Program Refinement, Perfect Secrecy and Information Flow

Annabelle K. McIver(✉)

Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
annabelle.mciver@mq.edu.au

Abstract. "Classical" proofs of secure systems are based on reducing the hardness of one problem (defined by the protocol) to that of another (a known difficult computational problem). In standard program development [1,3,14] this "comparative approach" features in stepwise refinement: describe a system as simply as possible so that it has exactly the required properties and then apply sound refinement rules to obtain an implementation comprising specific algorithms and data-structures.

More recently the stepwise refinement method has been extended to include "information flow" properties as well as functional properties, thus supporting proofs about secrecy within a program refinement method.

In this paper we review the security-by-refinement approach and illustrate how it can be used to give an elementary treatment of some well known security principles.

Keywords: Proofs of security · Program semantics · Compositional security · Refinement of ignorance

1 Introduction

The challenge of designing secure programs is controlling information in such a way that program execution achieves something useful without, in the process, divulging secrets. *Provable security* means that there is a sound mathematical argument demonstrating that executing a program incurs no such security breach. A crucial first step in provable security is to specify what the secrets are and how they may be accessed. But specifying security properties accurately is extremely difficult because it requires the specifier to be very precise about often informally understood but subtle concepts. Such concepts can often seem counterintuitive without a lot of experimentation and a high degree of proficiency in specialised logics.

More recently researchers have explored an alternative approach to security verification based on a new notion of "refinement of ignorance" first described in

A.K. McIver — We acknowledge the support of the Australian Research Council Grant DP140101119.

Z. Liu and Z. Zhang (Eds.): SETSS 2014, LNCS 9506, pp. 80–102, 2016.
DOI: 10.1007/978-3-319-29628-9_2

Morgan's *Shadow semantics* [16]. Instead of proving rigorously that a given program "is secure", the analysis becomes comparative: we say "this program is more secure than that one", where the semantics provides the means to prove that such is the case. The refinement viewpoint focusses the technical scrutiny on information and access control, important ideas introduced by Denning [4], and later studied by Landauer [6] who suggested a lattice of information to provide the mathematical structure for comparing security of some deterministic programs.

The Shadow semantics is a significant generalisation of Landauer's work. It allows secret information to be updated, as well as supporting nondeterminism in the sense of underspecification. Those capabilities mean that protocols can be described very succinctly in terms of how information must change to achieve something useful, and what information must necessarily leak (and thus be "downgraded") in order to affect that change. In such a description, a security specialist can reflect on whether the security risk incurred by any unavoidable information leaks is balanced by the benefits the protocol brings to the particular application.

Although, in some sense, the Shadow semantics is unrealistic because it adopts an information theoretic approach to security, its clean treatment of information control provides a sound and straightforward basis on which to introduce fundamental principles of security. Teaching security principles to undergraduates in a formal way helps them to think critically and precisely about the control of information without them having to understand –at the same time– the complexities of cryptographic primitives used in real systems as an engineering device to implement those principles.

In this paper we review the Shadow semantics and use it to illustrate some well-known principles of secure communication. In particular we show how the refinement of ignorance approach supports straightforward algebraic proofs of intricate data access problems.

1.1 Notational Conventions

Throughout we use left-associating dot for function application, so that $f.x.y$ means $(f(x))(y)$ or $f(x,y)$, and we take (un-)Currying for granted where necessary. Comprehensions/quantifications are written uniformly, as $(\mathsf{Q}x\colon T|R\cdot E)$ for quantifier Q, bound variable(s) x of type(s) T, range-predicate R (probably) constraining x and element-constructor E in which x (probably) appears free: for sets the opening "$(\mathsf{Q}$" is "$\{$" and the closing "$)$" is "$\}$" so that e.g. the comprehension $\{x,y\colon\mathbb{N} \mid y{=}2^x \cdot yz\}$ is the set of numbers $z, 2z, 4z, \cdots$.

2 Principles of Non-interference

2.1 Review of the Shadow Semantics

The shadow model of security extends Goguen's classical model of non-interference security [5] by tracking the effect of observed information flows on correlations between variables. As with established methods for analysing information flow, the Shadow semantics is based on a partitioning of the state space into *high* and *low* security variables: an observer has full read access only of the low-security variables, and cannot read the high-security variables at all. The Shadow semantics is sensitive to "run-time" observations, such as the values of low-security variables and the program counter. This allows the attacker to infer *possible* values of the high-security variables even without the benefit of direct observation. These run-time observations of the low-security variables can be used in conjunction with a static analysis of the program source, possibly resulting in very accurate predictions of the values of high-security variables. We set out, more precisely, the threat model below at Sect. 2.3, but for now we give an informal description of the underlying principles, noting here that a fundamental feature of their design is to ensure *compositionality* of the related security-refinement order, also explained below.

We use types $\mathcal{V}$ and $\mathcal{H}$ respectively to distinguish between "visible" (i.e. low-security) variables and "hidden" (i.e. high-security) variables mentioned above. In traditional non-interference security, if the attacker cannot infer anything at all about the hidden variables by observing the visible variables then the program is deemed "Goguen-secure". Although an influential idea, this notion of security is normally not achievable in practical security protocols. This can be seen clearly in a password checker: if the correct password is entered then the observer deduces exactly what the password is, but even if he enters the incorrect password he learns what the password is not. Either outcome *necessarily* leaks some information.

The Shadow semantics provides support for analysing the extent to which an attacker can deduce the value of the hidden state. A program can still be deemed secure provided that the information revealed does not compromise a *specified level of secrecy*. In the password checker, leaking what the password is not is deemed an acceptable risk associated with the convenience of secure access control by password.

Consider the two programs set out at Fig. 1. Both have a single hidden variable h which is initialised to a value drawn from the set $\{0, 1, 2\}$, and a visible variable v initialised to a value drawn from $\{0, 1\}$. In standard program semantics we would be able to say that *ProgA* is refined by *ProgB*, since in *ProgB*, v's value is determined by that of h, whereas in *ProgA*, v's value is chosen arbitrarily in $\{0, 1\}$. However, in terms of non-interference, we will see that *ProgB* is actually *less secure* than *ProgA*, since it potentially can leak quite a lot of information about h. Taking that into account would force us to conclude that *ProgA* is not refined by *ProgB* after all.

To compare information-flow characteristics of programs we use **hid** and **vis** respectively as *visibility* declarations: these determine semantically how the state is divided between $\mathcal{V}$ and $\mathcal{H}$ (introduced above). Variables with the **vis** declaration mean that the observer has full "read access" at runtime, and these variables are mapped to $\mathcal{V}$. On the other hand variables with the **hid** declaration mean that the observer cannot see runtime updates, but can only infer values based on the program source and the observed state changes of visible variables. Variables with the **hid** declaration are mapped to $\mathcal{H}$ in the semantics.

ProgA	*ProgB*
vis $v{:\in}\,\{0,1\}$;	**hid** $h{:\in}\,\{0,1,2\}$;
hid $h{:\in}\,\{0,1,2\}$	**vis** $v{:=}\,(h \bmod 2)$

A **vis** declaration means that the observer has full "read access" to v. A **hid** declaration means that he is not able to read directly the value of h.

