20

A probabilistic approach to information hiding

Annabelle McIver and Carroll Morgan

Abstract

Security in a computer system must at some level be regarded as the control of information flow. Here we approach the issue 'from first principles', linking information flow in the sense of Shannon to a sequential programming model enriched with probabilities. From that we extract criteria for secure encapsulation of data, and we discuss the interaction of our criteria with refinement.

20.1 Introduction

The analysis of how computer systems keep secrets has at its basis 'information flow' as developed by Shannon [13]. Whilst the analysis of information flow is rather complicated (involving the consideration of probabilities) one expects that the analysis of program properties should be much simpler, and indeed standard semantics of programs does not address probability at all. Thus to acheive our expectation we need to find the right way to abstract from the probabilistic aspects of information flow, leaving a non-probabilistic characterisation that can be formulated simply in a standard program semantics.

In this paper we first develop a theory of information flow in a model of sequential programs that does accommodate the possibility of a probabilistic context. That supports a very general theory of information flow, similar to the classical theory of 'leaky channels', but adapted to allow for program-specific features. In particular we consider *demonic nondeterminism*, which is beyond the scope of classical information models. Our contribution is then to specialise the theory to non-probabilistic programs; and our second contribution is to abstract from the probability altogether, thus giving a characterisation of information flow in the traditional program-refinement context.

In Sec. 20.2 we set out the background of information flow, and we develop our own theory in Secs. 20.3 and 20.4. The definitions are illustrated by the small example of a secure file store, in Sec. 20.5.

We write $f.s$ for function f applied to argument s, with application associating to the left; and "$:=$" means "is defined to be".

20.2 Background: multi-level security and information flow

Historically, information flow is asociated with 'interference' in the context of multi-user systems [11, 3, 1] where the users have differing security clearances. The simplest scenario consists of two users *High* and *Low*, with respectively high and low security classifications, and the implication of the low classification is to prevent the access of *Low* to *High*'s activities or personal information. In a nutshell, *High*'s activities (whatever they are) should not 'interfere' with *Low*'s because, if they do, *Low* might deduce something about *High*.

Definitions of non-interference focus on *Low*'s ability to correlate behavioural phenomena with *High*'s activities, hence the introduction of the notion of users' 'views'. Roughly speaking, *Low*'s view is a description of the possible actions available to *Low*, including their outputs, during execution of the system. We say that *High*'s activities do not interfere just when *Low*'s view is independent of their presence or absence.

For example, a *history* of the system is a sequence of *High* and *Low* actions, interleaved; let tr and tr' be two histories. If they give the same subtrace when projected onto *Low*'s actions alone, then *Low*'s *view* of the system should be the same after either tr or tr' — even if the two traces' projections onto *High*'s actions differ. That is, the actions available to *Low* should be the same after either tr or tr'; and if such is the case then *High* is deemed not to interfere with *Low*.

Consideration of views forms the basis of many process-algebra style formulations of information flow, and have led to 'unwinding theorems' that express the security as local, checkable restrictions on individual actions.

However there are problems with focusing too much on sequences. In fact there are two ways that privacy is compromised in multi-user systems; and as Roscoe pointed out [10], sequence-based conditions fail to identify one of them (at (2) below):

1. *Low* can infer something about *High*'s activities, because they're not independent; and
2. *High* can act as a 'mole', leaking information to *Low* using an agreed encoding of sequences of actions.

Breach (2) can occur whenever there is some 'demonic nondeterminism' in the system, leading to the so-called "refinement paradox" which bedevilled many early characterisations of non-interference [12]. It's called a paradox because a desirable property (in this case pertaining to security) is not preserved under the relation of program refinement,[1] and as Roscoe observed [10] it would "require a a high order of sophistry to argue that a process which is allowed to behave as though it were insecure can be secure".

The impact of (2) was to suggest that definitions of non-interference be based on determinism [9], thus avoiding the paradox by default. More recently Lowe [5] has argued that another way to avoid the paradox is to adjust the program model to one that can distinguish between nondeterminism controllable by *High* and nondeterminism that cannot be so controlled (rather than banning it outright).

We discuss such a model in Sec. 20.6, and because nondeterminism is so important as a design tool, we will stress that probabilistic models provide the key to understanding the distinction between allowed and disallowed nondeterminism.

In the next section, however, we concentrate on (1), with the aim of finding a locally checkable semantic characterisation of information flow for imperative programs. To avoid the difficulties encountered with sequences, however, we do not attempt to formulate non-interference directly: instead we take classical information theory as our starting point. For that we need to consider programs in the context of probability.

20.3 Classical information theory, and program refinement

Shannon's classical theory of information [13] is founded in probability theory, and the measure of 'uncertainty' communicated by a probability distribution. We begin this section by introducing and adapting some of the standard ideas from that theory for expressing information flow.

20.3.1 Probability distributions and random variables

Let S be a finite state space and and let $\overline{S}$ be the set of probability distributions over S, so that

$$\overline{S} \ := \ \{\Delta\colon S \to [0,1] \mid \textstyle\sum_{s:S} \Delta.s = 1\} \, .$$

As special cases, we shall frequently refer to the "uniform distribution" and "point distribution" defined as follows. For S' a subset of S, the *uniform distribution* $\overline{S'}$ over that subset is defined

$$\overline{S'}.s \ := \ \tfrac{1}{|S'|} \ \mathbf{if}\, s{\in}S' \,\mathbf{else}\ 0 \, ,$$

[1]But it is of course not a paradox; rather it is an exposure of the relative nature of the concept of refinement. For example, 'traditional' refinement does not preserve the desirable property of speedy execution either.

for s in S. And we define the *point distribution* $\overline{s}$, the case that assigns probability 1 to s and probability 0 to all other points, as

$$\overline{s} \ := \ \overline{\{s\}}\,,$$

so that $(\bar{\cdot})$ can be seen in both cases — for both subsets and elements of S — as a canonical injection into $\overline{S}$.[2]

Subsets of S are called *events*, and if we apply a distribution Δ to an event E (rather than to a point) we mean its aggregate value over that subset; thus $\Delta.E$ is just $\sum_{s:E}\Delta.s$.

