Experiments in Information Flow Analysis

Annabelle McIver(✉)

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

Abstract. Designing programs that do not leak confidential information continues to be a challenge. Part of the difficulty arises when partial information leaks are inevitable, implying that design interventions can only limit rather than eliminate their impact.

We show, by example, how to gain a better understanding of the consequences of information leaks by modelling what adversaries might be able to do with any leaked information.

Our presentation is based on the theory of Quantitative Information Flow, but takes an experimental approach to explore potential vulnerabilities in program designs. We make use of the tool Kuifje [12] to interpret a small programming language in a probabilistic semantics that supports quantitative information flow analysis.

Keywords: Quantitative Information Flow · Probabilistic program semantics · Security · Confidentiality

1 Introduction

Consider the following scenarios:

(I) Paris needs a secretary and intends to pick the best candidate from a list of N applicants. His father King Priam knows Paris to be extremely impulsive making it highly likely that he will hire someone immediately after interviewing them, thereby running the risk that he might miss the best candidate. Knowing that he will not be able to suppress entirely Paris' impetuous nature, Priam persuades Paris to interview first a certain number n, and *only then* to select the next best from the remaining candidates. In that way Priam hopes to improve Paris' chance of hiring the best secretary. "But father, what value of n should I choose so that this strategy increases the prospect of my finding the best candidate overall? Wouldn't I be better off just relying on my instinct?"

This research was supported by the Australian Research Council Grant DP140101119.

© Springer Nature Switzerland AG 2019

(II) Remus wants to hack into Romulus' account. Romulus knows this and wonders whether he should upgrade his system to include the latest password checking software. He has lately been reading about a worrying "side channel" that could reveal prefixes of his real password. However he is not sure whether the new software (which sounds quite complicated, and is very expensive) is worth it or not. How does the claimed defence against the side channel actually protect his password? What will be the impact on useability?

Both these scenarios share the same basic ingredients: there is a player/adversary who wants to make a decision within a context of partial information. In order to answer the questions posed by Paris and Romulus, we need to determine whether the information available is actually useful given their respective objectives. The result of such a determination for Paris might be that he would be able to form some judgement about the circumstances under which his proposed strategy could yield the best outcome for him. For Romulus he might be better able to decide whether his brother could marshal the resources required to breach the security defences in whichever password checking software he decides to install.

The aim of this paper is to illustrate, by example, how to analyse the consequences of any information leaks in program designs. The analysis is supported by the theory of Quantitative Information Flow (QIF). In particular QIF formalises how adversaries stand to benefit from any information leaks, allowing alternative designs to be compared in terms of specific adversarial scenarios. The presentation takes an experimental approach with the goal of understanding the impact and extent of partial information leaks. In so doing we acknowledge that this form analysis is almost certain to be incomplete, however its strength is to highlight relative strengths and weaknesses of program designs that handle confidential information.

In Sect. 2 we summarise the main ideas of QIF and show in Sects. 3 and 4 how they can be applied to the scenarios outlined above. We also touch on some logical features of modelling information flow in Sect. 5. In Sect. 6 we describe briefly the features of a recent QIF analysis tool Kuifje [12], and sketch how it can be used to describe the scenarios above and to carry out experiments to assess their information leaks.

2 Review of Quantitative Information Flow

The informal idea of a secret is that it is something about which there is some uncertainty, and the greater the uncertainty the more difficult it is to discover exactly what the secret is. For example, one's mother's maiden name might not be generally known, but if the nationality of one's mother is leaked, then it might rule out some possible names and make others more likely. Similarly, when some information about a secret becomes available to an observer (often referred to as an adversary) the uncertainty is reduced, and it becomes easier to guess its value. If that happens, we say that information (about the secret) has leaked.

Quantitative Information Flow (QIF) makes this intuition mathematically precise. Given a range of possible secret values of (finite) type $\mathcal{X}$, we model a secret as a probability distribution of type $\mathbb{D}\mathcal{X}$, because it ascribes "probabilistic uncertainty" to the secret's exact value. Given $\pi{:}\,\mathbb{D}\mathcal{X}$, we write π_x for the probability that π assigns to $x{:}\,\mathcal{X}$, with the idea that the more likely it is that the real value is some specific x, then the closer π_x will be to 1. Normally the uniform distribution over $\mathcal{X}$ models a secret which could equally take any one of the possible values drawn from its type and we might say that, beyond the existence of the secret, nothing else is known. There could, of course, be many reasons for using some other distribution, for example if the secret was the height of an individual then a normal distribution might be more realistic. In any case, once we have a secret, we are interested in analysing whether an algorithm, or protocol, that uses it might leak some information about it. To do this we define a measure for uncertainty, and use it to compare the uncertainty of the secret before and after executing the algorithm. If we find that the two measurements are different then we can say that there has been an information leak.

The original QIF analyses of information leaks in computer systems [3,4] used Shannon entropy [17] to measure uncertainty because it captures the idea that more uncertainty implies "more secrecy", and indeed the uniform distribution corresponds to maximum Shannon entropy (corresponding to maximum "Shannon uncertainty"). More recent treatments have shown that Shannon entropy is not the best way to measure uncertainty in security contexts because it does not model scenarios relevant to the goals of the adversary. In particular there are some circumstances where a Shannon analysis actually gives a more favourable assessment of security than is actually warranted if the adversary's motivation is taken into account [18].