Fig. 1. Similar output values but dissimilar information-flow characteristics

Now with these declarations, we can analyse the information about h leaked at run-time. We write $h{:\in}\,\{0,1,2\}$ to mean that the hidden variable h is set to any of the three possible values, but the observer's knowledge of the state cannot be any more precise than that. For visible variables the observer always has complete knowledge of the runtime values. When we write $v{:\in}\,\{0,1\}$, it means that from the source code alone the observer cannot predict which of 0 or 1 will be assigned to v; however he can observe at run-time exactly which value is selected. In particular *ProgA* of Fig. 1 is non-interference secure in the sense of Goguen, because whatever the run-time value of v is observed, the attacker is unable to use that information to determine the value of h more accurately than its initialisation set. An attacker observing *ProgB* on the other hand can deduce a great deal about the value of h by observing the run-time value of v. Since v's final value depends on the parity of h, if v is set to 1 it can only mean that h is also 1, since it is the only odd value in the set of values that h can have. If v is observed to be 0, then the attacker can rule out 1 as a possible value for h. Thus we can deduce that *ProgB* is not non-interference secure in the classical sense. However the Shadow semantics has given us a precise relationship between the visible state and the hidden state, rather than a single judgement of non-interference. This detailed relationship between the hidden state and visible behaviour can be used to compare the relative security of programs using a "security refinement" relation which takes both functional and non-interference security into account. We review this relationship next.

The basic "Shadow state" for programs is a pair (v, H) where v is the current state of the visible variables v, and H is a subset of *possible* values for the hidden state variable h that the attacker has deduced is consistent with his observations. Thus (v, H) should be thought of as pairing visible values together with "equivalence classes" of possible values for h which the attacker is able to infer from run-time observations and the source code.[1]

The Shadow semantics of a program is then a mapping from initial paired states to sets of paired states $\mathcal{V} \times \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathbb{P}\mathcal{H})$, where the multiplicity of the result sets accounts for nondeterminism in the observations. In Fig. 2 we can see the final result sets for our examples at Fig. 1. In the case of *ProgA* there are two possible observations, depending on the nondeterministic setting

[1] Another way to think of the pairs (v, H) are abstractions of "prior"/"posterior" distributions in a full probabilistic semantics. See for example [12,13].

of visible variable v. However for each observation, no information about the possible value of the hidden variable is leaked, thus each visible state is paired with the set $\{0, 1, 2\}$ denoting that the attacker is unable to make any run-time deductions about the value of the hidden h by comparing what he knows before with what he knows after executing and observing the program's behaviour. In the case of *ProgB*, significant information about h is leaked at run-time, and this is taken into account by the different subsets paired with the observations, which the attacker can take advantage of when the program executes. For example if the attacker observes that v is 1 at run-time then he can deduce precisely the value of h, thus the singleton subset $\{1\}$ is paired with that observed state.

Shadow refinement of programs remains consistent with standard functional refinement, but prevents refinements which lead to inconsistent security properties between the specification and the implementation. As mentioned above, a standard semantics of *ProgA* and *ProgB* (i.e. one that ignores the visibility declarations) would imply that *ProgA* is actually "functionally refined" by *ProgB*, because the nondeterminism (in the final values of v) has been reduced in *ProgB*. However the security properties of *ProgB* are *worse* than those of *ProgA*, since, as explained above, information about h is leaked to the attacker. Thus with respect to *Shadow refinement* *ProgA* and *ProgB* are unrelated.

Two programs *are* in the Shadow refinement relation if *both* their functional and security properties are improved. The Shadow semantics incorporates assumptions in the threat model to ensure that secure refinement is *compositional.* "Compositionality" means that security (and functional) properties of a program can be determined by the corresponding properties of its components. To ensure compositionality in the Shadow semantics the attacker must have the following two important capabilities.

The first is "perfect recall", which means that previous information flows are carried forward, and can be used to make additional deductions when combined with subsequent run-time information flows. For example consider $ProgB; v{:=}\,0$ in Fig. 4, where *ProgB* first leaks information about h, and then resets the value of v. We see that, in spite of overwriting the visible variable, the effect of the information flow is sustained in the semantics, so that although the visible variables have the same value in the end, the control flow of (the original) *ProgB*, which leaks information about h, is preserved by the result set.

The second capability relates to the consistency of the observer's deductive powers. We say that the observer *cannot* deduce facts that are inconsistent with his observations, but can deduce facts that are. In particular if the observations imply that a value taken by h *could* be possible, then it must be included in some output (v, H). Similarly if the observations imply that a value taken by h could *not* be possible then it must be excluded by all outputs (v, H). We define observation consistency as follows.

Definition 1 *(Observation consistency).* Given a set of observations $\mathcal{O}_{\mathsf{v}}$ associated with the same visible state v,

$$\mathcal{O}_{\mathsf{v}} := \{(\mathsf{v}, H_1), \ldots, (\mathsf{v}, H_k)\}\,,$$

we say that any observation (v, H') is *consistent with* $\mathcal{O}_{\mathsf{v}}$ if there is some subset $\mathcal{U} \subseteq \mathcal{O}_{\mathsf{v}}$ such that $H' = \bigcup_{(\mathsf{v},H)\in\mathcal{U}} H$. □

For example in $ProgB; v{:=}\,0$ the observer can deduce that $h \in \{0,1,2\}$, but can also deduce that $h \notin \{3,4,5\}$, thus $(0,\{0,1,2\})$ is consistent with all the observations, but $(0,\{0,1,2,3\})$ is not. This is why the semantics Definition 2 (below) mandates the inclusion of all consistent observations in the result set.

In fact consistency is related to *union closure* on the H-component: we say that a set of observations $\mathcal{O}$ is union-closed for v if whenever (v, H_1) and $(\mathsf{v}, H_2) \in \mathcal{O}$ then $(\mathsf{v}, H_1 \cup H_2) \in \mathcal{O}$ as well. If a set of observations is union-closed for all v then any observation that the attacker can deduce is consistent with all observations in the set.

Definition 2 *(Shadow semantics).* The space of *Shadow programs* is given by $\mathcal{V} \times \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathbb{P}\mathcal{H})$, where the result sets are union-closed for all v.[2]

Given two programs $P, Q : \mathcal{V} \times \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathbb{P}\mathcal{H})$ we say that P is *secure refined by* Q, or $P \sqsubseteq Q$, provided that for all $(\mathsf{v}, H) \in \mathcal{V} \times \mathbb{P}\mathcal{H}$ we have $P.(\mathsf{v}, H) \supseteq Q.(\mathsf{v}, H)$. □

Definition 2 we can now see that in fact $ProgA; v{:=}\,0$, with the nondeterminism in v now removed by the final assignment, is a refinement of $ProgB; v{:=}\,0$, i.e.

$$ProgB; v{:=}\,0 \sqsubseteq ProgA; v{:=}\,0\,.$$

The refinement tells us that any observation that the attacker can deduce about $ProgA; v{:=}\,0$ is something that can also be deduced about $ProgB; v{:=}\,0$. In other words, the attacker can deduce fewer properties about the secrets of $ProgA; v{:=}\,0$ than he can about the secrets of $ProgB; v{:=}\,0$.

2.2 Semantics of a Simple Programming Language

In Fig. 3 we set out the semantics of a small programming language for describing straight-line programs. The semantic brackets $[\![\cdot]\!]$ take a program text and map it to a function of type $\mathcal{V} \times \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathbb{P}\mathcal{H})$ as described in Definition 2.

The general principle for keeping track of the correlations between the observed behaviour of the program is to form "equivalence classes" of the hidden state consistent with the observables. The formulae given in Fig. 3 express the division into equivalence classes of the Shadow variables once the information flow and the state updates have been carried out. For example when the visible state is set according to an expression that depends on the hidden state, the

[2] For simplicity we do not consider the empty set, i.e. outputs of the form $(\mathsf{v}, \{\})$.