Finally, let f be a function over S, a *random variable*; then a distribution Δ on S induces a distribution on the codomain of that function: the probability that $f.s = x$ is just the value assigned by Δ to the event $f^{-1}.x$. We write $f.\Delta$ for that induced distribution $\Delta \circ f^{-1}$, on $f.S$, which is what we will mean if we refer to "the distribution of the random variable f".

20.3.2 Conditional distributions

Given a distribution Δ and event E, the *conditional distribution* Δ/E of Δ with respect to E is defined

$$(\Delta/E).s \ := \ \Delta.(E \cap \{s\})/\Delta.E\,,$$

with the value 1 being taken, by convention, should $\Delta.E$ happen to be 0.

The *conditional distribution of random variable f given E* is just the distribution $f.(\Delta/E)$ induced by f over Δ/E. Where Δ is understood, we will write just f/E.

20.3.3 Entropy

Entropy is a means to quantify uncertainty [13]. Given a distribution Δ, the entropy $\mathcal{H}.\Delta$ of Δ is defined

$$\mathcal{H}.\Delta \ := \ -\textstyle\sum_{s:S}\Delta.s \times \lg(\Delta.s)\,,$$

where the sum is taken only over s with $\Delta.s \neq 0$ and the logarithm "lg" is conventionally taken to the base 2. It is a measure of the uncertainty that Δ represents, and is largest when the distribution is uniform, smallest when the distribution is a point. In Fig. 20.1 we list some standard properties of the entropy of a distribution.

Given a random variable f over Δ, by the entropy of f we mean the entropy $\mathcal{H}.(f.\Delta)$ of the distribution induced by f. Again, we write just $\mathcal{H}.f$ when Δ is understood.

In our application to programs, we will use entropy as a measure of how much *Low* can know about the value of *High*'s variables.

[2]Whether the notation $\overline{S}$ itself denotes the whole space of distributions, or the (single) uniform distribution over that space, will be clear from context.

$\mathcal{H}.\overline{s} = 0$	Point distributions exhibit no uncertainty.
$0 \le \mathcal{H}.\Delta \le \lg \mid S' \mid$	The largest possible entropy of Δ over subset S', *i.e.* when $\Delta.S' = 1$, is $\lg \mid S' \mid$.
$\mathcal{H}.\overline{S'} = \lg \mid S' \mid$	Uniform distributions exhibit maximal uncertainty.
If random variable f is injective then $\mathcal{H}.(f.\Delta) = \mathcal{H}.\Delta$.	Entropy is independent of the domain values of Δ.

Figure 20.1. Some standard properties of entropy.

20.3.4 Conditional entropies

Given Δ and E, the *conditional entropy of* Δ *given* E is just the entropy of the conditional distribution Δ/E. The conditional entropy of random variable f with respect to E is the entropy of its induced conditional distribution $(f.\Delta)/E = f.(\Delta/E)$, which we may write $\mathcal{H}.(f/E)$ when Δ is clear.

More generally, suppose we have a second random variable g over S, and let y be an element of its codomain. The conditional entropy of f given that $g.s = y$ would be

$$\mathcal{H}.(f/\{s \mid g.s = y\})\,, \tag{20.1}$$

and we would in effect be looking at the (remaining) uncertainty in our knowledge of f's value given that we know a specific value y that g has taken.

The expression (20.1) above can itself be regarded as a random variable (a function of y) over the co-domain $g.S$ of g, and as such it has an expected value over the induced distribution $g.\Delta$. That quantity is known as the *conditional entropy of* f *given* g, and is defined

$$\mathcal{H}_\Delta.(f/g) \;:=\; \textstyle\sum_{y:g.S} g.\Delta.y \times \mathcal{H}.(f.\Delta/\{s \mid g.s = y\})\,.$$

We drop the subscripted Δ when it is understood.

We shall use conditional entropy as follows. Think of f as the projection of the state space onto the high-level variables, and g as the projection onto the low-level variables. Then $\mathcal{H}_\Delta.(f/g)$ measures the average uncertainty about *High*'s variables that still remains after the value of *Low*'s variables have been observed.

20.3.5 Information escape and channel capacity

In information theory, a channel is a model of data transmission and is described by a probabilistic function from input values to output values. The 'channel capacity' measures the additional information knowledge of the output values gives about the inputs that led to them. For our purposes we will consider the input and output spaces both to be S.

In this paper the 'channel' will be the operation of a program fragment, combined with channel-like observations of the 'before' and 'after' state via projection onto the *Low* variables: the information escape is the difference between what can be deduced about h before the program runs and what can be deduced after, the deduction being in both cases via l-observations only. Thus if l, h are respectively the projection functions from S onto the *Low*,*High* variables, then for a specific execution of the program that took initial distribution Δ to final distribution Δ', the 'net information escape' would be

$$\mathcal{H}_{\Delta}.(h/l) - \mathcal{H}_{\Delta'}.(h/l)\,.$$

It is the appropriately conditioned 'uncertainty before' minus the 'uncertainty after'.

Our next task is to make that definition available for our model of programming, which admits both demonic nondeterminism and probability.

20.3.6 Probabilistic guarded commands

The probabilistic guarded command language [7] consists of traditional guarded commands [2] together with a binary probabilistic choice operator ${}_p\oplus$. The operational meaning of the expression $A\ {}_p\oplus\ B$ is that either A or B is executed with probability respectively p or $1{-}p$. Since there is no determined output, this behaviour is sometimes called "probabilistic nondeterminism". It is however very different from 'demonic nondeterminism', already present in standard guarded commands and which represents underspecification or demonic scheduling in distributed systems.