Alvim et al. [2] proposed a more general notion of uncertainty based on "gain functions". This is the notion we will use. A *gain function* measures a secret's uncertainty according to how it affects an adversary's actions within a given scenario. We write $\mathcal{W}$ for a (usually finite) set of actions available to an adversary corresponding to an "attack scenario" where the adversary tries to guess something (e.g. some property) about the secret. For a given secret $x{:}\,\mathcal{X}$, an adversary's choice of $w{:}\,\mathcal{W}$ results in the adversary gaining something beneficial to his objective. This gain can vary depending on the adversary's choice (w) and the exact value of the secret (x). The more effective is the adversary's choice in how to act, the more he is able to overcome any uncertainty concerning the secret's value.

Definition 1. *Given a type $\mathcal{X}$ of secrets, a gain function $g{:}\,\mathcal{W}{\times}\mathcal{X} \to \mathbb{R}$ is a real-valued function such that $g(w,x)$ determines the gain to an adversary if he chooses w and the secret is x.*

A simple example of a gain function is bv, where $\mathcal{W}{:=}\,\mathcal{X}$, and

$$\mathsf{bv}(x,x') \quad := \quad 1 \;\; \textit{if} \;\; x = x' \;\; \textit{else} \;\; 0. \tag{1}$$

For this scenario, the adversary's goal is to determine the exact value of the secret, so he receives a gain of 1 if he correctly guesses the value of a secret, and zero otherwise. Assuming that he knows the range of possible secrets $\mathcal{X}$, he therefore has $\mathcal{W} := \mathcal{X}$ for his set of possible guesses.

Elsewhere the utility and expressivity of gain functions for measuring various attack scenarios relevant to security have been explored [1,2]. Given a gain function we define the *vulnerability* of a secret in $\mathbb{D}\mathcal{X}$ relative to the scenario it describes: it is the maximum average gain to an adversary. More explicitly, for each guess w, the adversary's average gain relative to π is $\sum_{x\in\mathcal{X}} g(w,x)\times\pi_x$; thus his maximum average gain is the guess that yields the greatest average gain.

Definition 2. *Let $g{:}\,\mathcal{W}\times\mathcal{X}\rightarrow\mathbb{R}$ be a gain function, and $\pi{:}\,\mathbb{D}\mathcal{X}$ be a secret. The* vulnerability *$V_g[\pi]$ of the secret wrt. g is:*

$$V_g[\pi] \quad := \quad \max_{w\in\mathcal{W}} \sum_{x\in\mathcal{X}} g(w,x)\times\pi_x .$$

For a secret $\pi{:}\,\mathbb{D}\mathcal{X}$, the vulnerability wrt. bv is $V_{\mathsf{bv}}[\pi] := \max_{x{:}\,\mathcal{X}} \pi_x$, i.e. the maximum probability assigned by π to possible values of x. The adversary's best strategy for optimising his gain would therefore be to choose the value x that corresponds to the maximum probability under π. This vulnerability V_{bv} is called *Bayes Vulnerability.*

A *mechanism* is an abstract model of a protocol or algorithm that uses secrets. As the mechanism executes we assume that there are a number of observables that can depend on the actual value of the secret. We define $\mathcal{Y}$ to be the type for observables. The model of a mechanism now assigns a probability that $y{:}\,\mathcal{Y}$ can be observed given that the secret is x. Such observables could be sample timings in a timing analysis in cryptography, for example.

Definition 3. *A* mechanism *is a stochastic channel*[1] $C{:}\,\mathcal{X}\times\mathcal{Y}\rightarrow[0,1]$. *The value C_{xy} is the probability that y is observed given that the secret is x.*

Given a (prior) secret $\pi{:}\,\mathbb{D}\mathcal{X}$ and mechanism C we write $\pi\rangle C$ for the joint distribution *in $\mathcal{X}\times\mathcal{Y}$ defined*

$$(\pi\rangle C)_{xy} \quad := \quad \pi_x\times C_{xy} .$$

For each $y{:}\,\mathcal{Y}$, the marginal probability *that y is observed is $p_y := \sum_{x{:}\,\mathcal{X}} (\pi\rangle C)_{xy}$. For each observable y, the corresponding* posterior probability *of the secret is the conditional $\pi|^y{:}\,\mathbb{D}\mathcal{X}$ defined $(\pi|^y)_x := (\pi\rangle C)_{xy}/p_y$.*[2]

Intuitively, given a prior secret π and mechanism C, the entry $\pi_x\times C_{xy}$ of the joint distribution $\pi\rangle C$ is the probability that the actual secret value is x and the observation is y. This joint distribution contains two pieces of information: the probability p_y of observing y and the corresponding posterior $\pi|^y$ which represents the adversary's updated view about the uncertainty of the secret's value.

[1] Stochastic means that the rows sum to 1, i.e. $\sum_{y\in\mathcal{Y}} C_{xy} = 1$.

[2] We assume for convenience that when we write p_y the terms C, π and y are understood from the context. Notation suited for formal calculation would need to incorporate C and π explicitly.