$\llbracket ProgA \rrbracket.(\mathsf{v}, H)$	$\llbracket ProgB \rrbracket.(\mathsf{v}, H)$
$\{\ (0, \{0,1,2\})\ ,$	$\{\ (0, \{0,2\})\ ,$
$(1, \{0,1,2\})\ \}$	$(1, \{1\})\ \}$

We use (v, H) to stand for an arbitrary initial paired state, and the semantic brackets $\llbracket\cdot\rrbracket$ maps program texts to functions $\mathcal{V} \times \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathbb{P}\mathcal{H})$; the subsets denote the final values of the paired states after executing the programs in Fig. 1.
Full details of the semantic function $\llbracket\cdot\rrbracket$ are given at Fig. 3.

Fig. 2. Final result sets

	Program P	Semantics $\llbracket P \rrbracket.(\mathsf{v}, H)$
Assign to visible	$v{:=}\,\phi(v,h)$	$\{\mathsf{v}' : \phi(\mathsf{v}, H) \cdot (\mathsf{v}',\ \ \{\mathsf{h}{:}\, H \mid \phi(\mathsf{v},\mathsf{h}) = \mathsf{v}' \cdot h\})\}$
Assign to hidden	$h{:=}\,\phi(v,h)$	$\{(\mathsf{v},\ \ \{\mathsf{h} : H \cdot \phi(\mathsf{v},\mathsf{h})\})\}$
Choose visible	$v{:\in}\, S.v.h$	$\{\mathsf{v}'{:}\, S(\mathsf{v}, H) \cdot\ (\mathsf{v}',\ \{\mathsf{h}'{:}\, H \mid \mathsf{v}' \in S.\mathsf{v}.\mathsf{h}' \cdot h'\})\ \}$
Choose hidden	$h{:\in}\, S.v.h$	$\{\mathsf{h}'{:}\, S(\mathsf{v}, H) \cdot\ (\mathsf{v},\ \{\mathsf{h}'{:}\, H; \mathsf{h}''{:}\, S.\mathsf{v}.\mathsf{h}' \cdot \mathsf{h}''\})\ \}$
Composition	$P_1; P_2$	$\mathsf{lift}.\llbracket P_2 \rrbracket.(\llbracket P_1 \rrbracket.v.h.H)$
Demonic choice	$P_1 \sqcap P_2$	$\llbracket P_1 \rrbracket.(\mathsf{v}, H) \cup \llbracket P_2 \rrbracket.(\mathsf{v}, H)$
Conditional	**if** $\phi(v,h)$ **then** P_t **else** P_f **fi**	$\llbracket P_t \rrbracket.(\mathsf{v},\ \{\mathsf{h}'{:}\, H \mid \phi(\mathsf{v},\mathsf{h}') = \mathbf{true} \cdot h'\})\ \ \bigcup$ $\llbracket P_f \rrbracket.(\mathsf{v},\ \{\mathsf{h}'{:}\, H \mid \phi(\mathsf{v},\mathsf{h}') = \mathbf{false} \cdot h'\})$

For an expression ϕ that formally is evaluated on variables v and h, on the right-hand-side, we write $\phi(\mathsf{v},\mathsf{h})$ for the value that is produced when v has value v and h has value h. Similarly, we write $\phi(\mathsf{v}, H)$ for the set of values $\{\mathsf{h} : H \cdot \phi(\mathsf{v},\mathsf{h})\}$ that could arise by varying over H.
The function $\mathsf{lift}.\llbracket P_2 \rrbracket$ applies $\llbracket P_2 \rrbracket$ to all paired states in its set-valued argument, un-Currying each time, and then takes the union of all results.
The extension to many variables $v_1, v_2, \cdots$ and $h_1, h_2, \cdots$, including local declarations, is straightforward [15, 16].

Fig. 3. Semantics of non-looping commands

original set H then becomes divided into equivalence classes depending on the visible value observed. In Figs. 1 and 2, *ProgB* illustrates this. A similar situation can occur when the hidden state is set according to the value of the visible state. For example $h{:=}\,v$ implies that after the assignment to h, the Shadow variable corresponds to a singleton.

The conditional always releases information depending on whether it resolves to **true** or not. We note also that non-determinism in the assignments to the visible variables can arise through information flows whenever the uncertainty in the hidden state becomes resolved through a definite observation. For example the statement $v{:=}\,(h \bmod 2)$ can lead to a non-deterministic setting of v when the incoming uncertainty of H allows the possibilities that h could be both odd and even.

$[\![ProgA; v{:=}\,0]\!].(\mathsf{v}, H)$	$[\![ProgB; v{:=}\,0]\!].(\mathsf{v}, H)$
$\{\ (0, \{0, 1, 2\})\ \}$	$\{\ (0, \{0, 2\})\ , (0, \{1\})\ ,$ $(0, \{0, 1, 2\})\ \}$

Setting the final value of v to 0 preserves the prior information leaks — in effect the attacker has perfect recall. Union-closure allows functionally equivalent programs with better security properties to be possible refinements.

Fig. 4. Perfect recall and refinement

2.3 Refinement of Ignorance

We can understand the secure refinement in Definition 2 in terms of increasing ignorance (of the observer) as programs become more refined. The more refined a program, the less definite is an observer's knowledge concerning the value of the hidden variables at run-time. We use a "possibility modality" to express degrees of observer ignorance. Let ϕ be an ordinary expression in the program variables. Given a paired state (v, H) we say that it *possibly satisfies* ϕ, or $(\mathsf{v}, H) \models P(\phi)$ provided that:

$$(\exists \mathsf{h} \in H \mid \phi(\mathsf{v}, \mathsf{h})), \tag{1}$$

where we write $\phi(\mathsf{v}, \mathsf{h})$ for ϕ with the free occurrences of v and h replaced by the corresponding values v and h.

Similarly, given a set S of paired states, we say $P(\phi)$ is satisfied for S provided that all (proper) paired states in S possibly satisfy ϕ, that is:

$$S \models P(\phi) \quad \textit{iff} \quad (\forall (\mathsf{v}, H) \in S \mid (\mathsf{v}, H) \models P(\phi))\,. \tag{2}$$

For example if ϕ is "*h is even*", then $(0, \{0, 1, 2\}) \models P(\phi)$, but $[\![ProgB]\!].(\mathsf{v}, H) \not\models P(\phi)$, since for $(1, \{1\}) \in [\![ProgB]\!].(\mathsf{v}, H)$ there is no possibility that the final value of h is even. Thus, if for some property ϕ, program *Prog* and initial state (v, H), if $[\![Prog]\!].(\mathsf{v}, H) \models P(\phi)$, then the attacker is unable to distinguish any of the outputs of *Prog* in respect of ϕ. This feature is preserved by refinement.

Lemma 1 (*Ignorance refinement [16]*). If $Prog \sqsubseteq Prog'$ then for any expression ϕ in the program variables, and initial state (v, H),

$$[\![Prog]\!].(\mathsf{v}, H) \models P(\phi) \quad \Rightarrow \quad [\![Prog']\!].(\mathsf{v}, H) \models P(\phi).$$

□

2.4 Summary of the Non-interference Threat Model

We end the review of the Shadow semantics by summarising the operational principles underlying the threat model.

An observer:

1. **Has complete knowledge of the source code.**
   This implies that the attacker can perform a static analysis to make informed predictions about possible correlations between the hidden and visible state.

2. **Is able to observe run-time control flow.**
   By making run-time observations the attacker is able to rule out some of the static predictions made at step 1, thus significantly improving his current knowledge of the secret.

3. **Has perfect recall.**
   This means that once information about the value of a hidden variable has leaked, there is no way to cover it up except by re-setting the value.