Indeed the two operators are modelled very differently — as usual probabilistic information is described by probability distributions, whereas demonic behaviour is is described by subsets of possibilities. Put together in probabilistic guarded commands that leads to a model in which programs are described by functions from initial state to sets of distributions over final states, with the possible multiplicity of the result set representing a degree of nondeterminism and the distributions recording the probabilistic information once that nondeterminism is resolved. We recall the following definition for the probabilistic program space $\mathcal{M}S$ [6, 7]:[3]

$$\mathcal{M}S \ := \ S \to \mathbb{P}\overline{S}\,.$$

We order programs using program refinement, which measures the range of nondeterminism — programs higher up the refinement order exhibit less nondeterminism than those lower down:

$$Q \sqsubseteq P \ \text{ iff } \ (\forall\, s{:}\, S \cdot P.s \subseteq Q.s)\,.$$

[3]In our earlier work the space is $\mathcal{H}S$, but here we reserve it as the traditional notation for entropy.

$$
\begin{array}{ll}
(\mathbf{assign} f).s & := \{\overline{f.s}\}\ \text{, for function } f \text{ in } S \to S \\[1ex]
\mathbf{skip} & := \mathbf{assign}\ id \\[1ex]
(r\ {}_p\!\oplus r').s & := r.s\ {}_p\!\oplus r'.s \\
 & \quad \text{where } {}_p\!\oplus \text{ acts over all pairs from its arguments} \\[1ex]
(r \sqcap r').s & := (\cup p\colon [0,1] \cdot r.s\ {}_p\!\oplus r'.s) \\[1ex]
(r; r').s & := \{\Delta\colon r.s; f'\colon S \to \overline{S} \mid r' \supseteq f' \cdot \mathrm{Exp}.\Delta.f'\} \\
 & \quad \text{where } \mathrm{Exp}.\Delta.f' \text{ is the expected value of } f' \text{ over } \Delta
\end{array}
$$

$$(r\ \ \mathbf{if} B\, \mathbf{else}\ \ r').s := r.s\ \ \mathbf{if} B.s\, \mathbf{else}\ \ r'.s$$

We also write $s{:}\epsilon\, S'$ for the nondeterministic assignment to s from the set S, and similarly $s{:}\epsilon\, \Delta$ for the probabilistic assignment to s according to distribution Δ. (We tell the difference by noting whether the right-hand side is a set or a distribution.) Particularly in the latter case the syntax would be rather convoluted otherwise, for we would have to write

$$x := i_0\ {}_{\Delta.i_0}\!\oplus (x := i_1\ {}_{\Delta.i_1/(1-\Delta.i_0)}\!\oplus \cdots)\ .$$

Figure 20.2. Probabilistic relational semantics[4]

The full semantics is set out in Fig. 20.2.

The consequences of the refinement order are, for example, that if Q guarantees to establish a predicate ϕ with probability at least p (irrespective of the nondeterminism) then P must establish ϕ with probability at least that same p. That probabilistic properties are robust against program refinement is one way to understand the difference between probability and demonic nondeterminism.

For example a fair coin flip cannot be refined by any program except itself, hence in particular we have

$$s\colon = 0\ {}_{1/2}\!\oplus s\colon = 1 \quad \not\sqsubseteq \quad s\colon = 1\ ,$$

since the left-hand program establishes $s = 0$ with probability $1/2$, but the right-hand side establishes $s = 0$ only with probability 0. This is the sense in which probability cannot be ‘refined away”. Demonic nondeterminism, on the other hand, can always be refined away:

$$s\colon = 0 \sqcap s\colon = 1 \quad \sqsubseteq \quad s\colon = 1\ .$$

[4]For simplicity we do not treat non-termination, although the theory extends easily to accommodate it.

In that refinement, the left-hand side is never guaranteed to execute the assignment $s := 0$; but the right-hand side is guaranteed never to do so.

In summary, the program $A\ {}_p\!\oplus\ B$ means that A is executed predictably with probability p, whereas no such quantitative statements can be made about the program $A \sqcap B$. We shall use that property of probability to to guarantee 'refinement-proofness' in our characterisation of security, to which we now turn.

20.4 Information flow in imperative programs

Following the general scheme of *High* and *Low* users, we associate them with corresponding variables named h and l; in this framework we imagine that *Low* can read variables named l but not h. Security systems are built from a collection of procedures or operations which, when called, grant users the opportunity of updating (possibly) the values of the variables. In this setting, as for Rushby's approach [11], *Low*'s view is based on the traces of the values held on l during the use of the system, and the intention of system security can be described as follows:

> A system comprising operations *Op* is secure provided that if the value of *High*'s variables are not known (to *Low*) initially, then they cannot be inferred at any later time during use.

Put another way, we could say that *Low* cannot infer the value of *High*'s variables only if *Low*'s variables remain uncorrelated to those of *High*. And from Sec. 20.3.5 we can measure the degree of correlation using conditional entropy and channel capacity.

Let the projection functions from S onto its *High*- and *Low* components be called h, l; and let the corresponding types be H, L. Given a distribution Δ in $\overline{S}$, the conditional entropy of h with respect to l is, as we have seen, given by $\mathcal{H}_\Delta.(h/l)$.

To relate that to our programming model, we look for the *maximum* change in that entropy over all possible executions of a given operation; that is we maximise over all its possible initial distributions as well as all possible resolutions of demonic nondeterminism within it.

For program operation *Op* and distribution Δ, we write simply $Op.\Delta$ for the set of possible state distributions that could result from executing *Op* with Δ as the distribution of the initial state. Then, based on the notion of information escape from Sec. 20.3.5, we define the insecurity introduced by the program — with respect to flow from h to l — by considering the 'worst case' of the program's behaviour, taken over all its demonic choices and possible incoming h, l-distributions:

Definition 20.4.1 *Given an operation Op in $\mathcal{M}S$, the h-to-l channel capacity of Op is given by*

$$\mathcal{C}_l^h[\![Op]\!] \ :=\ (\sqcup \Delta : \overline{S};\ \Delta' : Op.\Delta \cdot \mathcal{H}_\Delta.(h/l) - \mathcal{H}_{\Delta'}.(h/l))\ .$$

The maximising over Δ considers all possible initial distributions; and the maximising over Δ' considers all possible resolutions of nondeterminism once the initial distribution has been selected.

