

Adversaries and Information Leaks (Tutorial)

Geoffrey Smith

School of Computing and Information Sciences, Florida International University,
Miami, FL 33199, USA
`smithg@cis.fiu.edu`

Abstract. Secure information flow analysis aims to prevent programs from leaking their H (high) inputs to their L (low) outputs. A major challenge in this area is to relax the standard noninterference properties to allow “small” leaks, while still preserving security. In this tutorial paper, we consider three instances of this theme. First, we consider a type system that enforces the usual Denning restrictions, except that it specifies that encrypting a H plaintext yields a L ciphertext. We argue that this type system ensures security, assuming strong encryption, by giving a reduction that maps a noninterference adversary (which tries to guess which of two H inputs was used, given the L outputs) to an IND-CPA adversary (which tries to guess which of two plaintexts are encrypted, given the ciphertext). Second, we explore termination leaks in probabilistic programs when typed under the Denning restrictions. Using a notion of probabilistic simulation, we show that such programs satisfy an approximate noninterference property, provided that their probability of nontermination is small. Third, we consider quantitative information flow, which aims to measure the amount of information leaked. We argue that the common information-theoretic measures in the literature are unsuitable, because these measures fail to distinguish between programs that are wildly different from the point of view of an adversary trying to guess the H input.

1 Introduction

Suppose that a program c processes some sensitive information. How do we know that c will not *leak* the information, either accidentally or maliciously? How can we ensure that c is *trustworthy*?

The approach of *secure information flow analysis* is to classify c 's variables into different security levels, such as H (high) or L (low), and to do a *static analysis*, often in the form of a *type system*, on c prior to executing it. The goal is to prove that c conforms to some specified flow policy, which can encompass both confidentiality and integrity concerns; in this paper, we will focus exclusively on confidentiality. See [1] for a survey of this area.

It is important to recognize that the secure information flow problem involves *two adversaries*: the *program* c itself, and also the *observer* \mathcal{O} of c 's public output. These two adversaries have distinct capabilities:

- The program c has *direct access* to the sensitive information (the initial values of H variables), but its behavior is *constrained* by the static analysis.
- The observer \mathcal{O} has *direct access* only to c 's public output (the final values of L variables, etc.), but its behavior is *unconstrained*, except for computational resource bounds.

The decision as to what constitutes c 's public output is quite important, of course; in particular secure information flow becomes far more difficult if we consider c 's *running time* to be a public output.

A classic approach to secure information flow in imperative programs is based on the *Denning restrictions* proposed in [2]:

- An expression is classified as H if it contains any H variables; otherwise, it is classified as L .
- To prevent *explicit flows*, a H expression cannot be assigned to a L variable.
- To prevent *implicit flows*, an **if** or **while** command whose guard is H may not make *any* assignments to L variables.

If c satisfies the Denning restrictions, then it can be proven [3] that c satisfies *noninterference*, which says (assuming that c always terminates) that the final values of the L variables are *independent* of the initial values of the H variables. Hence observer \mathcal{O} , seeing the final values of the L variables, can deduce *nothing* about the initial values of the H variables.

Unfortunately, noninterference is often too strong in practice. This leads to a major practical challenge: how can we relax noninterference to allow “small” information leaks, while still preserving security? In the next three sections, we consider three instances of this theme. In Sections 2 and 3, we consider secure information flow analyses that are permissive about leaks caused by *encryption* and *nontermination*, respectively; these sections summarize [4] and [5], where additional details can be found. In Section 4, we present some preliminary ideas about a general theory of *quantitative* information flow, which aims to measure the “amount” of information leaked.

2 Secure Information Flow for a Language with Encryption

Suppose that \mathcal{E} and \mathcal{D} denote encryption and decryption with a suitably-chosen shared key K . We allow program c to call \mathcal{E} and \mathcal{D} , but we do not give it direct access to K . Intuitively, we would like to extend the Denning restrictions with the following rules:

- If expression e is H , then $\mathcal{E}(e)$ is L .
- If expression e is either L or H , then $\mathcal{D}(e)$ is H .

