryptographers tosses a (shared) coin so that the result is only revealed to the pair who share it, and not the remaining cryptographer. Then each cryptographer announces the exclusive-or of the two coins he sees together with his "did-he-pay" bit. Finally, the exclusive-or of the announcements—which are visible to everyone—is announced. If it is false then the NSA paid, and if it is true then one of the cryptographers paid, but only the payer knows for sure which one of them it is.

In Fig. 5 we illustrate a derivation of Chaum's *Dining Cryptographers (DC)* in our security language. We model the algorithm according to the view of the waiter. The initial specification contains three hidden Boolean variables `a`, `b`, `c` representing whether the named cryptographer paid or not. It is an assumption that at most one of `a`, `b`, `c` is set to True. The result of the protocol is modelled as a visible variable `s`, revealing if one of the cryptographer is paying, i.e. the exclusive-or of the three bits. Our initial model specifies this action in a single (atomic) labelled command `result`

```
result: s = (a xor b xor c);
```

which simply reveals the combined exclusive-or of the three variables `a`, `b`, `c`. Because of the assumption that at most one of them is True, the final value of `s` reveals exactly whether a cryptographer or the NSA paid, and no other information.

The refinement modelling Chaum's algorithm has three additional hidden variables `ab`, `bc` and `ca`, representing the values of the coin tosses between pairs of cryptographers. There are also three additional visible variables `sa`, `sb` and `sc`, corresponding to the announced value of the cryptographers. First, each cryptographer announces the exclusive-or of the coins that he sees and did-he-pay. For cryptographer A, this is modelled as follows.

```
announceA: sa = (ab xor ca xor a);
```

Abstract command `result` is refined accordingly using the value of the visible variables.

```
result: s = (sa xor sb xor sc);
```

Formulating this problem in Event-B presented some challenges. As noted in Sect. 5, we translate each command into two events, an approach that was motivated by this case study. A single event approach fails because any new command required for a refinement step is translated to a new event, i.e. refining `skip`. For the *Dining Cryptographers (DC)* algorithm, an attempt to prove that `announceC` refines `skip` fails. This seems at first to be unreasonable, since functionally, `announceC` does not change the value of the original abstract variables `a`, `b` and `c`. However, a closer look at `announceC` reveals that this (not refining `skip`) is indeed the case. The fact is that after the last cryptographer (here `C`) makes his announcement, some information about the value of hidden values `a`, `b` and `c` is "leaked": their exclusive-or. As a result, `announceC`, which reveals the exclusive-or, is not the same as "do nothing", which reveals nothing.

The key point is that refinement of protocol variables and Shadow variables are treated differently. On the one hand, functionally, we need prove that the concrete `result` refines the abstract event `result`, i.e. when value of `s` are set. On the other hand, for the refinement of the Shadow variables, the concrete `announceC` must be "glued to" the abstract `result` command.

```
VIS s : BOOL
CONSTANT HID a : BOOL, b : BOOL, c : BOOL

result: s = (a xor b xor c);

[=

VIS sa : BOOL, sb: BOOL, sc: BOOL
HID ab : BOOL, bc : BOOL, ca : BOOL

SHADOW REFINEMENT
  result <: announceC

announceA: sa = (ab xor ca xor a);
announceB: sb = (bc xor ab xor b);
announceC: sc = (ca xor bc xor c);
result: s = (sa xor sb xor sc);
```

Fig. 5. A development of the dining cryptographers expressed in our security language as a specification, followed by a refinement

Based on this analysis, in the latest version of our tool we adopt a modelling style in which a command in our security language is modelled as two separate events, scheduled sequentially. The first event updates the shadow variable and the second event functionally updates the ordinary variables. As a result, consider the abstract command `result`

```
result: s = (a xor b xor c);
```

which is translated into the following two events.

```
resultShadow
  when
    $pc1 = INIT\Delta$
    $insync\Delta = \mathsf{true}$
  then
    $insync\Delta := \mathsf{false}$
    $H1 := \{(va, vb, vc) \in H1 \mid va \oplus vb \oplus vc = a \oplus b \oplus c\}$
  end
```

```
result
  when
    $pc1 = INIT\Delta$
    $insync\Delta = \mathsf{false}$
  then
    $pc1 := RESULT\Delta$
    $insync\Delta := \mathsf{true}$
    $\mathsf{s} := a \oplus b \oplus c$
  end
```

The concrete commands are translated into pairs of events in similar fashion, in particular `announceC` is translated into events announceCShadow and announceC. The refinement relationship between the shadow update events is declared in Fig. 5 by the `SHADOW REFINEMENT` declaration; otherwise, the refinement would be between the two result shadow update events. We can now prove that announceCShadow is a refinement of the abstract resultShadow, and the concrete event result refines its corresponding abstract event.