In the special (but common) case that S is just the Cartesian product of the spaces H and L, it can be shown that one need not consider incoming distributions over L: the initial distribution can be taken (effectively) over H alone, provided one maximises over all possible initial values of l. That is, Def. 20.4.1 implies in this case that $\mathcal{C}_l^h[\![Op]\!]$ is

$$\begin{array}{lr}
(\sqcup \lambda \in L \bullet & \text{The maximum over all initial } l \text{ values } \lambda \text{ of} \\
\quad (\sqcup \Delta\colon \overline{H} \bullet & \text{the maximum over all initial } h \text{ distributions of} \\
\quad\quad \mathcal{H}.\Delta - & \text{the incoming } h\text{-entropy } \textit{minus} \\
\quad\quad (\sqcap \Delta'\colon (l\colon= \lambda\, ;\, h{:}\epsilon\, \Delta;\, Op) \bullet & \text{the minimum over all outgoing}\ldots \\
\quad\quad\quad \mathcal{H}_{\Delta'}.(h/l) & \ldots(h/l)\text{-conditional entropies.} \\
)))\,, &
\end{array} \tag{20.2}$$

where we have written $(l\colon= \lambda\, ;\, h{:}\epsilon\, \Delta;\, Op)$ for a the set of distributions potentially resulting from running that program fragment. As an extension of that notation, because we are minimising (acting demonically) at that point, later we will write

$$l\colon= \lambda\, ;\, h{:}\epsilon\, \Delta;\, Op;\, \mathcal{H}.(h/l)$$

for $(\sqcap \Delta'\colon (l\colon= \lambda\, ;\, h{:}\epsilon\, \Delta;\, Op) \bullet \mathcal{H}_{\Delta'}.(h/l))$, which has the intuitive sense of 'execute the fragment $(l\colon= \lambda\, ;\, h{:}\epsilon\, \Delta;\, Op)$ and take the minimum possible conditional entropy $\mathcal{H}.(h/l)$ at its end".[5]

We now consider the following program fragments in order to investigate how Def. 20.4.1 — and its equivalent (20.2) — measure up with our intuitive notions of security. The state space S is the Cartesian product $H \times L$ of the high- and low-level types, and the projection functions are the standard Cartesian projections.

1. $\mathcal{C}_l^h[\![l\colon= h]\!] \;=\; \lg \mid H \mid$.
2. $\mathcal{C}_l^h[\![h\colon= l]\!] \;=\; \lg \mid H \mid$.
3. $\mathcal{C}_l^h[\![h\colon= (h+1)\ \mathbf{mod} \mid H \mid]\!] \;=\; 0$.
4. $\mathcal{C}_l^h[\![l\colon= h\ \mathbf{mod}\ 2]\!] \;=\; 1$.
5. $\mathcal{C}_l^h[\![l{:}\epsilon\, L]\!] \;=\; \lg \mid H \mid\ \mathbf{min}\ \lg \mid L \mid$.

Examples (1) and (2) give channel capacities for programs that we would consider to be totally insecure — in both cases the result of execution is to establish that h and l have the same value, so that the final conditional entropy of h with respect to l is 0; the maximum possible initial conditional entropy is when h and l are independent, and is $\lg \mid H \mid$. That agrees with the quantitative assessment

[5]That notation is motivated by the 'expression blocks" of some Algol-like languages.

which gives maximal channel capacities — it is an extreme case of insecurity since all details of h are revealed via l.[6]

Example (3) on the other hand represents only a shuffling of the values of h and so should normally be regarded as secure — if the exact value of h is not known before execution then it cannot be known afterwards. Again we see that this agrees with the quantitative assessment, for the channel capacity of 0 signals the opposite extreme, in that there is no correlation at all between h and l.

Examples (4) and (5) lie somewhere between those extreme cases. Example (4) publishes something about h — but not everything — and the channel capacity suggests that it reveals one bit of information.

Finally, the demonic nondeterminism in (5) introduces insecurity because it can be resolved to reveal part or all of h depending on the relative ranges of h and l. When $\mid H \mid \leq \mid L \mid$, clearly all information $\lg \mid H \mid$ about h can be revealed; but one can in any case never reveal more than $\lg \mid L \mid$.

Encouraged by all this we can finally define our security property by insisting that *Op* is secure only if its execution releases no information at all.

Definition 20.4.2 *Using the* High *and* Low *conventions, we say that Op in* $\mathcal{MS}$ *is* h, l*-secure iff*[7]

$$\mathcal{C}_l^h[\![Op]\!] \;=\; 0\,.$$

In most realistic cases the calculation of exact channel capacities will of course be very difficult, and using Def. 20.4.2 directly would therefore be impractical. In the rest of this section we shall look for simpler formulations based on program refinement, and will concentrate on programs which contain no probabilistic choice at all. We call these programs standard programs, and they form a subset of $\mathcal{MS}$.[8]

20.4.1 Information flow and program refinement

We begin by considering properties of standard programs that imply Def. 20.4.2.

Suppose $0 \leq h < N$, and consider the program fragment

$$h{:}=(h+1)\ \mathbf{mod}\ N\,.$$

It is considered to be secure, since it merely permutes *High*'s variables. The program $h{:}=0$, on the other hand is not considered to be secure, since it guarantees to set h to 0 whatever its initial value. Nor is it a permutation (*i.e.* it does not inject H into H).

These observations are not surprising since Def. 20.4.2 is based on entropy, which we recall from Fig. 20.1 is left invariant under injections.

[6]Note that in some formulations [4] example (2) would be *secure* because h's initial value is not revealed. (See Sec. 20.4.2.)

[7]Because channel capacities cannot be negative, we may equate no information flow with zero channel capacities.

[8]Although standard programs contain no probabilistic choice, we must analyse them in the probabilistic model in order to use our notions of entropy and information flow.