But are these rules *sound*? Note that they *violate* noninterference, since $\mathcal{E}(e)$ depends on e .

In fact these rules are *unsound* if encryption is *deterministic*. For example, suppose that *secret* is a H n -bit variable and that *leak* and *mask* are L variables. Consider the following program, in which “|” denotes bitwise-or, and “ $\gg 1$ ” denotes right shift by one bit:

```

leak := 0;
mask :=  $2^{n-1}$ ;
while mask  $\neq$  0 do (
  if  $\mathcal{E}(\text{secret} | \text{mask}) = \mathcal{E}(\text{secret})$  then
    leak := leak | mask;
    mask := mask  $\gg$  1
)

```

This program is allowed under the proposed rules. But if \mathcal{E} is deterministic, then the program efficiently copies *secret* into *leak*, because then the test in the **if** command is true iff $\text{secret} | \text{mask} = \text{secret}$.

In fact it is well understood in the cryptographic community that deterministic encryption cannot give good security properties.¹ We recall the definitions of *symmetric encryption scheme* and *IND-CPA security* from [6]:

Definition 1. A symmetric encryption scheme \mathcal{SE} with security parameter k is a triple of algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$, where

- \mathcal{K} is a randomized key-generation algorithm that generates a k -bit key; we write $K \stackrel{?}{\leftarrow} \mathcal{K}$
- \mathcal{E} is a randomized encryption algorithm that takes a key and a plaintext and returns a ciphertext; we write $C \stackrel{?}{\leftarrow} \mathcal{E}_K(M)$.
- \mathcal{D} is a deterministic decryption algorithm that takes a key and a ciphertext and returns the corresponding plaintext; we write $M := \mathcal{D}_K(C)$.

We recall the notion of *IND-CPA security*, which stands for *indistinguishability under chosen-plaintext attack*. An adversary \mathcal{A} is given an *LR oracle* of the form

$$\mathcal{E}_K(\text{LR}(\cdot, \cdot, b)),$$

where K is a randomly generated key and b is an internal *selection bit*, which is either 0 or 1. When \mathcal{A} sends a pair of equal-length messages (M_0, M_1) to the LR oracle, it selects either M_0 or M_1 according to the value of b , encrypts it using \mathcal{E}_K , and returns the ciphertext C to \mathcal{A} . Thus when \mathcal{A} sends a sequence of pairs of messages to the LR oracle, it either gets back encryptions of the *left* messages (if $b = 0$) or else encryptions of the *right* messages (if $b = 1$). \mathcal{A} 's challenge is to guess which of these two “worlds” it is in.

¹ However, it is standard to *implement* strong encryption using a deterministic block cipher (modeled as a pseudo-random permutation) and random vectors, using techniques like cipher-block chaining with random initial vector [6]. Interestingly, Courant, Ene, and Lakhnech [7] have considered secure information flow in that lower implementation level, using an ingenious type system that tracks both the security level as well as the *randomness* of expressions.

Formally, \mathcal{A} is executed in two different *experiments*, depending on the choice of the selection bit b :

Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-b}}(\mathcal{A})$
 $K \stackrel{?}{\leftarrow} \mathcal{K}$;
 $d \stackrel{?}{\leftarrow} \mathcal{A}^{\mathcal{E}_K}(\text{LR}(\cdot, \cdot, b))$;
 return d

The *IND-CPA advantage* of \mathcal{A} is defined as

$$\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(\mathcal{A}) = \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-1}}(\mathcal{A}) = 1] - \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-0}}(\mathcal{A}) = 1].$$

Thus \mathcal{A} 's IND-CPA advantage is its probability of (correctly) guessing 1 in world 1, minus its probability of (wrongly) guessing 1 in world 0. Finally, \mathcal{SE} is *IND-CPA secure* if no adversary \mathcal{A} running in polynomial time in the security parameter k can achieve a non-negligible advantage. (As usual, $s(k)$ is *negligible* if for any positive polynomial $p(k)$, there exists k_0 such that $s(k) \leq \frac{1}{p(k)}$, for all $k \geq k_0$.)