If the vulnerability of the posterior increases, then information about the secret has leaked and the adversary can use it to increase his gain by changing how he chooses to act. The adversary's average overall gain, taking the observations into account, is defined to be the average posterior vulnerability (i.e. the average gain of each posterior distribution, weighted according to their respective marginals):

$$V_g[\pi\rangle C] := \sum_{y\in\mathcal{Y}} p_y \times V_g[\pi|^y] \ , \qquad \textit{where } p_y,\ \pi|^y \textit{ are defined at Definition 3.} \tag{2}$$

Now that we have Definitions 2 and 3 we can start to investigate whether the information leaked through observations $\mathcal{Y}$ actually have an impact in terms of whether it is useful to an adversary. It is easy to see that for any gain function g, prior π and mechanism C we have that $V_g[\pi] \leq V_g[\pi\rangle C]$. In fact the greater the difference between the prior and posterior vulnerability, the more the adversary is able to use the leaked information within the scenario defined by g. In our investigations of problems (I) and (II) we use the following implications of this idea.

(A) The adversary is able to use information leaked through mechanism C *only* when $V_g[\pi] < V_g[\pi\rangle C]$. In the case that the prior and posterior vulnerability are the same then, although information has leaked, the adversary acting within a scenario defined by g is not able to use the information in any way that benefits him. Of course there could be other scenarios where the information could prove to be advantageous, but those scenarios would correspond to a different gain function. If $V_g[\pi] = V_g[\pi\rangle C]$ for all gain functions g then in fact the channel has leaked no information at all that can be used by any adversary.

(B) For each observation y there is an action w that an adversary can pick to optimise his overall gain. This corresponds roughly to a strategy that an adversary might follow in "attacking" a mechanism C. Where here "attack" is a passive attack in the sense that the adversary observes the outputs of the mechanism C and then revises his opinion about the uncertainty of the secret to be that of the relevant posterior. This might mean that he can benefit by changing his mind about how to act, after he observes the mechanism.

(C) If P, Q are two different mechanisms corresponding to different designs of say a system, we can compare the information leaks as follows. If $V_g[\pi\rangle P] > V_g[\pi\rangle Q]$ then we can say that P leaks more than Q for the scenario defined by g in the context of prior π. If it turns out that $V_g[\pi\rangle P] \geq V_g[\pi\rangle Q]$ for all g and π then we can say that Q is more secure than P, written $P \sqsubseteq Q$, because in every scenario Q leaks less useful information than does P. Indeed any mechanism that only has a single observable leaks nothing at all and so it is maximal wrt. $\sqsubseteq$. We explore some aspects of $\sqsubseteq$ in Sect. 5.

In the remainder of the paper we illustrate these ideas by showing how adversaries can reason about leaked information, and how defenders can understand potential vulnerabilities of programs that partially leak information. We note that some of our examples are described using a small programming language, and therefore a fully formal treatment needs a model that can account for both state updates and information leaks. This can be done using a generalisation of Hidden Markov Models [15]; however the principal focus in all our examples is information flows concerning secrets that are initialised and never subsequently updated, and therefore any variable updates are incidental. In these cases a simple channel model, as summarised here, is sufficient to understand the information flow properties. We do however touch on some features of the programming language which, although illustrated only on very simple programs, nevertheless apply more generally in the more detailed Hidden Markov Model.

3 Experiment (I): When Is a Leak Not a (Useful) Leak?

The secretary problem and its variations [6] have a long history having been originally posed around 1955 although the earliest published accounts seem to date back to the 1960's [7–9]. They are classic problems in reasoning with incomplete information. Here we revisit the basic problem in order to illustrate the relationship between gain functions and adversarial scenarios and how gain functions can be used to model what an adversary can usefully do with partial information.

We assume that there are N candidates who have applied to be Paris' secretary and that they are numbered $1, \ldots, N$ in order of their interview schedule. However their *ability rank* in terms of their fitness for the position is modelled as a permutation $\sigma : \{1, \ldots, N\} \to \{1, \ldots, N\}$, and we assume that the precise σ is unknown to Paris, who is therefore the adversary. The best candidate scheduled in time slot i will have ability rank $\sigma[i] = N$, and the worst, interviewed at slot j will have $\sigma[j] = 1$. We write Σ for the set of all such permutations. We model Paris' prior knowledge of σ as the uniform probability distribution $u{:}\,\mathbb{D}\Sigma$—i.e. $u_\sigma = 1/N!$ for each $\sigma{:}\,\Sigma$ reflecting the idea that initially Paris cannot distinguish the candidates on suitability for the position.

When Paris interviews candidate i, he does not learn $\sigma[i]$, but he does learn how that candidate ranks in comparison to the others he has already seen. This implies that after the first interview all he can do is establish an initial baseline for subsequent comparison. For instance, after the second interview he will learn either that $\sigma[1]<\sigma[2]$ or that $\sigma[2]<\sigma[1]$. We write C_n for the channel that has as observables the relative rankings of the first n candidates. This means that C_1 has one observation (and so leaks nothing), C_2 has two observations (as above), C_3 has six observations, and in general C_n has $n!$ observations. Thus when $n = N$, C_N leaks the precise rankings of all candidates (and in so doing identifies σ). In Fig. 1 we illustrate the channel C_2 in the situation where there are only 3 candidates.

But what exactly can Paris do with these observations to help him select the best candidate, i.e. to enable him to offer the position to the candidate i that satisfies $\sigma[i] = N$? Since he makes an offer directly to candidate i at the end of the i'th interview, the risk is that he offers the job too early (he hasn't yet interviewed the best candidate) or too late (the best candidate has already been let go).