4. **Is able to make logical deductions.**
   This means that the attacker can put together all the basic facts from the analyses mentioned above, including static and run-time analysis, to draw additional logical facts about the secret data.

5. **Is unable to guess or prove the actual value of the hidden state** more precisely than that which is implied by the above forms of information flow.
   This is an important limitation on the accuracy of his knowledge of the secrecy: it says that if there is no *logical evidence* to suggest that the value of the secret *is not in* some set H, then the attacker cannot improve his guess further.

6. **Can use refinement in context.**
   If $S \sqsubseteq I$, then any property which the attacker can deduce about S can also be deduced about I. This last property imposes "compositionality" for reasoning, and makes secure refinement useful in practice.

3 The Reveal Statement

It is often useful to analyse exactly what the adversary can deduce at various stages of the program execution; to do this we introduce a program statement, additional to the language constructs at Fig. 3. The statement **reveal** ϕ, where ϕ is an expression in the program variables v and h, is an explicit publication of a value. It does so without changing any variables. Semantically it is equivalent to publishing the value $\phi(v, h)$ in a (local) visible variable.[3]

[3] Equivalently it can be done by publishing the value in any visible variable and then overwriting that variable with its former value, leaving perfect recall to retain the information revealed.

Definition 3 *(Reveal Statement).* Let ϕ be an expression involving program variables v and h. The program **reveal** ϕ publishes the value of ϕ based on the current actual value of the variables:

$$[\![\mathbf{reveal}\,\phi]\!].(\mathsf{v}, H) \quad := \quad \{\mathsf{G} : \mathcal{G} \cdot (\mathsf{v}, \quad \{\mathsf{h}{:}\, H \mid \phi(\mathsf{v},\mathsf{h}) = \mathsf{G} \cdot h\})\}\,,$$

where $\mathcal{G}$ is the set of distinct values taken by $\phi(\mathsf{v},\mathsf{h})$ as h ranges over possible values in H. □

We can use the reveal statement in two ways when analysing a program. The first is as a specification statement because it separates very clearly the overall effect of a program in terms of how variables are updated, and which facts about those variables are known to the attacker. For example, the program *ProgB*; $v{:=}\,0$ is equivalent to a more straightforward one that sets the variables and then publishes a property about the hidden state:

$$\begin{array}{l} \mathbf{hid}\ h{:\in}\,\{0,1,2\}; \\ \mathbf{vis}\ v{:=}\,0; \\ \mathbf{reveal}\ (h \mod 2). \end{array} \tag{3}$$

Second, we can use the **reveal** statement to prove information flow properties. Since the **reveal** statement is only concerned with information flow and not state updates, it satisfies a number of simple but powerful algebraic laws, and these can be used in conjunction with the programming language to deduce information flows about programs which do include state updates.

Theorem 1 *(Reveal properties [9, 10]).* Let *Prog*, *Prog′* be programs with visible variable v and hidden variable h, and let ϕ, ψ be expressions in those program variables. The following properties hold.

1. **reveal** $\phi \quad \sqsubseteq \quad$ **skip**;
2. **reveal** ϕ ; **reveal** $\psi \quad = \quad$ **reveal** ψ ; **reveal** ϕ;
3. $v{:=}\,\psi(v,h)$; **reveal** ϕ = **reveal** ϕ ; $v{:=}\,\psi(v,h)$, whenever ϕ does not depend on v;
4. $v{:=}\,\phi(v,h) \quad = \quad$ **reveal** ϕ ; $v{:=}\,\phi(v,h)$;
5. $v{:=}\,\phi(v,h)$; $v{:=}\,\psi(v,h) \quad = \quad$ **reveal** ϕ ; $v{:=}\,\psi(\phi(v,h),h)$; □

Theorem 1(1) says that only revealing information, without changing any variables, is always an anti-refinement of **skip**. Theorem 1(2) says that information can be revealed in any order. In fact we define a useful shorthand for multiple reveals:

$$\mathbf{reveal}\ (\psi,\phi) \quad := \quad \mathbf{reveal}\ \phi\ ;\ \mathbf{reveal}\ \psi. \tag{4}$$

Next, Theorem 1(3) says that revealing information about h that is independent of the visible state can occur before or after updates to the visible state.

Theorem 1(4) says that if an assignment to a visible variables already reveals information then making an explicit reveal of that information before the update is redundant.

Theorem 1(5) allows two updates to v to be combined provided the information flow of the first update is recorded in a reveal statement.

To illustrate reasoning with **reveal** we consider $\mathit{ProgB}; v{:=}\,0$:

$$
\begin{array}{lll}
 & \mathit{ProgB}; v{:=}\,0 & \\
= & \textbf{hid } h{:\in}\,\{0,1,2\}; \textbf{vis } v{:=}\,(h \bmod 2); v{:=}\,0 & \text{“Fig. 1”} \\
= & \textbf{hid } h{:\in}\,\{0,1,2\}; \textbf{reveal } (h \bmod 2)\ ; \textbf{vis } v{:=}\,0 & \text{“Thm. 1(5)”} \\
= & \textbf{hid } h{:\in}\,\{0,1,2\}; \textbf{vis } v{:=}\,0;\ \textbf{reveal } (h \bmod 2), & \text{“Thm. 1(3)”}
\end{array}
$$

demonstrating the asserted equality between (3) and $\mathit{ProgB}; v{:=}\,0$.

3.1 Reveal Statements as Tests for Information Leaks

Theorem 1(4) suggests a testing interpretation for reveal statements. If

$$\mathit{Prog}\ ;\ \textbf{reveal}\ \phi \quad = \quad \mathit{Prog},$$

then *Prog* already reveals a relationship between v and h through publication of ϕ, since following it by the explicit reveal incurs no further information leak. When two programs are functionally equivalent, but differ in their Shadow semantics it is because one reveals different information than the other. We can capture these differences using reveal statements.

Definition 4 *(Test for information leaks).* A *test for an information leak* is any program T such that $T \sqsubseteq$ **skip**. We say that it is possible that a program *Prog* can *leak* information expressed by T provided $\mathit{Prog}; T = \mathit{Prog}$. □

The program **skip** is a very weak test because all programs satisfy *Prog*; **skip** = *Prog*, and indeed **skip** expresses no relationship between observations and the hidden state. More demanding is the test given by **reveal** $(h \bmod 2)$ — programs whose results remain unchanged after this explicit reveal already leak the parity of h.

For example ProgB; **reveal** $(h \bmod 2) = \mathit{ProgB}$, but ProgA; **reveal** $(h \bmod 2) \neq \mathit{ProgA}$.

4 The Shadow Semantics and Information-Theoretic Security

In 1949 Claude Shannon [20] set out the principles and properties of encryption schemes which guarantee perfect secrecy. In this section we review those definitions in terms of the Shadow semantics, illustrating some of those principle's consequences using algebraic reasoning set out above.

Let E represent an “encryption mechanism” described by Shannon consisting of a mapping from $\mathcal{M}\times\mathcal{K} \to \mathcal{E}$, where $\mathcal{M}$ is the set of possible messages, $\mathcal{K}$ is the set of possible keys and $\mathcal{E}$ is the set of possible encryptions. For any key $k \in \mathcal{K}$ the mapping $m \mapsto E.m.k$ must be injective, so that any given encryption $E.m.k$ can be decrypted with key k. Shannon studied *information theoretic* security in which an attacker is accorded unlimited computational power; in such a model the notion of perfect secrecy means that if a secret is chosen uniformly at random from $\mathcal{M}$, and the key is chosen uniformly at random from $\mathcal{K}$, then the encryption $E.m.k$ is perfectly secure provided that whatever the value $E.m.k$ observed by the attacker, he cannot guess the message m with probability greater than $1/|\mathcal{M}|$.