Now we define the programming language that we will consider. We use a simple imperative language with the following syntax:

(expressions) $e ::= x \mid m \mid e_1 + e_2 \mid \dots \mid \mathcal{D}(e_1, e_2)$
 (commands) $c ::= x := e \mid$
 $x \stackrel{?}{\leftarrow} \mathcal{R} \mid$
 $(x, y) \stackrel{?}{\leftarrow} \mathcal{E}(e) \mid$
skip \mid
if e **then** c_1 **else** $c_2 \mid$
while e **do** $c \mid$
 $c_1; c_2$

In the syntax, metavariables x and y range over identifiers and m over integer literals. Integers are the only values; we use 0 for false and nonzero for true. The command $x \stackrel{?}{\leftarrow} \mathcal{R}$ is a random assignment; here \mathcal{R} ranges over some set of probability distributions on the integers.

The commands for encryption and decryption are slightly non-obvious. There are two issues: first, encryption cannot conceal the *length* of the plaintext; second, for IND-CPA security there must be *many* ciphertexts corresponding to a given plaintext, so ciphertexts must be *longer* than plaintexts. We deal with these issues in our language by assuming that all integer values are n bits long, for some n , and that encryption always takes an n -bit plaintext and produces a $2n$ -bit ciphertext. Thus the encryption command has the form $(x, y) \stackrel{?}{\leftarrow} \mathcal{E}(e)$; it encrypts the n -bit value of expression e , putting the first n bits of the ciphertext into x and the second n bits into y . Symmetrically, the decryption expression $\mathcal{D}(e_1, e_2)$ takes two expressions, giving $2n$ bits of ciphertext, and produces the corresponding n -bit plaintext.

As shown in [3], the Denning restrictions can be enforced using a type system. We extend such a type system with rules for the new constructs; we do not show the rules here (they can be found in [4]), but they enforce the following rules:

- $\mathcal{E}(e)$ is L , even if e is H .
- $\mathcal{D}(e_1, e_2)$ is H , even if e_1 and e_2 are L .
- \mathcal{R} (a random value) is L .

The reason for the last rule is that a random value is independent of the initial values of H variables.

We now wish to argue that our type system is sound. To do this, we introduce the idea of a *leaking adversary*. A leaking adversary \mathcal{B} has a H variable h and a L variable l , and other variables typed arbitrarily. It is run with h initialized to either 0 or 1, each with probability 1/2. It can call $\mathcal{E}()$ and $\mathcal{D}()$, and it tries to copy the initial value of h into l . Formally, \mathcal{B} is executed in the following experiment:

Experiment $\mathbf{Exp}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B})$

```

 $K \stackrel{?}{\leftarrow} \mathcal{K};$ 
 $h_0 \stackrel{?}{\leftarrow} \{0, 1\};$ 
 $h := h_0;$ 
initialize all other variables of  $\mathcal{B}$  to 0;
run  $\mathcal{B}^{\mathcal{E}\kappa(\cdot), \mathcal{D}\kappa(\cdot)}$ ;
if  $l = h_0$  then return 1 else return 0

```

Here $h_0 \stackrel{?}{\leftarrow} \{0, 1\}$ assigns a random 1-bit integer to h_0 . Variable h_0 must not occur in \mathcal{B} ; it is used to record the initial value of h . Finally, the *leaking advantage* of \mathcal{B} is defined as

$$\mathbf{Adv}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B}) = 2 \cdot \Pr[\mathbf{Exp}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B}) = 1] - 1.$$

The leaking advantage is defined in this way to reflect the fact that \mathcal{B} can trivially succeed with probability 1/2.