C_2	$(\sigma[1]{<}\sigma[2])$	$(\sigma[2]{<}\sigma[1])$
σ_1	1	0
σ_2	0	1
σ_3	1	0
σ_4	1	0
σ_5	0	1
σ_6	0	1

There are six possible permutations $\sigma_1, \ldots \sigma_6$. These are secret and therefore label the rows of C_2. Next, C_2's columns are labelled by the possible observations after the first two interviews. Paris is able to observe only the relative ordering of those two candidates. Either the first candidate is not as good as the second $(\sigma[1]{<}\sigma[2])$ or vice versa $(\sigma[2]{<}\sigma[1])$.
If we let σ_1 be the permutation with relative ordering $\sigma_1[1]{<}\sigma_1[2]{<}\sigma_1[3]$, and σ_2 be the permutation with relative ordering $\sigma_2[2]{<}\sigma_2[1]{<}\sigma_2[3]$ then Paris would record that "$(\sigma[1]{<}\sigma[2])$" for σ_1; this is denoted by a 1 in the first column. For σ_2 Paris would record the opposite, and therefore a 1 appears in the second column for σ_2.

Fig. 1. Information flow channel C_2 after interviewing 2 candidates from a total of 3

In any information flow problem, there is little point in measuring how much information is released if that measurement does not pertain to the actual decision making mechanism that an adversary uses. For example suppose that $N = 6$. A traditional information flow analysis might concentrate on measuring Shannon entropy as a way to get a handle on Paris' uncertainty and how it changes as he interviews more candidates. At the beginning, Paris knows nothing, and so his uncertainty as measured by Shannon entropy over the uniform distribution of 6! orderings is $\log_2(6!) \approx 9.5$. He doesn't know about 9 bits of hidden information. After two interviews, his uncertainty has dropped to $\log_2(360) \approx 8.5$, and after three interviews it becomes $\log_2(120) \approx 7$. The question is, how do these numbers help Paris make his decision? If candidate 2 is better than candidate 1 should he make the appointment? If he waits, should he make the appointment if candidate 3 is the best so far? With as much as 7 bits from a total of 9 still uncertain, maybe it would be better to wait?

Of course a detailed mathematical analysis [6] shows that if Paris' objective is to *maximise his probability* of selecting the best candidate using *only* the actions he has available (i.e. "appoint now" or "wait") then it would indeed be best for him **not** to wait but rather to appoint candidate 3 immediately if they present as the best so far. In fact the numbers taken from a Shannon analysis do not appear to help Paris at all to decide what he should do because Shannon entropy is not designed for analysing his objective when faced with his particular scenario and the set of actions he has available.

A more helpful analysis is to use a gain function designed to describe exactly Paris' possible actions so that his ability to use any information leaks –by enabling the capability to favour one action over another– can be evaluated. Recall that after interviewing each candidate Paris can choose to perform two actions: either to "appoint immediately" (a) or to "wait" (w) until the next best.

We let $\mathcal{W} := \{a, w\}$ represent the set of those actions after the n'th interview. Next we need to define the associated gains. For action a, the gain is 1 if he manages to appoint the overall maximum, i.e. $g_n(a, \sigma) = 1$ exactly if $\sigma[n] = N$. For w Paris does not appoint the n'th candidate but instead continues to interview candidates and will appoint the next best candidate he has seen thus far. His gain is 1 if that appointment is best overall, thus $g_n(w, \sigma) = 1$ if and only if the first $k{>}n$ which is strictly better than all candidates seen so far turns out to be the maximum overall.[3] We write $\sigma[i, j]$ for the subsequence of σ consisting of the interval between i and j inclusive, and $\sigma(i, j)$ for the subsequence consisting of the interval strictly between i and j (i.e. exclusive of $\sigma[i]$ and $\sigma[j]$). Furthermore we write $\sqcup\rho$ for the maximum item in a given subsequence ρ. We can now define Paris' gain function as follows:

$$
\begin{aligned}
g_n(a, \sigma) \quad &:= \quad 1 \;\; \textit{iff} \;\; \sigma[n] = N & (3)\\
g_n(w, \sigma) \quad &:= \quad 1 \;\; \textit{iff} \;\; \exists (N{\geq}k{>}n) \;\; s.t. \sqcup\sigma(n, k) < \sqcup\sigma[1, n] < N = \sigma[k]. & (4)
\end{aligned}
$$

Given uniform prior u (over the possible $N!$ permutations), we can approximate Paris' residual uncertainty after interviewing n candidates by considering the posterior vulnerability $V_{g_n}[u \rangle C_n]$. Using (A) from Sect. 2 we can evaluate whether the information leaked is useful by comparing $V_{g_n}[u \rangle C_n]$ with the prior vulnerability $V_{g_n}[u]$; next we can use (B) from Sect. 2 to examine, for each posterior which of a or w is Paris' best action relative to each observation. In the case that N is 6, we can show that:

$$
V_{g_2}[u] = V_{g_2}[u \rangle C_2] \;, \quad \textit{but} \;\; V_{g_3}[u] < V_{g_3}[u \rangle C_3],
$$

i.e. there is no leakage wrt. the question "should the candidate just interviewed be selected?" after two interviews because the prior and posterior vulnerabilities wrt. g_2 are the same. On the other hand the information leaked after interviewing *three* candidates is sufficient to help him decide whether to appoint the third candidate—he should do so if that third candidate is the best so far.