Shannon proved an important limitation of perfect security: that the set of keys must be the same size as the set of messages.

Theorem 2 *(Perfect security [20]).* Let $E : \mathcal{M}\times\mathcal{K} \to \mathcal{E}$ be a perfectly secure encryption mechanism. Then the set of possible keys $\mathcal{K}$ must be at least as big as the set of possible messages $\mathcal{M}$. □

Theorem 2 suggests a definition of a “generic perfectly secure” encryption mechanism using the Shadow semantics: revealing $E.m.k$ leaks nothing about the precise value of m or k, if neither are known initially.

Definition 5 *(Perfect encryption).* Let $E : \mathcal{M}\times\mathcal{K} \to \mathcal{E}$ be an encryption mechanism, where $\mathcal{M}$ is set of possible messages, $\mathcal{K}$ is a set of encryption keys and $\mathcal{E}$ the set of possible encryptions. We say that E corresponds to a *perfect encryption mechanism* if:

$$\begin{array}{l} \textbf{hid}\ m{:}{\in}\,\mathcal{M};\\ \textbf{hid}\ k{:}{\in}\,\mathcal{K};\\ \textbf{reveal}\ E.m.k \end{array} \quad = \quad \begin{array}{l} \textbf{hid}\ m{:}{\in}\,\mathcal{M};\\ \textbf{hid}\ k{:}{\in}\,\mathcal{K} \end{array}$$

□

As an example, consider a “one time pad” implemented with the “exclusive-or” operator denoted $\oplus$. If we define $E : \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$ as $E.b.b' := b \oplus b'$ then it can be shown directly [11] that E satisfies Definition 5, validating the method of “masking” key bits with randomly chosen values.

4.1 The Encryption Lemma

We state Morgan’s encryption lemma for perfectly secure encryption schemes.

Lemma 2 *(Encryption Lemma [16]).* Let $E : \mathcal{M}\times\mathcal{K} \to \mathcal{E}$ correspond to a perfectly secure encryption mechanism as set out at Definition 5. If m is a hidden variable, and k is a fresh, hidden variable, then:

$$\begin{array}{l} \textbf{hid}\ k{:}{\in}\,\mathcal{K};\\ \textbf{reveal}\ E.m.k;\\ \textbf{skip}_{\mathcal{M}} \end{array} \quad = \quad \textbf{skip},$$

where $[\![\textbf{skip}_{\mathcal{M}}]\!].(\mathsf{v},\mathsf{H}) := \{(\mathsf{v},\mathsf{H}_{\mathcal{M}})\}$ and $\mathsf{H}_{\mathcal{M}}$ is the projection of $\mathsf{H} \subseteq \mathcal{M} \times \mathcal{K}$ onto $\mathcal{M}$. □

Although Lemma 2 is equivalent to Definition 5, stated in this way, it is a concise representation of how much information about h is released by publishing the result of the encryption. When E is a perfectly secure mechanism then it reveals no more information about h than was known before.

4.2 The Decryption Lemma

Shannon's analysis says that, even when the key and message sets are the same size, then for the mechanism to be perfectly secure it must be very like a one-time pad. The next theorem summarises this idea.

Theorem 3 *(Perfect secrecy [20]).* Let $E : \mathcal{M}\times\mathcal{K} \to \mathcal{E}$ be a perfectly secure encryption mechanism such that $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{E}|$. Then the following must hold:

1. Each key must be chosen with probability $1/|\mathcal{K}|$;
2. For every message $m \in \mathcal{M}$ and cipher text $\mathsf{E} \in \mathcal{E}$ there is a unique k such that $E.m.k = \mathsf{E}$. □

Given Theorem 3 we can now prove a property similar to Shannon's Theorem 3(2), saying that if the attacker has knowledge of both the secret and the cipher text, then he can deduce the key.

Lemma 3 *(Decryption Lemma).* Let $E : \mathcal{M}\times\mathcal{K} \to \mathcal{E}$ be a perfectly secure mechanism known to the observer such that $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{E}|$. Then

$$\textbf{reveal}\ (m, E.m.k) \quad = \quad \textbf{reveal}\ (m, k). \tag{5}$$

Proof: Theorem 3(2) states that for every $m \in \mathcal{M}$, if $E.m.k = E.m.k'$ then $k = k'$. Hence if the observer knows both m and $E.m.k$ for some k then he must be able to infer the value of k exactly. □

Lemma 3 expresses a second limitation of perfectly secure encryption systems, that both the cipher text and the message must be kept secret for the encryption not to leak anything.[4]

5 Modelling Protocols Between Multiple Agents

Most security protocols involve several participants, each with different security concerns, and each with different accessibilities to the information content within the system. The security proofs of such systems must ensure that each participant's (individual) specified security objectives are met. Thus far we have described a semantics which can handle a simple scenario of a single attacker; in this section we review how to adapt it so that we can describe security protocols with multiple participants, where we treat each participant as a potential observer of all the others.

[4] This abstract algebraic property also captures the capability of a "known plain text attack". An encryption mechanism is susceptible to a known plain-text attack if, given the message m and the encryption $E.m.k$ is able to work out the key k. If such an attack succeeds against an encryption method then we would say that the encryption mechanism satisfies (5).

5.1 Multiple Agents and Their Potential Attackers

A typical security protocol involves a communication between at least two "agents". Typically both agents have their own secret information which is private to them individually. The objective of a security protocol between the agents might be to share some secret information without leaking it to a third party, or to convince each other that they really are who they claim to be. To implement the protocol means sharing some part of their private data, but in a constrained manner so that they do not reveal too much.

The simplest example of such a "multi-party" protocol is publishing the conjunction of two secret bits, described informally as follows.

> Two agents A and B each have a private bit held in variables a and b respectively. They wish to compute the conjunction $a \wedge b$ without either one telling the other the actual value of their own bit, and without releasing any more information than can be deduced from knowing $a \wedge b$.

The objective of this protocol is to publish *only* the value $a \wedge b$, but without revealing any more than that, even to each other. This, of course, necessarily involves some information flow but, depending on the actual value of $a \wedge b$, some uncertainty can still remain. If the value of $a \wedge b$ is 1, then indeed publishing this will reveal exactly that $a = b = 1$. However if $a \wedge b$ is 0 then there are several different scenarios which could lead to this result: either both a and b are 0 or exactly one is 0. By publishing the value of $a \wedge b$ in this case, there should be no distinction between relevant scenarios from the point of view of any participating agent or third party observer. In particular if a is 0, then agent A should not know whether b is 0 or 1.

We adapt the simple semantics described above so that it can be used to take into account each "agent's viewpoint", and the individual constraints on information access required by protocols such as this.

Elsewhere we introduced agent viewpoints [11] into the system by embellishing the declaration of each variable to include a list of agents which treat the variable as an observable. In detail we use the following declarations to classify various explicit visibility relationships between agents.

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

For example in the case of the secure conjunction protocol, we can use the visibility declarations to include which agents can observe which parts of the state. The formal protocol is set out at Fig. 5. First A and B each choose (at random) a secret bit and save it to their respective private variables. Those variables are declared with a visibility that makes explicit that only A can observe a, and only B can observe b directly. Of course A and B each have a "copy" of the source code. Finally the universally visible variable c (visibility declaration **vis**) is set to the value $a \wedge b$.