We argue the soundness of our type system via a *reduction*; for the moment, we drop decryption from our language:

Theorem 1. *Given a well-typed leaking adversary \mathcal{B} that does not call $\mathcal{D}()$ and that runs in polynomial time $p(k)$, there exists an IND-CPA adversary \mathcal{A} such that*

$$\mathbf{Adv}_{S\mathcal{E}}^{\text{ind-cpa}}(\mathcal{A}) \geq \frac{1}{2} \cdot \mathbf{Adv}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B}).$$

Moreover, \mathcal{A} runs in $O(p(k))$ time.

The theorem gives the following immediate corollary:

Corollary 1. *If $S\mathcal{E}$ is IND-CPA secure, then there is no polynomial-time, well-typed leaking adversary \mathcal{B} that achieves a non-negligible advantage.*

The proof of Theorem 1 is by explicit construction. Given leaking adversary \mathcal{B} , we construct IND-CPA adversary \mathcal{A} that runs \mathcal{B} with a randomly-chosen 1-bit value of h . Whenever \mathcal{B} calls $\mathcal{E}(e)$, \mathcal{A} passes $(0^n, e)$ to its oracle $\mathcal{E}_K(LR(\cdot, \cdot, b))$ and returns the result to \mathcal{B} . If \mathcal{B} terminates within $p(k)$ steps and succeeds in leaking h to l , then \mathcal{A} guesses 1; otherwise \mathcal{A} guesses 0.

To understand this construction, the first thing to notice is that if the selection bit b is 1, then \mathcal{B} is run *faithfully*—whenever \mathcal{B} calls $\mathcal{E}(e)$, it correctly receives in reply $\mathcal{E}_K(e)$. But if the selection bit b is 0, then \mathcal{B} is run *unfaithfully*—now whenever \mathcal{B} calls $\mathcal{E}(e)$, it receives in reply $\mathcal{E}_K(0^n)$, which is a *random value* that has nothing to do with e .

What is \mathcal{A} 's IND-CPA advantage? When the selection bit b is 1, then \mathcal{B} is run faithfully and hence

$$\Pr[\mathbf{Exp}_{S\mathcal{E}}^{\text{ind-cpa-1}}(\mathcal{A}) = 1] = \Pr[\mathbf{Exp}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B}) = 1] = \frac{1}{2} \cdot \mathbf{Adv}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B}) + \frac{1}{2}.$$

When the selection bit b is 0, then \mathcal{B} is run as a well-typed program in a language with random assignment but no encryption—in other words, \mathcal{B} no longer can take advantage of the leak associated with the typing rule for encryption. Hence we would expect that standard noninterference results will prevent \mathcal{B} from copying h to l with probability better than $1/2$. However, there is a subtlety here—when \mathcal{B} is run unfaithfully, it might fail to *terminate*. (For example, $\mathcal{E}(h)$ and $\mathcal{E}(h+1)$ are always distinct if \mathcal{B} is run faithfully, but they have a small probability of being equal if \mathcal{B} is run unfaithfully.) To deal with this possibility, we need a careful analysis of the behavior of well-typed probabilistic programs that might not terminate. Such an analysis is described in Section 3 of this paper; it allows us to show that

$$\Pr[\mathbf{Exp}_{S\mathcal{E}}^{\text{ind-cpa-0}}(\mathcal{A}) \leq \frac{1}{2}]$$

as expected. (Details are given in [4].) In conclusion we get

$$\begin{aligned} \mathbf{Adv}_{S\mathcal{E}}^{\text{ind-cpa}}(\mathcal{A}) &= \Pr[\mathbf{Exp}_{S\mathcal{E}}^{\text{ind-cpa-1}}(\mathcal{A}) = 1] - \Pr[\mathbf{Exp}_{S\mathcal{E}}^{\text{ind-cpa-0}}(\mathcal{A}) = 1] \\ &\geq \frac{1}{2} \cdot \mathbf{Adv}_{S\mathcal{E}}^{\text{leak}}(\mathcal{B}) \end{aligned}$$

as claimed.