To see how this reasoning works, consider the situation after interviewing two candidates as described above. Note that both outcomes occur with probability 1/2 (since the prior is uniform). If we look at the two possible actions available in g_2 we can compare whether the probability of the second candidate being best overall actually changes from its initial value of $^1/_6$ with this new information. In fact it does not—the probability that the second candidate is the best overall remains at $^1/_6$, and the probability that the next best candidate is best overall also does not change, remaining at $0.427 = V_{g_2}[u \rangle C_2]$. This tells us that although some uncertainty has been resolved, it can't be used by Paris to help him with his objective—he will still not appoint candidate 2 even when they rank more highly than candidate 1.

[3] In the traditional analysis Paris only employs this strategy after interviewing approximately N/e candidates. Here we are interested in studying this strategy at each stage of the interviewing procedure.

On the other hand if we compare the case where Paris interviews three candidates, and observes that the best candidate so far is $\sigma[3]$, then there is a good chance that this is the best candidate overall. Indeed his best strategy is to appoint that candidate immediately—and he should only continue interviewing if $\sigma[3]$ rates lower than either the other two. Overall this gives him the greatest chance of appointing the best candidate which is $0.428 = V_{g_3}[u \rangle C_3]$. [4]

A similar analysis applies to other C_n's but when $N = 6$ his overall best strategy is to start appointing after $n = 3$.

We note that the analysis presented above relates only to the original formulation of the Secretary problem; the aim is to explain the well-known solution (to the Secretary problem) in terms of how an adversary acts in a context of partial information. More realistic variations of the Secretary problem have been proposed. For example the original formulation only speaks of hiring the best, whereas in reality hiring the second best might not be so bad. Analysing such alternative adversarial goals (such as choosing either the best or the second best) would require designing a gain function that specifically matches those alternative requirements.

4 Experiment (II): Comparing Designs

Consider now the scenario of Romulus who does not trust his brother Remus. Romulus decides to password protect his computer but has read that the password checker, depicted in Fig. 2, suffers from a security flaw which allows an adversary to identify prefixes of his real password. Observe that the program in Fig. 2 iteratively checks each character of an input password with the real password, so that the time it takes to finish checking depends on how many characters have been matched. Romulus wonders why anyone would design a password checker with such an obvious flaw since an easy fix would be to report whether the guess succeeds only after waiting until all the characters have been checked. But then Romulus remembers that he often mistypes his password and would not want to wait a long time between retries.

In Fig. 2 we assume that `pw` is Romulus' secret password, and that `gs` is an adversary's input when prompted by the system for an input. To carry out a QIF analysis of a timing attack, as described in Sect. 2, we would first create an information flow channel with observations describing the possible outcomes after executing the loop to termination, and then we would use it to determine a joint distribution wrt. a prior distribution. In this case the observations in the channel would be: "does not enter the loop", "iterates once", "iterates twice"

[4] Interestingly the two $V_{g_2}[u \rangle C_2] = V_{g_3}[u \rangle C_3]$—although the actions wrt. g_2 and g_3 are different, in the case they are interpreted over the scenario of 6 candidates they dictate the same behaviour. Under the g_2 strategy which becomes operational after interviewing 2 candidates, it is never to appoint candidate 2, but take the very next best. Under the g_3 case (which becomes operational after interviewing three candidates) it is to take candidate 3 if she is the best so far, or if not select the next best.

```
// Assume pw has been initialised as the secret password,
// and that gs is a input password that needs to be checked
i := 0;
ans := true;
while (ans && i<N) {
          Print (ans && i<N);       // Side Channel
          if (pw[i] != gs[i])
                    then ans := false;
          else skip;
          i++;
}
Print ans          // Information leak
```

All assignments and updates are considered unobservable; information leaks are indicated at "Print" statements. The "timing" attack is modelled as the Remus' ability to observe the number of executions of the loop.

Fig. 2. Basic password checker, with early exit

etc. There is however an alternative approach which leads to the same (logical) outcome, but offers an opportunity for QIF tool support summarised in Sect. 6. Instead of creating the channel in one hit, we interpret each program fragment as a mini-channel with its own observations. At each stage of the computation the joint distribution described at Definition 3 is determined "on the go", by amalgamating all subsequent information leaks from new mini-channels with those already accounted for. The situation is a little more complicated because state updates also have to be included. But in the examples described here, we can concentrate on the program fragments that explicitly release information about the secret `pw`, and assume that the additional state updates (to the counter i for example) have no impact on the information leaks.