$\mathbf{vis}_A\ a{:}{\in}\,\{0,1\};$ $\leftarrow$ *A chooses a secret bit*
$\mathbf{vis}_B\ b{:}{\in}\,\{0,1\};$ $\leftarrow$ *B chooses a secret bit*
$\mathbf{vis}\ c{:}{=}\,a \wedge b$ $\leftarrow$ *Their conjunction is published in c.*

$\mathbf{vis}_A$ and $\mathbf{vis}_B$ are visibility declarations placing constraints on which part of the state is visible to which agent.

Fig. 5. Specification of the secret conjunction

5.2 Interpreting Agent Viewpoints

The state of the small program at Fig. 5 is defined by the values of the variables a, b and c but, given the visibility declarations summarised above, each agent has a very different perspective when we take into account which variables they can read at runtime. We adapt the simple semantics of variable classified as either hidden or visible by treating each agent individually via their *viewpoint.*

Definition 6 *(Viewpoint).* Let program *Prog* with named agents $A, B, \ldots$. We write Prog_W for agent W's viewpoint of *Prog*, where all declarations $\mathbf{vis}_{\mathit{list}}$ become **hid**, if W does not appear in *list* and **vis** otherwise. All **hid**, **var** and **vis** declarations remain unchanged.

The semantics of agent W's viewpoint of *Prog* is given by $[\![\mathit{Prog}_W]\!]$. □

Given the visibility declarations at Fig. 5, we define A's viewpoint to be the Shadow semantics of the program at Fig. 6, where all declarations in which A is on the list are set to **vis** and all declarations where A is not on the list are set to **hid**.

Similarly agent B has a viewpoint similar to agent A's except that b is declared visible and a is declared hidden. We use a special universal agent called U to represent a third-party observer who is never included on any visibility list so that all variables except those with the **vis** declaration are hidden from U. Agent U's viewpoint of secret conjunction therefore has both a and b declared hidden, and c declared visible.

In Fig. 7 we summarise the semantics for each agent's viewpoint for secure conjunction. Observe that the implications of the visibility declarations are to set the Shadow sets. In the case of agent A, the observables are derived from variables a and c, and so in the result sets, the triples are of the form $(\mathsf{a}, \{\mathsf{b}, \mathsf{b}'\}, \mathsf{c})$, with the uncertainty in variable b captured as a set of possible values in the second position of the triple correlated with the observation. For agent U, the uncertainty is over variables a and b, and so its viewpoint result sets are of the form $(\{(\mathsf{a}, \mathsf{b}), (\mathsf{a}', \mathsf{b}')\}, \mathsf{c})$, since its Shadow sets are drawn from $\mathcal{A}{\times}\mathcal{B}$.

We define refinement of multiagent systems as follows.

Definition 7 *(Multiagent Refinement).* For specification S and implementation I, involving named agents $A, B, \ldots$, and universal agent U, we say that $S \sqsubseteq I$ provided that $S_W \sqsubseteq I_W$ for all agents W, where S_W, I_W denote the corresponding viewpoints of agent W with respect to S and I. □

$$\begin{array}{ll}
\textbf{vis}\ a{:\in}\,\{0,1\}; & \leftarrow \textit{A chooses a secret bit} \\
\textbf{hid}\ b{:\in}\,\{0,1\}; & \leftarrow \textit{B chooses a secret bit} \\
\textbf{vis}\ c{:=}\,a \wedge b & \leftarrow \textit{Their conjunction is published in c.}
\end{array}$$

Fig. 6. Agent A's viewpoint for secret conjunction

	Agent A	*Agent B*	*Agent U*
vis	a, c	b, c	c
hid	b	a	a, b
Shadow sets contained in	$\mathcal{B}$	$\mathcal{A}$	$\mathcal{A}{\times}\mathcal{B}$
Result sets	$\{\ (0,\{0,1\},0)$, $(1,\{0\},0)$, $(1,\{1\},1)\ \}$	$\{\ (\{0,1\},0,0)$, $(\{0\},1,0)$, $(\{1\},1,1)\ \}$	$\{\ (\{(0,1),(1,0),(0,0)\},0)$, $(\{(1,1)\},1)\ \}$

The state of each viewpoint is derived from triples of variable values a, b, c. In each case the shadow sets are determined by the visibility declarations so that the structure of split states is to group equivalence classes of triples together. For clarity we keep the order of the triples as $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ with the hidden part of the state indicated by braces. For example elements in agent A's final result sets are of the form $(\mathsf{a}, \{\mathsf{b}, \mathsf{b}'\}, \mathsf{c})$ indicating that A can observe the first and last item of the triple but not the middle item. Similarly B's final result sets are of the form $(\{\mathsf{a}, \mathsf{a}'\}, \mathsf{b}, \mathsf{c})$, and agent U's result sets are of the form $(\{(\mathsf{a}, \mathsf{b}), (\mathsf{a}', \mathsf{b}')\}, \mathsf{c})$ since U cannot observe either of the first two components in the triple.

Fig. 7. Semantics for each agent's viewpoint for secret conjunction Fig. 5

We note that visibility declarations can be thought of as placing access restrictions on variables rather than proscribing that an agent can never deduce the value of variables not on his visibility list: that depends on the code. For example, hidden h is published once the statement $v{:=}\,h$ has been executed, and this knowledge is available to all agents. Visibility declarations do however have an impact on which refinements will be judged ultimately to be valid.

For example the following refinement fails, because although nothing has changed with regards agents A, B, for agent U, the program on the left reveals strictly less information than the program on the right.

$$\begin{array}{lcl}
\textbf{vis}_A\ a{:\in}\,\{0,1\}; & & \textbf{vis}_A\ a{:\in}\,\{0,1\}; \\
\textbf{vis}_B\ b{:\in}\,\{0,1\}; & \not\sqsubseteq & \textbf{vis}_B\ b{:\in}\,\{0,1\}; \\
\textbf{vis}_{A,B}\ c{:=}\,a \wedge b & & \textbf{vis}\ c{:=}\,a \wedge b
\end{array} \tag{6}$$

5.3 Agent Viewpoint of Knowledge

Within a multiagent system, it is useful to be able to prove what individual agents know. We recall a special reveal statement $\textbf{reveal}_{list}\phi$ introduced elsewhere [9], which behaves like **reveal** ϕ from the perspective of any agent viewpoint if that agent is on the list *list*, and **skip** otherwise.

Definition 8. *We say that an expression ϕ is effectively list-visible at a point in a program just when putting a statement* $\mathbf{reveal}_{list}\phi$ *there would not alter the program's meaning.*

For example in the program on the left at (6), the expression $a \wedge b$ is effectively $\{A, B\}$-visible, but is not effectively $\{U\}$-visible, since the Universal agent cannot observe the conjunction of a and b, and so following it with $\mathbf{reveal}_U\ a \wedge b$ will change its information flow properties and hence its meaning.

6 Elementary Impossibilities Using the Shadow

One of the basic concerns of secure communication is to establish a shared secret between agents in the system. If the agents are able to communicate already over a private channel, then they can continue to share secrets between each other via that private channel. If they do not have a private channel then they must transmit messages encrypted over a public channel.

Shared secrets are important because they provide the basis for secret communication using an efficient encryption mechanism known as "symmetric encryption". But how do two agents establish any shared secret in the first place if they don't already share a secret channel?

There is a well-known "folk theorem" which describes a limitation of symmetric encryption mechanisms for establishing a shared secret, namely that they cannot do so if they are only able to use a symmetric encryption mechanism. The fact that this general impossibility result exists is the reason why more complicated forms of encryption are used for setting up shared secrets, e.g. Shamir's secret sharing protocol [19]. In this section we review the symmetric encryption folk theorem as an exercise in using algebraic specification and reasoning in the Shadow semantics.