We have shown that our type system rules out well-typed, efficient leaking adversaries. But can we get a result more like noninterference? To this end, let c be a well-typed, polynomial-time program in our language, and let μ and ν be memories that agree on L variables. Suppose we run c under either μ or ν , each with probability $1/2$, and show the final values of the L variables of c to observer \mathcal{O} , which we here refer to as a *noninterference adversary*. Could \mathcal{O} guess which initial memory was used?

More formally, a noninterference adversary \mathcal{O} for c , μ , and ν is a program that refers only to the L variables of c and outputs its guess into a new variable g . \mathcal{O} is run in the following experiment, where h_0 is a new variable:

Experiment $\mathbf{Exp}_{\mathcal{SE},c,\mu,\nu}^{\text{NI}}(\mathcal{O})$
 $K \stackrel{?}{\leftarrow} \mathcal{K};$
 $h_0 \stackrel{?}{\leftarrow} \{0, 1\};$
if $h_0 = 0$ **then** initialize the variables of c to μ
 else initialize the variables of c to ν ;
 $c;$
 $\mathcal{O};$
if $g = h_0$ **then return** 1 **else return** 0

The *noninterference advantage* of \mathcal{O} is defined as

$$\mathbf{Adv}_{\mathcal{SE},c,\mu,\nu}^{\text{NI}}(\mathcal{O}) = 2 \cdot \Pr[\mathbf{Exp}_{\mathcal{SE},c,\mu,\nu}^{\text{NI}}(\mathcal{O}) = 1] - 1.$$

Now we have the following theorem:

Theorem 2. *If c is a well-typed, polynomial-time program and μ and ν are memories that agree on L variables, then no polynomial-time noninterference adversary \mathcal{O} for c , μ , and ν can achieve a non-negligible noninterference advantage.*

As in Theorem 1, the proof is by explicit construction. Given noninterference adversary \mathcal{O} , we can construct a well-typed leaking adversary \mathcal{B} . An interesting point here is that \mathcal{O} *cannot* be assumed to be well typed. But because \mathcal{O} sees only the L variables of c , we can give all its variables type L , making \mathcal{O} *automatically* well typed under our typing rules. Hence we can use \mathcal{O} in constructing \mathcal{B} :

Adversary \mathcal{B}
 initialize the L variables of c according to μ and ν ;
if $h = 0$ **then**
 initialize the H variables of c according to μ
else
 initialize the H variables of c according to ν ;
 $c;$
 $\mathcal{O};$
 $l := g$

It is easy to see that \mathcal{B} is well typed and that its leaking advantage is the same as \mathcal{O} 's noninterference advantage.

Thus we have shown that, on polynomial-time programs c , our type system ensures a property that is essentially as good as noninterference—a polynomial-time observer \mathcal{O} is basically unable to determine *anything* about the initial values of the H variables from the final values of the L variables.

Finally, we remark that Theorem 1 can be generalized to the full language including decryption. A similar reduction can be done, except that from leaking adversary \mathcal{B} we now construct an *IND-CCA adversary* \mathcal{A} [6], which has a decryption oracle $\mathcal{D}_K(\cdot)$ in addition to its LR-oracle $\mathcal{E}_K(\text{LR}(\cdot, \cdot, b))$. It is not clear to us whether this is strictly necessary—see [4] for more discussion.

We conclude this section by mentioning some related work. Peeter Laud has pioneered the area of secure information flow analysis in the presence of encryption; his works include [8,9,10]. The third of these papers treats a richer language with primitives for generating and manipulating keys directly, though not handling decryption explicitly, and necessitating more complex typing rules and proofs. Other recent work in this area includes [11,12,13,14]; a major goal of these works is to “scale up” the language to the point that practical applications can be built.