To enable this approach we introduce a "Print" statement into the programming language which is interpreted as a mini-channel—"Print" statements have no effect on the computations except to transmit information to an observer during program execution. In Fig. 2 the statement `Print(ans && i<N)` occurring each time the loop body is entered transmits that fact to the adversary by emitting the observation "condition `(ans && i<N)` is true". Similarly a second statement `Print ans` transmits whether the loop terminated with `ans` true or false. To determine the overall information flow of the loop, we interpret `Print(ans && i<N)` and `Print ans` separately as channels, and use "channel sequential composition" [14] to rationalise the various sequences of observations with what the adversary already knows. It is easy to see that this use of "Print" statements logically (at least) accounts for the timing attack because it simulates an adversary's ability to determine the number of iterations of the loop, and therefore to deduce the prefix of the `pw` of length the number of iterations, because it must be the same as the corresponding prefix of the known `gs`.

```
// Assume pw has been initialised as the secret password,
// and that gs is an input password that needs to be checked

list := [0,...,n-1];                       // Indices for checking
ans := true;
       while (ans && list!=[]) {
         Print (ans && list!=[]);          // "Smudged" leak
         i := uniform(list);               // Uniform choice
         ans := ans && (pw[i] = gs[i]);
         list := list-{i};                 // Remove checked index
       }
Print ans
```

Fig. 3. Password checker, with early exit and random checking order

With this facility, Romulus can now compare various designs for password checking programs. Figure 3 models the system upgrade that Romulus is considering. Just as in Fig. 2, an early exit is permitted, but the information flow is now different because rather than checking the characters in order of index, the checking order is randomised uniformly over all possible orders. For example if Fig. 2 terminates after checking a single character of input password `gs`, then the adversary knows to begin subsequent guesses with `gs[0]`. But if Fig. 3 similarly terminates after trying a single character, all the adversary knows is that one of `gs[0]`, ... `gs[n-1]` is correct, but he does not know which one.

At first sight, this appears to offer a good trade-off between Romulus' aim of making it difficult for Remus to guess his password and his desire not to wait too long before he can try again in case he mistypes his own password. To gain better insight however we can look at some specific instances of information flow with respect to these password checking algorithms.

We assume a prior uniform distribution over all permutations of a three character password `"abc"`, i.e. we assume the adversary knows that `pw` is one of those permutations, but nothing else. Next we consider the information leaks when the various algorithms are run with input `gs = "abc"`. Since the adversary's intention is to figure out the exact value of `pw`, we use Bayes Vulnerability V_{bv} (recall 1 for definition of `bv`) to measure the uncertainty, where $\mathcal{X}$ is the set of permutations of `"abc"` and $\mathcal{W}{:=}\,\mathcal{X}$ is the set of possible guesses. [5]

Consider first the basic password checker depicted at Fig. 2. There are three amalgamated observations: the loop can terminate after 1, 2 or 3 iterations. For each observation, we compute the marginal probability and the posterior probabilities associated with the residual uncertainty (recall Definition 3). For this example the marginal, corresponding posterior and observations are as follows:

[5] In this case, we imagine that an adversary uses an input `gs` in order to extract some information. His actions described by $\mathcal{W}$ are related to what he is able to do in the face of his current uncertainty.

```
marginal  {posterior}
1 ÷ 6        {1 ÷ 1    "abc"}   ← termination after 3 iterations
1 ÷ 6        {1 ÷ 1    "acb"}   ← termination after 2 iterations
2 ÷ 3        {1 ÷ 4    "bac"    ← termination after 1 iteration
              1 ÷ 4    "bca"
              1 ÷ 4    "cab"
              1 ÷ 4    "cba"}
```

The first observation with marginal probability $1/6$, is the case where `gs` and `pw` are the same: only then will the loop run to completion. The associated posterior probability is the point distribution over `"abc"`. For the second observation –termination after 2 iterations– the marginal is also $1/6$ and the associated posterior is also a point distribution over `"acb"`. That is because if `gs[0]` matches `pw[0]` but `gs[1]` does not match `pw[1]` it must be that the second and third characters in `gs` are swapped.

The third observation is the case where the loop terminates after one iteration and it occurs with marginal probability $2/3$. This can only happen when `gs[0]` is not equal to `pw[0]` (2 possibilities), and the second and third characters can be in either order (2 more possibilities), giving $1/2{\times}2$ for each posterior probability.

Overall the (posterior) Bayes Vulnerability has increased (from $1/6$ initially relative to the uniform prior) to 1/2 because the chance of guessing the password, taking the observations into account, is now $1/2$.

Comparing now with the obfuscation of the checking order Fig. 3, we again have the situation of three possible observations, but now the uncertainty is completely reduced only in the case of 3 iterations. The resulting marginal, corresponding posterior and observations are as follows:

```
marginal    {posterior}
1 ÷ 6         {1 ÷ 1     "abc"}   ← termination after 3 iterations
2 ÷ 3         {1 ÷ 6     "acb"    ← termination after 2 iterations
               1 ÷ 6     "bac"
               1 ÷ 4     "bca"
               1 ÷ 4     "cab"
               1 ÷ 6     "cba"}
1 ÷ 6         {1 ÷ 3     "acb"    ← termination after 1 iteration
               1 ÷ 3     "bac"
               1 ÷ 3     "cba"}
```

In spite of the residual uncertainty in the case of an incorrect input, the overall posterior vulnerability of $7/18$ is only slightly lower than the $1/2$ for the original early exit Fig. 2.

For longer passwords, displaying the list of marginals and posteriors is not so informative; but still we can give the posterior Bayes vulnerability. We find for example that for Fig. 3, even with its early exit, only reduces the probability of guessing the password by half when compared with Fig. 2. For a 5 character password, Fig. 2 gives a posterior vulnerability of $1/24$, but Fig. 3 reduces it only to about half as much at $^{13}/_{600}$. Perhaps Romulus' upgrade is not really worth it after all.