In finite state spaces, permutations correspond to isomorphisms, and the condition "*Op* permutes the hidden part of the state space" amounts to saying that for each initial l value, the number of possible final h values after running *Op* must be exactly $\mid H \mid$. But permutations are not quite enough for security, because we need to ensure as well that the value of h is not communicated by the value of l. (Note that the program $l{:=}\ h$ is the identity permutation on H, yet is insecure.) Those requirements are met by this definition, whose motivation follows.

Definition 20.4.3 *We say that Op in $\mathcal{MS}$ acts as a* secure permutation *if any deterministic refinement Op' of Op satisfies*

$$Op';\ h{:\in} H \quad = \quad h{:\in} H;\ Op'\ .$$

Our definition of secure permutation must encapsulate two ideas involved in information flow: neither the initial nor the final high values can be deduced from the final values of *Op*. The secrecy of the initial values is preserved if the refinement

$$Op';\ h{:\in} H \quad \sqsubseteq \quad h{:\in} H;\ Op'\ .$$

holds: its refinement states that one possible behaviour of Op' is as if it were run after any initialisation of h whatsoever (from $h{:\in} H$ on the *rhs*). so that it cannot be communicating any information about the actual value of h (*lhs*). (The $h{:\in} H$ on the left is only to allow the refinement to go through for any action on the right of *Op* on h.)

Similarly the secrecy of the final values can be captured by the reverse refinement. Def. 20.4.3 simply combines the two.

The next lemma shows that indeed programs that are secure permutations are secure.

Lemma 20.4.4 *If Op is a secure permuation, then Op is secure according to Def. 20.4.2.*

Proof. For any λ in L, it follows from Def. 20.4.3 that, for any deterministic refinement Op' of *Op*, the function $Op'.\lambda$, acting on h-values, is just a permutation. Since it is a permutation, Fig. 20.2 tells us that for any Δ' in ($l{:=}\ \lambda$; $h{:\in} \Delta$; *Op*) we have

$$\mathcal{H}_{\Delta'}.(h/l) \quad = \quad \mathcal{H}.\Delta' \quad = \quad \mathcal{H}.\Delta\ .$$

(The first equality os from Def. 20.4.3 again: we note that it implies a single final l-value λ', given fixed initial λ, which makes the conditional h/l degenerate.) Hence from Def. 20.4.1 we have $\mathcal{C}_l^h[\![Op]\!] = 0$, implying Def. 20.4.2. □

In fact, as we shall see, permutations account for all the secure standard programs. To prove it we turn to refinement properties of programs that are implied by security.

One way to see how security can be described by refinement is to think of it as a problem of data abstraction in which *High*'s variables are hidden. This is reminiscent of the idea of views, in that *Low*'s view with h abstracted should appear as though nothing about h is known. Since complete ignorance of the value of h is described by the uniform distribution, that view should remain invariant on execution of P. Another way of expressing that invariance is to say that P is 'uniform preserving':

Definition 20.4.5 *We say that Op in $\mathcal{MS}$ is* uniform preserving *if*

$$l{:}\in L;\ h{:}\in\overline{H}\ \sqsubseteq\ h{:}\in\overline{H};\ Op\,.$$

Def. 20.4.5 takes the form of a simulation equation typical of data abstraction, in which the program $h{:}\in\overline{H}$ (that is, 'choose h from H according to the uniform distribution $\overline{H}$') is the abstraction invariant: we are saying that the operation Op 'is a data refinement under h-is-uniform" of $l{:}\in L$. The effect is not to constrain Op's effect on l at all, except to maintain h's uniformity.

Lemma 20.4.6 *If Op in $\mathcal{MS}$ is secure, then it is uniform preserving.*

Proof. If Op is secure, then from Def. 20.4.2 we must have that $\mathcal{C}_l^h[\![Op]\!]=0$ and in therefore in particular that for any λ

$$l{:}=\lambda\,;\ h{:}\in\overline{H};\ Op;\ \mathcal{H}.(h/l)\ =\ \lg\mid H\mid\ . \tag{20.3}$$

Next from Fig. 20.1 we know that the entropy of $\overline{H}$ is maximal, and therefore (20.3) implies that the output distribution of h after executing $h{:}\in\overline{H};\ Op$ must be uniform as well. Put another way, $h{:}\in\overline{H};\ Op$ satisfies the refinement

$$l{:}\in L;\ h{:}\in\overline{H}\ \sqsubseteq\ h{:}\in\overline{H};\ Op\,,$$

and the lemma follows. □

We end this section with the result that shows that uniform preserving and permutations are equivalent to security.

Theorem 20.4.7 *The following are equivalent for program Op in $\mathcal{MS}$.*

Op is secure (Def. 20.4.2)
iff Op is uniform preserving
iff Op is a secure permutation of H.

Proof. From Lem. 20.4.6 and Lem. 20.4.4 we see that the theorem follows provided we show that "uniform preserving" implies "is a secure permutation".

We prove the contrapositive. Suppose then that Op is not a secure permutation with respect to H. That means that there is some deterministic refinement Op' of Op and some initial value λ taken by l such that Op' does not act as a permutation on H when initially $l=\lambda$. Thus it follows that the distribution of h after execution of $h{:}\in\overline{H};\ Op'$ from initial state with $l=\lambda$ is not uniform over the whole of H.

That is the same as saying that

$$l{:}\in L;\ h{:}\in\overline{H}\ \not\sqsubseteq\ h{:}\in\overline{H};\ Op'\,,$$

which shows that Op' and therefore Op is not uniform preserving. The theorem now follows. □

Unfortunately having removed probability (by focusing on secure permtations) has not yet simplified the analysis sufficiently. We end this section by considering a consequence of Thm. 20.4.7 that further reduces work in the analysis.

Thm. 20.4.7 implies that the analysis can be perfomed entirely within a model of standard (non-probabilistic) programs, and we will use the predicate transformer model since it is equivalent to relational models.