6.3. Tool performance

Lover's protocol Rodin generates 148 proof obligations for the lovers' protocol machines, with 71 relating to maintenance of invariants. 31 of the proof obligations (21%) cannot be proven automatically. All of those can be proved easily with manual assistance.

PIR As for PIR, Rodin is able to generate the proof obligations from the output of our tool. In this case there are 449 obligations in total, of which 43 (9.6%) are not discharged automatically. They can be proven with manual assistance.

Dining Cryptographers For the development in Event-B of the Dining Cryptographers, Rodin generates a total 246 proof obligations, amongst them 228 (92.5%) are discharged automatically by the built-in provers. The other 18 (7.5%) remaining obligations are proved interactively within the Rodin platform, most of them by applying/instantiating the right theorems about exclusive-or $\oplus$.

Even though the above examples use simple mathematical operators, some proof obligations still required manual assistance. The difficulty comes from the fact that for each Event-B machine, there is a single shadow variable for all hidden variables. Updating the shadow variable (necessarily) is complicated, resulting in complex proof obligations. One should expect that having more hidden variables will lead to more difficult proof obligations related to the shadow refinement. Another reason is the shadow refinement condition is captured by an invariant involving existential quantification. As a result, proving the maintenance of this invariant and other related supporting invariants involves instantiations, which is known to be a hard problem for automated provers. However, we see these challenges as the difficulty of the problem itself rather than a limitation of our approach and tool support.

7. Related work

The work reported here is an amalgam of a number of themes which we review briefly in this section.

Relation to other semantic approaches, both qualitative and quantitative.

The Shadow refinements implemented in Rodin are based on Morgan's original work [Mor06] and ultimately follow in the tradition of Mantel [Man01], Bossi et al. [BFPR03] and Leino and Joshi [LJ00]. It is different however in its framework. Mantel and Bossi for example take an unwinding-style approach to non-interference [GM84] in which the focus is on events and their possible occurrences. Relations between possible traces express what can or cannot be deduced, about a trace that includes high-security events, based on projections of it from which the high-security events have been erased. Morgan's Shadow semantics uses in contrast the Shadow sets as a kind of "digest" (or quotient) capturing the residual uncertainty of the hidden variable which remains after taking runtime observations and program structure into account. Unlike Leino and Joshi, the Shadow makes judgements over final- rather than initial values.

There are many formulations of probabilistic noninterference in the literature, but ours is closest to that of Backes [BP02] although his techniques are formulated rather differently. Indeed Pfitzmann, Waidner and Backes [BP04, PW01] have combined the notion of simulation with concurrency, channels, probability and computational-based cryptography. This use of simulation appears to be very similar to our notion of quantitative refinement in the quantitative Shadow [MMM10], and an interesting topic of future research is to investigate how these notions are related.

Our work in the quantitative Shadow has similar goals to language-based security (and its probabilistic variants) [SM03] in that our specifications describe the secrecy of variables within a programming language. In this paper however we sought to apply the quantitative semantics using the simpler reasoning tools available in qualitative models.

Comparison based approaches (simulation and refinement)

Other researchers have also used comparison-based approaches to reason about computer security. For example, in cryptography there are *simulation* relations for which—coarsely speaking—a concrete system correctly implements an abstract specification if certain "views" of the implementation are indistinguishable from the same views of the specification. Early formulations of this appear e.g. in work by Goldwasser and Levin [GL90], Micali and Rogaway [MR91] and Beaver [Bea91].

Other researchers have considered the notions of pessimistic- and probabilistic refinements separately. For example Hutter [Hut06] extends the possibilistic ideas of Mantel to action systems. In particular Hutter determines predicates which are preserved by action refinement; such predicates are similar to the Shadow sets which are designed to distinguish between nondeterminism due to observable variation and nondeterminism capturing lack of knowledge. Santen [San08] also considers probabilistic properties and studies information preserving refinements within a context of Shannon entropy formalised using a version of probabilistic CSP. The compositionality of our full quantitative refinement ensures the preservation of all quantitative security properties, including Bayes Vulnerability and Shannon entropy [MMM10].

The comparative view of security is taken also in other early, definitive works on cryptography, for which Abadi and Rogoway provide a summary [AR00], and in particular in the computational-complexity approach to cryptography [Gol02]. Goldreich writes "The transformation is … accompanied by an explicit reduction of the violation of the security of the complex phenomenon to the violation of the simpler one" [Gol10].

In program construction, comparison-based methods are framed in terms of abstraction and refinement, with a key property being compositionality. This property is fundamental to our approach, which although not stressed here, has been used to simplify significantly the derivations of protocols. An illustration of this can be found elsewhere [MMM10] where the specification of the oblivious transfer was used in place of its implementation in a larger protocol which used it as a primitive.