5 Experiment (III): Properties of the Modelling Language

```
// h, k are secret bits, already initialised

hif (h==1) then Print (h XOR k) // Print the exclusive or of h, k
           else Print 1         // Print 1
fih
```

Fig. 4. A hidden if ... then ... else

Thus far we have considered information flows in specific scenarios. In this section we look more generally at the process of formalisation, in particular the information flow properties of the programming language itself. In standard program semantics, the idea of refinement is important because it provides a sound technique for simplifying a verification exercise. If an "abstract" description of a program can be shown to satisfy some property, then refinement can be used to show that if the actual program only exhibits consistent behaviours relative to its abstract counterpart, then it too must satisfy the property. To make all of this work in practice, the abstraction/refinement technique normally demands that program constructs are "monotonic" in the sense that applying abstraction to a component leads to an abstraction of the whole program.

In Sect. 2(C) we described an information flow refinement relative to the QIF semantics so that if $P \sqsubseteq P'$ then program P leaks more information than does program P' in all possible scenarios. In the examples we have seen above, the sequence operator used to input the results of one program P into another Q is written $P;Q$. It is an important fact that sequence is monotonic, i.e. if $P \sqsubseteq P'$ then $P;Q \sqsubseteq P';Q$. Elsewhere [14,16] define a QIF semantics for a programming language in which all operators are "monotonic" wrt. information flow—this means that improving the security of any component in a program implies that the security of the overall program is also improved. The experiment in this section investigates monotonicity for conditional branching in a context of information flow.

Consider programs at Figs. 4 and 5 which assume two secrets `h, k`. Each program is the same except for the information transmitted in the case that the secret `h` is 1—in this circumstance Fig. 4 Prints the exclusive OR of `h` and `k` whereas Fig. 5 always Prints 1. In this experiment we imagine that the adversary can only observe information transmitted via the "Print" statements, and cannot (for example) observe the branching itself. This is a "hidden if statement" denoted by "`hif` ... `fih`", and we investigate whether it can be monotonic as a (binary) program construct.

```
// h, k are secret bits, already initialised

hif (h==1) then Print 0 // Print 0
            else Print 1 // Print 1
fih
```

Since `Print 1` is strictly more secure than `Print (h XOR k)`, shouldn't it be the case that this program is at least as secure as Fig. 4 ?

Fig. 5. A program which emits 0's and 1's

To study this question, we first observe that the statement `Print 1` is more secure than `Print (h XOR k)`; that is because `Print 1` transmits no information at all because its observable does not depend on the value of the secret, and so is an example of the most secure program (recall (C) in Sect. 2). This means if "**`hif`** ... **`fih`**" has a semantics which is monotonic, it must be the case that Fig. 5 is more secure than Fig. 4.

In these simple cases, we can give a QIF semantics for the two programs directly, taking into account the assumption that only the "Print" statements transmit observations. Assume that the variables `h, k` are initialised uniformly and independently. In the case of Fig. 4 the adversary reasons that if 1 is printed then this is either because the lower branch was executed (`h == 0`) or the upper branch is executed (`h == 1`) and `h XOR k` evaluates to 1. When he observes 0 however that can only happen if (`h == 1`). We can summarise the relative marginal and posterior probabilities for these cases as follows. We write the values of the variables as a pair, with `h` as the left component and `k` the right component. For example the pair `(1, 0)` means that `h` has value 1 and `k` has value 0. In the scenario just described, there is a $1/4$ marginal probability that the pair is `(1, 1)`, and in this case the adversary knows the value of `h` and `k`. But with probability $3/4$ a 1 is printed, and in this case the adversary does not know which of the remaining pairs it can be. The following marginal and posterior probabilities summarise this behaviour:

```
marginal  {posterior}
1 ÷ 4       {1 ÷ 1   "(1,1)"}   ← Prints 0
3 ÷ 4       {1 ÷ 3   "(0,0)"    ← Prints 1
             1 ÷ 3   "(0,1)"
             1 ÷ 3   "(1,0)"}
```

For Fig. 5 the situation is a little simpler—when `h` is 0 a 1 is printed and vice versa. This leads to the following result.

```
marginal  {posterior}
1 ÷ 2       {1 ÷ 2   "(1,1)"    ← Prints 0
             1 ÷ 2   "(1,0)"}
1 ÷ 2       {1 ÷ 2   "(0,0)"    ← Prints 1
             1 ÷ 2   "(0,1)"}
```

Comparing these two probabilistic results we see now that "`hif ...fih`" cannot be monotonic, because if we replace `Print(h XOR k)` by the more secure `Print 1` more information (not less) is leaked. For instance, under Bayes Vulnerability for `h`, the adversary can guess the value of `h` exactly under Fig. 5 but only with probability $^3/_4$ under Fig. 4.

This experiment tells us that a hidden "`hif ...fih`" construct cannot be defined in a way that respects monotonicity of information flow, and could therefore be too risky to use.

6 Experiments with Kuifje

The experiments described above were carried out using the tool Kuifje [12] which interprets a small programming language in terms of the QIF semantics, but extended to the Hidden Markov Model alluded to in Sect. 2. It supports the usual programming constructs (assignment, sequencing, conditionals and loops) but crucially it takes into account information flows consistent with the channel model outlined in Sect. 2. In particular the "Print" statements used in our examples correspond exactly to the observations that an adversary could make during program execution. This allows a direct model for eg. known side channels that potentially expose partially computation traces during program execution.