A predicate transformer over S is a function $\mathbb{P}S \to \mathbb{P}S$, monotonic with respect to the subset ordering on $\mathbb{P}S$. If Op is a standard program, then we write $wp.Op$ for the predicate transformer interpretation — that means that for any postcondition $post$, precondition $wp.Op.post$ is the weakest precondition which guarantees that Op will establish $post$.

Some standard properties of predicate transformers we use are that, for Op expressed in the non-probabilistic fragment of the programming language set out in Fig. 20.2, the transformer $wp.Op$ distributes $\wedge$; if Op is deterministic and terminating then $wp.Op$ distributes $\vee$ and $\neg$; and $wp.(x{:}\in X).post = (\forall x{:}X \cdot post)$.

With those conventions we have the following corollary to Thm. 20.4.7:

Corollary 20.4.8 *If Op is secure then, for any postcondition post and deterministic refinement Op' of Op,*

$$(\forall l{:}L \cdot (\exists h{:}H \cdot post)) \ \Rrightarrow\ (\forall l{:}L \cdot (\exists h{:}H \cdot wp.Op'.post))\,.$$

Proof. By Thm. 20.4.7 we know that security of P implies that it is a secure permutation and therefore by Def. 20.4.3 all deterministic refinements Op' of it satisfy

$$Op';\, h{:}\in H \ = \ h{:}\in H;\, Op'\,.$$

We continue our reasoning in predicate transformers from this point: for any $post$,

$$wp.(Op';\, h{:}\in H).\neg post \ \equiv\ wp.(h{:}\in H;\, Op').\neg post$$

iff $\qquad$ Op' is deterministic and terminating

$$\neg wp.Op'.(\exists h{:}H \cdot post) \ \equiv\ \neg(\exists h{:}H \cdot wp.Op'.post)$$

iff $\qquad wp.Op'.(\exists h{:}H \cdot post) \ \equiv\ (\exists h{:}H \cdot wp.Op'.post)$

implies $\qquad (\forall l{:}L \cdot (\exists h{:}H \cdot post)) \ \Rrightarrow\ (\exists h{:}H \cdot wp.Op'.post)$ $\qquad$ see below

iff $\qquad$ l not free in lhs

$$(\forall l{:}L \cdot (\exists h{:}H \cdot post)) \ \Rrightarrow\ (\forall l{:}L \cdot (\exists h{:}H \cdot wp.Op'.post))\,.$$

For the deferred justification we note that for any $post'$

$$(\forall l{:}L;\, h{:}H \cdot post') \ \Rrightarrow\ wp.Op'.post'\,,$$

and that when $post'$ is itself $(\exists h{:}H \cdot post)$, the universal $\forall h$ can be dropped. □

The implication of Cor. 20.4.8 is that security can be checked by considering a reduced set of postconditions. The condition on its left-hand side, namely

$$(\forall\, l{:}\, L \cdot (\exists\, h{:}\, H \cdot post))\,, \tag{20.4}$$

restricts predicates rather than programs. We say that a postcondition satisfying (20.4) is *total in l*. Thus we have the following corollary.

Corollary 20.4.9 *Op is secure provided that whenever post is total in l then so is wp.Op.post.*

Proof. Suppose for the contrapositive that for some total *post* we have non-total *wp.Op.post*. We choose deterministic refinement Op' of *Op* so that $wp.Op.post \equiv wp.Op'.post$, and observe from Cor. 20.4.8 that *Op* is therefore not secure. □

20.4.2 Comparisons with other work

Finally we end this section by considering other formulations of security in the imperative style. Typically they do not *maintain* the privacy of *High*'s variables, but rather seek to preserve secrecy of their initial values [4]. We call this property "weakly secure".

Definition 20.4.10 *We say that Op is* weakly secure *if it does not reveal any information about the initial values of h. This is equivalent to [4]*

$$Op;\ h{:}\in H \quad \sqsubseteq \quad h{:}\in H;\ Op\,.$$

As we noted earlier, this is only half of our Def. 20.4.3, which is therefore stronger.. In fact the definition we give in Sec. 20.4 is stronger than Def. 20.4.10. Intuitively, if *Op* reveals information about the initial value of *h* then it most likely reveals information about the final value as well, since that in many cases can be inferred from the text of the program. That intuition is formalised in the next lemma (whose proof is omitted).

Lemma 20.4.11 *Security of final values implies security of initial values (weak security) for standard deterministic programs.*

The difference between the two approaches is evident with program fragment $h{:=}\ 0$, which is weakly secure, but not secure. That is because the initial value of *h* has been obliterated by the call, and hence cannot be known; but the current value of *h* is certainly disclosed, on the assumption that *Low* has a copy of the program specification.

In fact Rushby's formulation of non-interference would also certify that the above operation is secure, if *Low* can call it, since his view of the system is certainly unaffected, whatever *High* does.

20.5 Example: The secure file store

We illustrate our definitions on the 'secure filestore' example [10].

A filestore comprises named files, and each file is classified as having either 'high' or 'low' security status. Low-level- and high-level users operate on the two classifications separately, with the usual operations such as *create*, *open*, *read*, *write*, *close* and *delete*; and the intention is that a low-level user cannot discover anything — via those operations — about the high-level portion of the filestore.

The security breach in this example occurs when the low-level user attempts to create a file whose name happens to be the same as an extant high-level file. When the system rejects the create operation ("File name exists."), the low-level user has learned the name of a high-level file.

We treat the example at increasing levels of complexity/realism.

20.5.1 The 'bare bones'

Since the security breach occurs independently for each (potential) filename, we concentrate first on a single, arbitrary name which therefore — since it is constant — we need not model explicitly. To refer to it, however, we will call it *fileName*.