More distantly related is the large body of recent work aimed at proving computational security properties of cryptographic protocols; examples include the work of Backes and Pfitzmann [15] and Laud [16]. Such work has a quite different adversary model than what is used in secure information flow analysis—the focus is on distributed systems in the presence of an *active adversary* which can insert and modify messages without constraint by a type system, but which does not have direct access to secrets.² Also type systems for cryptographic protocols seem to offer less support for general-purpose programming—for example, the type system in [16] does not allow branching on secret data.

3 Termination Leaks in Probabilistic Programs

In Section 2, we assumed that all adversaries run in time polynomial in k , the key size. This might seem to be “without loss of generality” (practically speaking) since otherwise the adversary takes too long. But what if program c either terminates quickly or else goes into an infinite loop? In that case, observer \mathcal{O} might quickly be able to observe whether c terminates.

Furthermore, the Denning restrictions allow H variables to appear in guards of loops, because disallowing them would seem too restrictive in practice. This means that H variables can affect termination, as in examples like

```
while secret = 0 do
  skip;
leak := 1
```

It is for this reason that the noninterference property discussed in Section 1 includes the assumption that program c *always terminates*.

In this section, we try to *quantify* such termination leaks. The setting we consider is probabilistic programs with random assignment, but no encryption or decryption. We use the same type system as in Section 2, except that we no longer need typing rules for encryption and decryption; thus we simply enforce the Denning restrictions, extended with a rule that says that random values are L . Semantically, our programs are modeled as *Markov chains* [17] of configurations (c, μ) , where c is the command remaining to be executed and μ is the memory.

² However, a similar active adversary is considered in some recent work in secure information flow, such as [14], that addresses *integrity* in addition to confidentiality.

In this setting, perfect security is given by *probabilistic noninterference*, which says that the final *probability distribution* on L variables is independent of the initial values of the H variables.

Here is an example of a program that violates probabilistic noninterference:

```

 $t \stackrel{?}{\leftarrow} \{0, 1\};$ 
if  $t = 0$  then (
  while  $h = 1$  do skip;
   $l := 0$ 
)
else (
  while  $h = 0$  do skip;
   $l := 1$ 
)

```

Note that $t \stackrel{?}{\leftarrow} \{0, 1\}$ is a *random assignment* that assigns either 0 or 1 to t , each with probability $1/2$. Assuming that h is H and t and l are L , this program satisfies the extended Denning restrictions. But if $h = 0$, it terminates with $l = 0$ with probability $1/2$ and loops with probability $1/2$. And if $h = 1$, then it terminates with $l = 1$ with probability $1/2$ and loops with probability $1/2$.

However we can argue an *approximate probabilistic noninterference* property:

Theorem 3. *If c satisfies the extended Denning restrictions and loops with probability at most p , then c 's deviation from probabilistic noninterference is at most $2p$.*

In our example program, $p = 1/2$, and the deviation is $|1/2 - 0| + |0 - 1/2| = 1 = 2p$, achieving the bound specified by the theorem. (The first term compares the probability that $l = 0$ after $h = 0$ and after $h = 1$; the second compares the probability that $l = 1$ after $h = 0$ and after $h = 1$.)

To prove this theorem, we introduce the idea of a *stripped program*, denoted by $\lfloor c \rfloor$. We form $\lfloor c \rfloor$ from c by stripping out all subcommands that do not assign to L variables, replacing them with **skip**. (In terms of the type system, this is equivalent to stripping out all subcommands of type H cmd.) For example, the stripped version of our example program is the following:

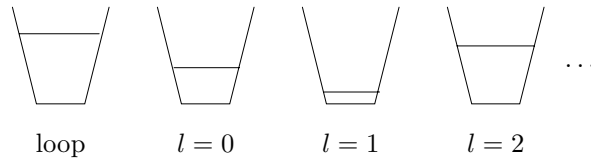
```

 $t \stackrel{?}{\leftarrow} \{0, 1\};$ 
if  $t = 0$  then (
  skip;
   $l := 0$ 
)
else (
  skip;
   $l := 1$ 
)

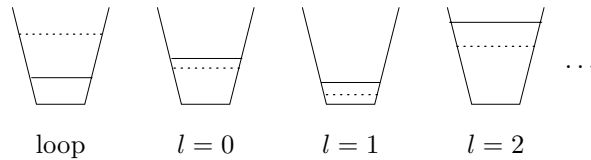
```

It turns out that if c satisfies the extended Denning restrictions, then $\lfloor c \rfloor$ contains no H variables. More interestingly, in this case c satisfies what we call

the *Bucket Property*, which relates to behavior of c and $\lfloor c \rfloor$. To visualize this property, imagine that the result of running c is shown as a sequence of buckets, one for each possible final value of c 's L variables; also, we have a bucket to represent looping. The probability of each outcome is indicated by the amount of water in each bucket. Suppose that c 's buckets look like this:



Then the Bucket Property says that $\lfloor c \rfloor$'s buckets are gotten simply by pouring some of the water from c 's loop bucket into some of the other buckets:



In other words, as we pass from c to $\lfloor c \rfloor$, the probabilities of L outcomes can only increase or stay the same; they cannot decrease.

In prior work on secure information flow, *probabilistic bisimulation* has often been a useful proof technique (see, for example, [18]). But in proving the Bucket Property, we use a non-symmetric *probabilistic simulation* [19] instead. Specifically, we define a *fast simulation*, which is a modification of the *weak simulation* considered by Baier, Katoen, Hermanns, and Wolf [20].

We develop the theory of fast simulation in the abstract setting of Markov chains. Intuitively, state t simulates state s if t can simulate whatever s can do. Thus if s can go to some state s' with probability p , then t should be able to match this by going to one or more states t', t'', t''', \dots , each of which simulates s' , with total probability at least p . However we must not “double count” t 's probabilities—for example, if s goes to s' with probability $1/3$ and t goes to t' with probability $1/2$, then if we use t' to simulate the move to s' we must remember that $1/3$ of t' 's probability is “used up”, leaving just $1/6$ to be used in simulating other moves of s . These considerations lead to what is called a *weight function* Δ that specifies how the probabilities are matched up. A further consideration is that s might go to a state that is *already* simulated by t —in this case s has made an “insignificant” move, which t should not need to match. Thus in general we partition the states reachable in one step from s into two sets, U and V , corresponding to the “significant” and “insignificant” moves, respectively.

Formally, given a (discrete-time) Markov chain with state set S and transition probabilities \mathbf{P} , we define:

Definition 2. Let R be a binary relation on S . R is a fast simulation if, whenever sRt , the states reachable in one step from s can be partitioned into two sets U and V such that

1. vRt for every $v \in V$, and
2. letting $K = \sum_{u \in U} \mathbf{P}(s, u)$, if $K > 0$ then there exists $\Delta : S \times S \rightarrow [0, 1]$ such that
 - (a) $\Delta(u, w) > 0$ implies that uRw ,
 - (b) $\mathbf{P}(s, u)/K = \sum_{w \in S} \Delta(u, w)$ for all $u \in U$, and $\mathbf{P}(t, w) = \sum_{u \in U} \Delta(u, w)$ for all $w \in S$.

We now describe the key theory associated with fast simulation. First, given binary relation R , we say that a set T of states is *upwards closed* if $s \in T$ and sRt implies that $t \in T$. Next, given state s , natural number n , and set T of states, let us write $\Pr(s, n, T)$ to denote the probability of reaching a state in T from s in at most n steps.

Now we have the key theorem, which says that if t fast simulates s , then t reaches any upwards closed set T with at least as great probability and at least as quickly as s does:

Theorem 4. If R is a fast simulation, T is upwards closed, and sRt , then $\Pr(s, n, T) \leq \Pr(t, n, T)$ for every n .