The basic assumption built into the semantics of Kuifje is that all variables cannot be observed unless revealed fully or partially through a "Print" statement. For example `Print x` would print the value of variable `x` and so reveal it completely at that point of execution, but `Print(x>0)` would only reveal whether `x` is strictly positive or not. As usual, we also assume that the adversary knows the program code.

Kuifje is implemented in Haskell and makes extensive use of the Giry monad [10] for managing the prior, posterior and marginal probabilities in the form of "hyperdistributions". A hyperdistribution is a distribution of distributions and is based on the principle that the names of observations are redundant in terms of the analysis of information flow. Hyperdistributions therefore only summarise the posterior and marginal probabilities, because these quantities are the only ones that play a role in computing posterior vulnerabilities (recall 2). Hyperdistributions satisfy a number of useful properties relevant to QIF [15], and provide the basis for algebraic reasoning of source level code.

7 Related Work

Classical approaches to measuring insecurities in programs are based on determining a "change in uncertainty" of some "prior" value of the secret—although how to measure the uncertainty differs in each approach. For example Clark et al. [3] use Shannon entropy to estimate the number of bits being leaked; and Clarkson et al. [5] model a change in belief. Smith [18] demonstrated the importance of using measures that have some operational significance, and the idea was developed further [2] by introducing the notion of g-leakage to express such significance in a very general way. The partial order used here on programs is the same as the g-leakage order introduced by Alvim et al. [2], but it appeared also in even earlier work [14]. Its properties have been studied extensively [1].

Jacobs and Zanasi [11] use ideas based on the Giry Monad to present an abstract state transformer framework for Bayesian reasoning. Our use of hyper-distributions means that the conditioning needed for the Bayesian update has taken place in the construction of the posteriors.

8 Conclusions and Discussion

Understanding the impact of information flow is hard. The conceptual tools presented here summarise the ideas of capturing the adversary's ability to use the information released, together with a modelling language that enables the study of risks associated with information leaks in complicated algorithms and protocols. The language itself is based on the Probability Monad which has enabled its interpretation in Kuifje.

It is hoped that the ability to describe scenarios in terms of adversarial gains/losses together with Kuifje that enables detailed numerical calculation of the impact of flows will lead to a better understanding of security vulnerabilities in programs.

Acknowledgements. I thank Tom Schrijvers for having the idea of embedding these ideas in Haskell, based on Carroll Morgan's talk at IFIP WG2.1 in Vermont, and for carrying it out to produce the tool Kuifje. Together with Jeremy Gibbons all four of us wrote the first paper devoted to it [12]. (It was Jeremy who suggested the name "Kuifje", the Dutch name for TinTin—and hence his "QIF".) Carroll Morgan translated the example programs into Kuifje.

References

1. Alvim, M.S., Chatzikokolakis, K., McIver, A., Morgan, C., Palamidessi, C., Smith, G.: Additive and multiplicative notions of leakage, and their capacities. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19–22 July 2014, pp. 308–322. IEEE (2014)
2. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: Proceedings 25th IEEE Computer Security Foundations Symposium (CSF 2012), pp. 265–279, June 2012
3. Clark, D., Hunt, S., Malacaria, P.: Quantitative analysis of the leakage of confidential data. Electr. Notes Theor. Comput. Sci. **59**(3), 238–251 (2001)
4. Clark, D., Hunt, S., Malacaria, P.: Quantified interference for a while language. Electr. Notes Theor. Comput. Sci. **112**, 149–166 (2005)
5. Clarkson, M.R., Myers, A.C., Schneider, F.B.: Belief in information flow. In: 18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), Aix-en-Provence, France, 20–22 June 2005, pp. 31–45 (2005)
6. Freeman, P.R.: The secretary problem and its extensions: a review. Int. Stat. Rev. **51**, 189–206 (1983)

7. Gardner, M.: Mathematical games. Sci. Am. **202**(2), 178–179 (1960)
8. Gardner, M.: Mathematical games. Sci. Am. **202**(3), 152 (1960)
9. Gilbert, J., Mosteller, F.: Recognising the maximum of a sequence. J. Am. Stat. Assoc. **61**, 35–73 (1966)
10. Giry, M.: A categorical approach to probability theory. In: Banaschewski, B. (ed.) Categorical Aspects of Topology and Analysis. LNM, vol. 915, pp. 68–85. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0092872
11. Jacobs, B., Zanasi, F.: A predicate/state transformer semantics for Bayesian learning. Electr. Notes Theor. Comput. Sci. **325**, 185–200 (2016)
12. Morgan, C., Gibbons, J., Mciver, A., Schrijvers, T.: Quantitative information flow with monads in haskell. In: Foundations of Probabilistic Programming. CUP (2019, to appear)
13. Mardziel, P., Alvim, M.S., Hicks, M.W., Clarkson, M.R.: Quantifying information flow for dynamic secrets. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, 18–21 May 2014, pp. 540–555 (2014)
14. 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). https://doi.org/10.1007/978-3-642-14162-1_19
15. McIver, A., Morgan, C., Rabehaja, T.: Abstract hidden markov models: a monadic account of quantitative information flow. In Proceedings LiCS 2015 (2015)
16. Morgan, C.: The shadow knows: refinement of ignorance in sequential programs. Sci. Comput. Program. **74**(8), 629–653 (2009). Treats Oblivious Transfer
17. Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**(379–423), 623–656 (1948)
18. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_21