Our state space S comprises two Boolean variables h and l, each indicating whether the *fileName* exists at that level, and a third Boolean r for 'result', to indicate success or failure of operations. Invariant will be that the file cannot exist at both levels, which is in fact the source of the insecurity:

$$S \;:=\; h, l, r : Boolean\,.$$

The create-a-low-level-file operation is then

```
CreateLow0  :=  if ¬(h ∨ l) then l, r := true, true
                            else r := false
                fi .
```

It is the conditional that prevents the creation of a file at both levels.

Our security criterion Cor. 20.4.9 requires that total postconditions give total preconditions; and in this case our security classification gives "total" the meaning "for every l, r there must be an h". For this example we demonstrate insecurity with the (total) postcondition $h = r$; the weakest precondition through *CreateLow0* is then the partial $l \wedge \neg h$ (which has no satisfying h-value when l is false).

The intuition for choosing such postconditions is this: if there is a security leak, then from some initial value of the low-level variables (it's our choice — we are looking for leaks) there is a possible low-level result (again our choice, if demonic nondeterminism allows more than one low-level result) for which we *know* that certain high-level values are not possible. That's the knowledge that could have leaked.

In this case, when l is initially **false**, we know that after *CreateLow0* it cannot be that r and h are equal: for if the operation succeeded ($r =$ **true**) there must be no high-level file (h cannot be true); and, if it failed ($r =$ **false**), then there must be a high-level file (h cannot be false).

```
$CreateLow1(n\colon FILENAME;\ r\colon Boolean)\ :=$

    if $n \notin domain.s$ then $s[n] := (ll, emptyData);\ r := $ true
                   else $r :=$ false
    fi ,

$ReadLow1(n\colon FILENAME;\ d\colon DATA;\ r\colon Boolean)\ :=$

    if $n \in domain.s \wedge s[n].level{=}ll$ then $d := s[n].data;\ r :=$ true
                                 else $r :=$ false
    fi ,

$WriteLow1(n\colon FILENAME;\ d\colon DATA;\ r\colon Boolean)\ :=$

    if $n \in domain.s \wedge s[n].level{=}ll$ then $s[n].data := d;\ r :=$ true
                                 else $r :=$ false
    fi ,
```

Figure 20.3. Selection of file-system operations

20.5.2 Adding filenames and data

The scenario of the previous section reveals the essence of the insecurity. In this section we show how the essential insecurity remains within a more realistic framework: we add filenames and data to the system.

Let a file comprise some data and an indication of its level, and let the system state be a (partial) mapping from filenames to files:

$$\begin{array}{rcl} FILE & := & (level : \{hh, ll\};\ data : DATA) \\ S & := & (s\colon FILENAME \rightarrow FILE\,)\,. \end{array} \tag{20.5}$$

Note that the invariant 'can't be both high and low' is 'built-in' to this model, since the *level* component of *FILE* can have only one value and *s* is a function. Rather than clutter the system state with input and output parameters, we will this time write them with the operations, that is in a more conventional way; formally, however, we continue to treat them as part of the state. A selection of the extended operations is given in Fig. 20.3.

Specifying the system as above — i.e. in a 'natural' way, without planning in advance for our security criterion — reveals a potential problem: which part of *s* in (20.5) is the hidden variable?

We formalise the intuition that 'it's the files with *level* set to *hh*' by imagining an alternative formulation partitioned between the two, splitting S into S_h and S_l with the obvious coupling invaraint linking s with s_h and s_l. Then by analogy with our treatment of *CreateLow0* we start off by considering postcondition $(n \in domain.s_h) \equiv (r = \mathbf{true})$. The precondition of that with respect to *CreateLow1* is

$$n \in domain.s_l\,, \tag{20.6}$$

which is not total.

20.5.3 *Proving security* vs. *demonstrating insecurity*

The complementary question is of course whether the other operations — *ReadLow1* and *WriteLow1* — are secure according to our criteria. In principle we would consider all possible total postconditions, showing the corresponding precondition to be total as well; in practice however one tries to avoid quantifying over postconditions, as the resulting second-order reasoning can be unpleasant or even infeasible.

One approach is to use an equivalent algebraic formulation of the property, and then to reason within the appropriate program algebra. For example, our criterion 'total post- yields pre-' can be written algebraically as

$$l{:}\in_{\sqcap} L;\;\; h{:}\in_{\sqcup} H \;\;\sqsubseteq\;\; h{:}\in_{\sqcup} H;\;\; Op\,,$$

where we now subscript the nondeterministic choices to indicate whether they are demonic or angelic. That, when specialised to operation *ReadLow1* in the file system, becomes

$$\begin{array}{ll} s_l, n, d, r \;\; {:}\in_{\sqcap} & S_l, FILENAME, DATA, Boolean; \\ s_h{:}\in_{\sqcup} S_h & \end{array}$$

$$\sqsubseteq s_h{:}\in_{\sqcup} S_h;\;\; ReadLow1\,.$$

Looking at the right-hand side, we note that the condition $n \in domain.s \wedge s[n].level{=}ll$ in *ReadLow1* can be rewritten

$$n \in domain.s_l\,,$$

because of the way in which s, s_h and s_l are related by the coupling invariant. Then *ReadLow1* does not refer to s_h at all; the two statements commute; and the refinement becomes trivial, as any criterion of this kind should when the high-level variables are not mentioned.

20.6 The Refinement Paradox

This so-called paradox is a modern manifestation of a (lack of) distinction between 'underspecification' and (demonic) nondeterminism. The most widely used formal models of programming — Hoare-triple, predicate-transformer, relational — rightly[9] do not distinguish these two specifications:

Prog0 ‘Always set l to 0, or always set l to 1’; and

Prog1 ‘Always set l to either 0 or 1’.

In our context the former is secure, since neither program $l\!:=\ 0$ nor program $l\!:=1$ can reveal anything about h; yet the latter is insecure, because it can behave like (equivalently, can be refined to) the program $l\!:=h$. Yet the two programs are identified in the usual models, and that is the ‘paradox’.