An encryption mechanism is *symmetric* if the decryption key is the same as the encryption key, or can be feasibly calculated from it. In the Shadow semantics that is the same as saying that knowing both the key and the encrypted message is the same as knowing both the key and the message in plaintext.

Definition 9 *(Symmetric encryption).* An encryption mechanism E is *symmetric*, if for any key k and message m we have

$$\mathbf{reveal}(E.m.k, k) \quad = \quad \mathbf{reveal}\ (k, m).$$

□

In the context of the Shadow semantics we can also apply the Decryption Lemma which, when put together with Definition 9, implies we have for any m, k, and a symmetric encryption mechanism E that:

$$\mathbf{reveal}(E.m.k, k) \quad = \quad \mathbf{reveal}(k, m) \quad = \quad \mathbf{reveal}\ (m, E.m.k). \qquad (7)$$

This says that within the threat model for the Shadow semantics, for any symmetric encryption mechanism E, if the observer knows *any two* of k, m or $E.m.k$ then he can deduce the remaining value. Recall that the Decryption Lemma is also implied by a "known plain text attack", thus our reasoning also applies for symmetric encryption mechanisms known to have this weakness.

With (7) we can demonstrate the folk theorem in an elementary way using program algebra.

6.1 Symmetric Encryption and the Folk Theorem

A secret sharing protocol consists of two agents A and B who can send messages to each other over a public channel using a given encryption mechanism E. Once a message is transmitted via the channel then the contents of the file are considered to be observable by any agent, even the universal agent U. We assume that all values are created by either A or B, and respectively stored in variables private to the creator. Values can be chosen at random, or calculated from existing values using E. The system is defined by the operations set out at Fig. 8. The operations model the capacity for each agent to choose a secret value, and to encrypt or decrypt values using the mechanism E. They communicate with each other by sending values over a publicly observable channel. One way to model this is to introduce an explicit channel variable *chan*, which takes values from one of the agents and then passes them to the other. For example, if B sends a value stored in b to A, we could write:

$$\textbf{vis}\ \mathit{chan}{:=}\,b;\textbf{vis}_A\ a{:=}\,\mathit{chan}.$$

However considering only the information flow concerning the variables declared by agents A and B, this is equivalent to

$$\textbf{reveal}\ b;\textbf{vis}_A\ a{:=}\,b \quad = \quad \textbf{vis}_A\ a{:=}\,b\ ;\textbf{reveal}\ b.$$

In Fig. 8 we use the latter formulation because we are interested in information flow, and using **reveal** statements directly will simplify our reasoning.

- $\textbf{vis}_A\ a_i{:\in}\,\mathcal{A}$ — Agent A chooses privately a random value drawn from set $\mathcal{A}$.
- $\textbf{vis}_B\ b_j{:\in}\,\mathcal{B}$ — Agent B chooses privately a random value drawn from set $\mathcal{B}$.
- $\textbf{vis}_A\ a_l{:=}\,E.a_j.a_k$ — Agent A encrypts or decrypts a value from known values stored in variables a_j and a_k .
- $\textbf{vis}_B\ b_l{:=}\,E.b_j.b_k$ — Agent B encrypts or decrypts a value from known values stored in variables b_j and b_k .
- $\textbf{vis}_A\ a{:=}\,Z;\textbf{reveal}\ Z$ — Agent B sends agent A a value Z over the public channel.
- $\textbf{vis}_B\ b{:=}\,Z;\textbf{reveal}\ Z$ — Agent A sends agent B a value Z over the public channel.

Fig. 8. Computation steps in a secret sharing protocol

We model a *secret sharing protocol* between A and B as any sequence of the above statements described in Fig. 8. An example of such a protocol is:

$$\begin{array}{ll}\mathbf{vis}_A\ a{:}\in \mathcal{A}; & \leftarrow \textit{A chooses a random value}\\ \mathbf{vis}_A\ a'{:}\in \mathcal{A} & \leftarrow \textit{A chooses another random value}\\ \mathbf{vis}_B\ b{:=}\,E.a.a'; \mathbf{reveal}\ E.a.a' & \leftarrow \textit{A sends B variable a encrypted with } a'\end{array}$$

We say that a value Z is *generally known* after executing P if

$$P; \mathbf{reveal}\ Z \quad = \quad P. \tag{8}$$

In the above protocol, provided that neither $\mathcal{A}$ nor $\mathcal{A}'$ are singleton sets then neither a nor a' is generally known; in fact these values are also not known to agent B.

Learning a Value. Given any secret sharing protocol P, the agents only know the value of a variable if either they created it for themselves, or they were sent it, or they were able to decrypt it using E with values they already knew. For example, if agent B knows two values X and Y after executing P, and the last action was to receive value X from agent A, then either he knew Y before receiving X from A, or he was able to use X to decrypt $E.X.Y$, which value he already knew. We capture this "learning event" with the following property:[5]

$$\begin{aligned} &P; \mathbf{reveal}_B(Y, X) = P \Rightarrow \\ &\quad P'; \mathbf{reveal}_B\ Y = P' \vee P'; \mathbf{reveal}_B\ E.Y.X = P', \end{aligned} \tag{9}$$

where $P = P'; \mathbf{vis}_B\ b{:=}\,X; \mathbf{reveal}\ X$.

6.2 The Folk Theorem

We state the folk theorem as follows:

Theorem 4 *(Symmetric Encryption Folk Theorem).* Let P be a secret sharing protocol between agents A and B using symmetric encryption method E. Any value X that is known by both A and B after executing P is also known generally, i.e.

$$\begin{aligned} P; \mathbf{reveal}_A\ X\ &= P\ \wedge\ P; \mathbf{reveal}_B\ X\ = P \\ &\Rightarrow \quad P; \mathbf{reveal}\ X\ = P. \end{aligned} \tag{10}$$

Proof. We use structural induction. For simplicity we also assume that all variables are initialised at most once, to avoid questions of definability.

The base case is when $P = \mathbf{skip}$, and the result follows since we assume initially that all secrets known to both A and B are generally known.

We next consider $P = P'; Q$, where Q is one of the statements in Fig. 8. We assume by the inductive hypothesis that P' satisfies the theorem. We assume

[5] We prove in the Appendix Sect. A.1 that this property holds for E implemented with $\oplus$, exclusive-or.

also that $P; \textbf{reveal}_A\ Y = P; \textbf{reveal}_B\ Y = P$ for some value Y. We must show, for each case of Q, that $P; \textbf{reveal}\ Y = P$ also.

Suppose Q is the action "Agent A sends value X to Agent B on the public channel". i.e. $P = P'; \textbf{vis}_B\ b{:=}X; \textbf{reveal}\ X$. We have the following facts:

(i) Agent A knew value X already after P':

$$P'; \textbf{reveal}_A\ X = P'$$

(ii) Agent A knew value Y already after P':

$$P'; \textbf{reveal}_A\ Y = P'$$

(iii) Definition 7 for Agent A:

$$P'; \textbf{reveal}_A\ E.X.Y = P'$$

(iv) Assumption:

$$P; \textbf{reveal}\ Y = P$$

Fact (i) is assumed since Agent A sends value X to Agent B. Fact (ii) follows from (i) since:

$$\begin{array}{lll} & P'; \textbf{reveal}_A\ Y & \\ = & P'; \textbf{reveal}_A\ X; \textbf{reveal}_A Y & \text{“(i) above”} \\ = & P'; \textbf{reveal}_A X & \text{“Assumption: } P'; \textbf{reveal}_A\ (X,Y) = P'; \textbf{reveal}_A\ X\text{”} \\ = & P'\ . & \text{“(i) above”} \end{array}$$

Using the above facts, we can now reason as follows that value Y is also known generally after P.