In a similar manner Universal Composability proposed by Canetti [Can01] sets out a framework for compositionality in terms of cryptography, and thus is similar in goal to our approach, but differs in the precise notions of security used. In particular our notion of security is an abstraction rather than being based directly on cryptography.

Automatic methods in the analysis of information flow

The literature here is broad, and we provide some illustrative examples.

There are many automated methods for security analysis based on qualitative aspects of security, often where the strength of the underlying cryptography is characterised through an abstract notion of an attacker's ability to interact with encrypted messages e.g. as described by Dolev and Yao [DY83]. For example Paulson provides and early demonstration of security proofs [Pau98] which uses this approach. Similarly Blanchet [Bla01] uses the Dolev-Yao framework and the tool ProVerif [Pro] develops and extends this approach and has been used to analyse many security examples.

The investigation of secrecy in terms of information flow is less well developed. Recent work includes that of Heusser and Malacaria [HM10] whose automated technique is based on measuring information flow in terms of its Shannon entropy. Andrés et al. [APRS10] similarly consider efficient calculation of information leakage, which can provide diagnostic feedback to the designer.

In contrast our use of Rodin uses Shadow refinement as the basis for ensuring that secrecy properties are met rather than specifying a particular security property; our attacker is passive, but in the quantitative model it can assess likelihoods of security breaches in formulating his opinion as to the value of the hidden state.

8. Discussion and conclusions

We have investigated the relationship between two models of security-based refinement: one quantitative and one qualitative. In particular we found that by combining a qualitative semantic analysis with a syntactic restriction we are able to verify quantitative properties for a class of programs, which include a number of published security protocols. A second contribution is to demonstrate automation of the semantic proofs using the Rodin toolkit.

Our strategic motivation for this study is to explore the practicality of using a refinement-style development of security properties. The framework given here is an attempt to provide stronger guarantees whose severity is measurable but without the need to actually measure. We originally chose Rodin as the implementation platform because it fits well with our notion of security refinement; it then became a mechanism to explore designs of protocols rather than as a means to provide actual working code.

In terms of the value of our proofs, that will be determined by how accurate are the abstractions we used and the assumptions about the attack model—at this stage our attacker is merely passive. For our pre-uniform assumption, we are able to show that the uniform property over the hidden state is not disturbed; the actual behaviour of the protocol then relies on the quality of the random source over which to choose the encryption bits.

As noted above our restrictions on the language are imposed by our goal of recovering the quantitative semantics, although the refinements we are able to establish are valid in more general contexts realised by, for example, looping behaviour. That is because the semantic refinements that we prove are aligned with the general definition of Shadow refinement [Mor06] and quantitative secure refinement [MMM10] for which general looping behaviour is considered [MM11, MMM11]. Of course if the severity of the security leak (in terms of estimating its chance of occurring) is not an issue then the pure Shadow semantics can be used whose language restrictions are minimal. In future work we will investigate other properties of our restricted language, in particular whether it can be optimised as a domain specific modelling language supported by automated verification.

References

[ABH$^+$10] Abrial J-R, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6):447–466

[Abr96] Abrial J-R (1996) The B Book: assigning programs to meanings. Cambridge University Press, London

[APRS10] Andrés ME, Palamidessi C, Van Rossum P, Smith G (2010) Computing the leakage of information-hiding systems. In: Tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 6015. Springer, Berlin, pp 373–389

[AR00] Abadi M, Rogoway P (2000) Reconciling two views of cryptography (the computational soundness of formal encryption). In: Proceedings of IFIP International Conference on Theoretical Computer Science. LNCS, vol 1872. Springer, Berlin, pp 3–22

[Bea91] Beaver D (1991) Foundations of secure interactive computing. In: Feigenbaum J (ed) CRYPTO '91. LNCS, vol 576. Springer, Berlin, pp 377–391

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

[Bla01] Bruno Blanchet (2001) An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In 14th IEEE Computer Security Foundations Workshop (CSFW-14), IEEE Computer Society, pp 82–96.