The issue arises again in the interaction between any two of the three forms of nondeterminism demonic, angelic and probabilistic. Consider the two systems

$$\begin{array}{ll} & h\!:=0\ {}_{1/2}\!\oplus 1;\quad \mathit{Prog0} \\ \text{and} & h\!:=0\ {}_{1/2}\!\oplus 1;\quad \mathit{Prog1}\,. \end{array}$$

In the first case, on termination $h\ =\ l$ with probability $1/2$ exactly; but in the second case that probability can be as low as zero.

In our security context, the point is that we would like to be able to specify a modified *Prog1* that ‘sets l to 0 or 1 but without looking at h’, lying between the two extremes above. To do so would mean we had combined security and abstraction, thus avoiding the paradox.

20.6.1 Some algebraic difficulties

It is tempting to look for a model supporting ‘nuanced’ demonic choice, ranging from the ‘oblivious’ choice that sees nothing through an intermediate ‘cannot see h’ choice to the familiar ‘omniscient’ choice. But simple algebraic experiments show how tricky this can be.

Write $\sqcap_h$ for the choice that can’t look at h: then oblivious choice is $\sqcap_{h,l}$ and our normal omniscient choice remains $\sqcap$ (because it can’t look at nothing). Now consider this reasoning: we have

$$\begin{array}{lll} & l\!:=0 \sqcap_h 1 & \\ = & l\!:=0;\quad l\!:=0 \sqcap_h 1; & \text{Principle 1, below} \\ = & (l\!:=h;\, l\!:=0);\quad l\!:=0 \sqcap_h 1; & \text{Principle 2} \\ = & l\!:=h;\quad (l\!:=0;\, l\!:=0 \sqcap_h 1); & \text{associativity of composition} \\ = & l\!:=h;\quad l\!:=0 \sqcap_h 1; & \text{Principle 1} \\ \sqsubseteq & l\!:=h;\quad \textbf{skip}; & \text{allowed by } \sqcap_h \\ = & l\!:=h\,. & \end{array}$$

To avoid that insupportable conclusion (without giving up associativity, refinement or the meaning of **skip**), we must forgo either

Principle 1: Demonic choice $\sqcap_h$ can be treated as simple $\sqcap$ in contexts that don’t mention h, or

[9] ... because in everyday programming situations it does not matter.

Principle 2: Variables h, l may be treated normally in contexts that don't mention $\sqcap_h$.

Note that we describe the principles syntactically, since their purpose is a practical one: to lay down the restrictions for a simple (syntactic) algebra. But that is not to say that they do not have operational interpretations as well: for example, Principle 1 implies that $\sqcap_h$ can 'see past program states', since otherwise the overwriting of l's value might destroy information that $\sqcap_h$ 'could have used' and the equality would not hold.[10]

20.6.2 *The 'quantum' model*

We explore the consequences of abandoning Principle 2, choosing that alternative for two reasons. First, any program structure that supported scoping (of h) would make it unnecessary in practice to write $\sqcap_h$, since an implied subscript h on plain $\sqcap$ could be inferred from h's not being in scope: that makes loss of Principle 1 even more problematic in practice.

Second, the statement $l := h$ at least 'crosses the encapsulation boundary', and so alerts us to our having to employ 'enhanced' reasoning about it. Indeed, the statement would probably have to be located within the module in which h was declared, a further advantage. So we abandon $(l := h;\ l := 0) \ = \ l := 0$, if h is inside a module and l outside of it: variable h will be treated differently.

The *quantum* model treats secure variables h as a distribution, separately — in fact, within — the overall distribution on the state space as a whole. Thus for example, in the ordinary (probabilistic) model the program

$$h := 0 \,{}_{1/2}\!\oplus 1; \quad l := 0 \,{}_{1/2}\!\oplus 1$$

results in a uniform distribution between four final possiblilities; we could write it (pairing as (l, h))

$$(0,0) \bullet 1/4, \quad (0,1) \bullet 1/4, \quad (1,0) \bullet 1/4, \quad (1,1) \bullet 1/4\,. \tag{20.7}$$

In the quantum model, instead the result is a $1/2$-$1/2$ distribution (in l) between two further $1/2$-$1/2$ distributions (in h); we would write

$$\begin{array}{ll} (0,\ (0 \bullet 1/2, 1 \bullet 1/2)) & \bullet\, 1/2 \\ (1,\ (0 \bullet 1/2, 1 \bullet 1/2)) & \bullet\, 1/2 \end{array} \tag{20.8}$$

The effect of the assignment $l := h$ is to move from state (20.8) to state (20.7), in effect 'collapsing' the hidden distribution; and a subsequent $l := \ 0$ does not undo the collapse.

[10]In fact, the standard refinement $l := 0;\ l := 0 \sqcap 1 \ \sqsubseteq$ **skip** suggests the same about ordinary $\sqcap$.

20.6.3 Implications for security

With the more complicated model, we are able — as we should be — to combine demonic choice and security: for example, the statement $l := 0 \sqcap_h 1$ is no longer refined by $l := h$ if h is hidden. The details of the model, however, remain complex (as do those of other similar models [5]); and it will require further work to see whether it can be made practical.

References

[1] D. Denning. *Cryptography and Data Security.* Addison-Wesley, 1983.

[2] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[3] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, 1982.

[4] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37((1-3)):113–138, May 2000.

[5] G. Lowe. Defining information flow. Technical Report 1999/3, Dept. Maths. and Comp. Sci., Univ. Leicester, 1999.

[6] Carroll Morgan and Annabelle McIver. *pGCL*: Formal reasoning for random algorithms. *South African Computer Journal*, 22, March 1999. Also available at [8].

[7] C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[8] PSG. Probabilistic Systems Group: Collected reports.
`http://web.comlab.ox.ac.uk/oucl/research/areas/probs/bibliography.html`.

[9] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall.

[10] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1), 1996.

[11] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

[12] P.Y.A. Ryan. A CSP formulation of interference. *Cipher*, 19-27, 1991.

[13] C.E. Shannon A mathematical theory of communication. *Bell Syst. Tech. J.* Vol 27, 379–423 and 623–656, 1948.