We begin with (9) because after Agent B receives value X, then either he knew Y already, or he learned it. We treat each case separately.
Case 1: Suppose first that $P'; \textbf{reveal}_B\ Y = P'$. (Agent B knew Y already.) By (ii) and structural induction on P', we deduce immediately that Y was generally known after P', i.e. $P'; \textbf{reveal}\ Y = P'$.

Perfect recall then gives us immediately that $P; \textbf{reveal}\ Y = P$.

Case 2: Suppose instead from (9) that $P'; \textbf{reveal}_B\ E.X.Y = P'$. But we also have by (iii) that $P'; \textbf{reveal}_A\ E.X.Y = P'$, and so by structural induction we deduce that $P'; \textbf{reveal}\ E.X.Y = P'$.

Thus by perfect recall we must have that $P; \textbf{reveal}\ E.X.Y = P$. We put this together with (4) and (7) to deduce in fact that $P; \textbf{reveal}\ Y = P$, as required.

The other cases for Q, when it represents a local assignment by either A or B to local variables, does not transfer new knowledge of the other agent, and so there is nothing to prove.

□

7 Conclusions and Outlook

This paper reviews refinement of ignorance and how it applies to the analysis of information flow. The Shadow semantics is based on ideas originally due to Mantel [8], Leino [7] and Sebelfeld [17]. Mantel's emphasis is on events and traces, whilst Leino and Sabelfeld concentrate on information flow with respect to the (initial) values of the secret. The Shadow semantics is concerned with the final state, with the shadow variable H in an observation (v, H) acting as a digest of all information flows relating to the current value of the secret variable h.

Our treatment of the folk theorem is inspired by Schmidt *et al.* [18] who also provide some mechanical support to determine whether protocol designs based on a given equational theory are able to establish a shared secret.

Variations on the Shadow semantics include probability and this line of work has resulted in connections to the channel model of information flow [2,13], allowing the amount of information flowing to be quantified thus giving a better understanding of the degree to which programs leak information.

A Some Proofs

A.1 Learning a Value

We prove (9) for the special case where E is the exclusive-OR, $\oplus$. We re-state (9) here for $\oplus$.

$$\begin{aligned} &P; \mathbf{reveal}_B(Y, X) = P \Rightarrow \\ &\quad P'; \mathbf{reveal}_B\ Y = P' \vee P'; \mathbf{reveal}_B\ X \oplus Y = P', \end{aligned} \tag{11}$$

where $P = P'; \mathbf{vis}_B\ b{:=}\,X; \mathbf{reveal}\ X$.

Assume that B has (private) variables $b_1, b_2, \ldots b_k$, and that A has (private) variables $a_1, a_2 \ldots a_m$. We also assume that each variable is assigned exactly once and then never changed.

B's knowledge of the state after executing P' can be summarised as follows:

1. B knows the values of all its own variables $b_1, b_2, \ldots b_k$.
2. B knows the values of any encrypted value sent over the public channel. Each of those values is of the form

$$a_{i_1} \oplus a_{i_2} \oplus \cdots \oplus a_{i_n},$$

   for some selection of variables drawn from $a_1, a_2 \ldots a_m$.
3. B can learn new facts by using the facts he already has and computing new values using $\oplus$. Let $\mathcal{K}$ be the set of facts so-computed using $\oplus$.

Notice that whenever a value $\alpha \in \mathcal{K}$ we have that $P'; \mathbf{reveal}_B\ \alpha = P'$.

Assume that $Y \notin \mathcal{K}$, but that when B learns X then he can deduce Y. We will show that $X \oplus Y$ is contained in $\mathcal{K}$.

Since $Y \notin \mathcal{K}$, this means that $Y = X \oplus \alpha$ for some fact $\alpha \in \mathcal{K}$. But with this we reason:

$$
\begin{array}{ll}
 & Y = X \oplus \alpha \\
\Rightarrow & X \oplus Y = X \oplus X \oplus \alpha \\
\Rightarrow & X \oplus Y = \alpha, \qquad \text{“Property of } \oplus \text{”}
\end{array}
$$

implying that $X \oplus Y$ (i.e. $E.Y.X$ as at (9)) is contained in $\mathcal{K}$.

References

1. Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF 2012), pp. 265–279, June 2012
3. Back, R.-J.R.: Correctness preserving program refinements: proof theory and applications, Tract 131, Mathematisch Centrum, Amsterdam (1980)
4. Denning, D.: Cryptography and Data Security. Addison-Wesley, Boston (1983)
5. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 75–86. IEEE Computer Society (1984)
6. Landauer, J., Redmond, T.: A lattice of information. In: Proceedings of the 6th IEEE Computer Security Foundations Workshop (CSFW 1993), pp. 65–70, June 1993
7. Leino, K.R.M., Joshi, R.: A semantic approach to secure information flow. Sci. Comput. Program. **37**(1–3), 113–138 (2000)
8. Mantel, H.: Preserving information flow properties under refinement. In: Proceedings of the IEEE Symposium Security and Privacy, pp. 78–91 (2001)
9. McIver, A.K.: The secret art of computer programming. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 61–78. Springer, Heidelberg (2009)
10. McIver, A.K., Morgan, C.C.: A calculus of revelations. Presented at VSTTE Theories Workshop, October 2008. http://www.cs.york.ac.uk/vstte08/
11. McIver, A.K., Morgan, C.C.: *Sums and Lovers*: Case Studies in Security, Compositionality and Refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 289–304. Springer, Heidelberg (2009)
12. McIver, A., Meinicke, L., Morgan, C.: Compositional closure for bayes risk in probabilistic noninterference. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 223–235. Springer, Heidelberg (2010)
13. McIver, A., Morgan, C., Smith, G., Espinoza, B., Meinicke, L.: Abstract channels and their robust information-leakage ordering. In: Abadi, M., Kremer, S. (eds.) POST 2014 (ETAPS 2014). LNCS, vol. 8414, pp. 83–102. Springer, Heidelberg (2014)
14. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall, Upper Saddle River (1994). http://web.comlab.ox.ac.uk/oucl/publications/books/PfS/
15. Morgan, C.C.: The shadow knows: refinement of ignorance in sequential programs. In: Uustalu, T. (ed.) Math Prog Construction. Springer, vol. 4014, pp. 359–378. Springer, Treats Dining Cryptographers (2006)
16. Morgan, C.C., Knows, T.S.: Refinement of ignorance in sequential programs. Sci. Comput. Program. **74**(8), 629–653 (2009). Treats Oblivious Transfer

17. Sabelfeld, A., Sands, D.: A PER model of secure information flow in sequential programs. High.-Ord. Symbolic Comput. **14**(1), 59–91 (2001)
18. Schmidt, B., Schaller, P., Basin, D.: Impossibility results for secret establishment. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF), pp. 261–273 (2010)
19. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
20. Shannon, C.E.: Theory of secrecy systems. Bell Syst. Tech. J. **28**, 656–715 (1949)

Springer

http://www.springer.com/978-3-319-29627-2

Engineering Trustworthy Software Systems
First International School, SETSS 2014, Chongqing, China, September 8-13, 2014. Tutorial Lectures
Liu, Z.; Zhang, Z. (Eds.)
2016, XI, 325 p. 141 illus. in color., Softcover
ISBN: 978-3-319-29627-2