[BP02] Backes M, Pfitzmann B (2002) Computational probabilistic non-interference. In: 7th European Symposium on Research in Computer Security. LNCS, vol 2502, pp 1–23

[BP04] Backes M, Pfitzmann B (2004) Computational probabilistic noninterference. Int J Inf Secur 3(1):42–60

[BR93] Bellare M, Rogoway P (1993) Random oracles are practical: a paradigm for designing efficient protocols. In: ACM Conference on Computer and Communications Security, pp 62–73

[BvW98] Back R-JR, von Wright J (1998) Refinement calculus: a systematic introduction. Springer, Berlin

[Can01] Canetti R (2001) Universal composable security: a new paradigm for cryptographic protocols. In: Extended abstract appeared in proceedings of the 42nd Symposium on Foundations of Computer Science (FOCS), pp 136–145

[CGKS99] Chor B, Goldreich O, Kushilevitz E, Sudan M (1999) Private information retrieval. J ACM 45(6):965–982

[Cha88] Chaum D (1988) The dining cryptographers problem: unconditional sender and recipient untraceability. J Cryptol 1(1):65–75

[DY83] Dolev D, Yao A (1983) On the security of public key protocols. IEEE Trans Inf Theory 29(2):198–208

[GL90] Goldwasser S, Levin LA (1990) Fair computation of general functions in presence of immoral majority. In: Menezes A, Vanstone SA (eds) CRYPTO '90. LNCS, vol 537. Springer, Berlin, pp 77–93

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

[Gol02] Goldwasser S (2002) Mathematical foundations of modern cryptography: computational complexity perspective. In: Proceedings of the ICM, Beijing, vol 1, pp 245–272

[Gol10] O Goldreich (2010) Studies in complexity and cryptography. In: Security preserving reductions—revised terminology. LNCS, vol 6650. Springer, Berlin

[Gro] Probabilistic Systems Group. Collected publications. http://www.cse.unsw.edu.au/~carrollm/probs

[GW86] Grimmett GR, Welsh D (1986) Probability: an introduction. Oxford Science Publications, UK

[HM10] Heusser GR, Malacaria P (2010) Applied quantitative information flow and statistical databases. In: Formal aspects in security and trust. LNCS, vol 5983. Springer, Berlin, pp 96–110

[Hut06] Hutter D (2006) Possibilistic information flow control in MAKS and action refinement. In: Proceedings of the 2006 international conference on Emerging Trends in Information and Communication Security

[KB07] Köpf B, Basin D (2007) An information-theoretic model for adaptive side-channel attacks. In: Proceedings of 14th ACM Conference on Computer and Communication Security

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

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

[McI09] McIver AK (2009) The secret art of computer programming. In: Proceedings of ICTAC 2009. LNCS, vol 5684, pp 61–78 (invited presentation)

[MM09] McIver AK, Morgan CC (2009) The thousand-and-one cryptographers. At [Gro, `McIver:10web`]; includes appendices., April

[MM11] McIver AK, Morgan CC (2011) Compositional refinement in agent-based security protocols. Formal Aspects Comput 23(6):711–737

[MMM10] McIver A, Meinicke L, Morgan C (2010) Compositional closure for Bayes Risk in probabilistic noninterference. In: Proceedings of the 37th international colloquium conference on Automata, languages and programming: Part II, ICALP'10. Springer, Berlin, pp 223–235

[MMM11] McIver A, Meinicke L, Morgan C (2011) Hidden-markov program algebra with iteration. Math Struct Comput Sci (to appear)

[Mor87] Morris JM (1987) A theoretical basis for stepwise refinement and the programming calculus. Sci Comput Program 9(3):287–306

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

[Mor06] Morgan CC (2006) The Shadow Knows: refinement of ignorance in sequential programs. In: Uustalu T (ed), Math Prog Construction, vol 4014. Springer, Berlin, pages 359–378 (Treats Dining Cryptographers)

[Mor09] Morgan CC (2009) The Shadow Knows: refinement of ignorance in sequential programs. Sci Comput Program 74(8):629–653 (Treats Oblivious Transfer)

[MR91] Micali S, Rogaway P (1991) Secure computation (abstract). In: Feigenbaum J (ed) CRYPTO '91. LNCS, vol 576. Springer, Berlin, pp 392–404

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

[Pro] Proverif: Cryptographic protocol verifier in the formal model. http://www.proverif.ens.fr/

[PW01] Pfitzmann B, Waidner M (2001) A model for asynchronous reactive systems and its application to secure message transmission. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, pp 184–200

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

[RCH04] Goldreich O, Canetti R, Halevi S (2004) The random oracle methodology, revisited. JACM 51(4):557–594

[San08] Santen T (2008) Preservation of probabilistic information flow under refinement. Inf Comput 206(2–4):213–249

[Sha48] Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27:379–423, 623–656

[SM03] Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE J Sel Areas Commun 21(1):5–19

[Smi07] Smith G (2007) Adversaries and information leaks (Tutorial). In: Barthe G, Fournet C (eds) Proceedings of 3rd Symposium on Trustworthy Global Computing. LNCS, vol 4912. Springer, Berlin, pp 383–400

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

Received 14 October 2011
Revised 11 March 2012
Accepted 20 April 2012 by Eerke Boiten
Published online 13 June 2012