We remark that the *universal relation* $R_U = S \times S$ is trivially a fast simulation. But under R_U the only upwards closed sets are \emptyset and S itself, which means that Theorem 4 is uninteresting in that case.

We now apply the theory of fast simulation to the setting of probabilistic programs that satisfy the extended Denning restrictions. The key result is that we can define a fast simulation R_L such that $(c, \mu)R_L(\lfloor c \rfloor, \mu)$, for any well-typed command c .

Definition 3. If c and d are well-typed commands, then we say that cR_Ld if this can be proved from the following six rules:

1. $c_1R_L\mathbf{skip}$, if c_1 does not assign to L variables.
2. $(x := e)R_L(x := e)$.
3. $(x \stackrel{?}{\leftarrow} \mathcal{D})R_L(x \stackrel{?}{\leftarrow} \mathcal{D})$.
4. $(\mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2)R_L(\mathbf{if } e \mathbf{ then } d_1 \mathbf{ else } d_2)$, if $e : L$, $c_1R_Ld_1$, and $c_2R_Ld_2$.
5. $(\mathbf{while } e \mathbf{ do } c_1)R_L(\mathbf{while } e \mathbf{ do } d_1)$, if $e : L$ and $c_1R_Ld_1$.
6. $(c_1; c_2)R_L(d_1; d_2)$, if $c_1R_Ld_1$ and $c_2R_Ld_2$.

We extend R_L to configurations with the following two rules:

1. $\mu R_L \nu$, if μ and ν agree on L variables.
2. $(c, \mu)R_L(d, \nu)$, if cR_Ld and μ and ν agree on L variables.

It can be proved that R_L is a fast simulation, and that for any well-typed c , $cR_L\lfloor c \rfloor$. This implies the Bucket Property. For given some L outcome (such as

$l = 17$), let T be the set of memories that satisfy that outcome (for example, $T = \{\nu \mid \nu(l) = 17\}$). Since T is upwards closed under R_L , we can apply Theorem 4 to deduce that $\Pr((c, \mu), n, T) \leq \Pr(\lfloor c \rfloor, \mu, n, T)$, for every n . Finally, we can extend to the probability of *eventually* terminating in T , since this is just $\lim_{n \rightarrow \infty} \Pr((c, \mu), n, T)$.

Given the Bucket Property, we can now prove the approximate probabilistic noninterference property. Recall that $\lfloor c \rfloor$ contains no H variables. Hence if memories μ and ν agree on L variables, then the behavior of $(\lfloor c \rfloor, \mu)$ must be *identical* to that of $(\lfloor c \rfloor, \nu)$. Thus we can build a “bridge” between (c, μ) and (c, ν) :

$$(c, \mu) \xleftrightarrow{\text{Bucket Prop}} (\lfloor c \rfloor, \mu) \equiv (\lfloor c \rfloor, \nu) \xleftrightarrow{\text{Bucket Prop}} (c, \nu)$$

Since (c, μ) 's loop bucket contains at most p units of water, the sum of the absolute value of the differences between the L outcome buckets of (c, μ) and of $(\lfloor c \rfloor, \mu)$ is at most p . Similarly for (c, ν) . Hence the sum of the absolute value of the differences between the L outcome buckets of (c, μ) and of (c, ν) is at most $2p$.

We conclude this section by remarking that observer \mathcal{O} , given the final values of c 's L variables, could try to distinguish between initial memories μ and ν through statistical hypothesis testing. Assuming that the probability p of non-termination is small, then the approximate noninterference property gives us a way to bound \mathcal{O} 's ability to do this, as in the work of Di Pierro, Hankin, and Wiklicky [21]. Finally, we remark that the Bucket Property is used crucially in the proof of Theorem 1 in Section 2 of this paper, to bound the advantage of leaking adversary \mathcal{B} when run unfaithfully.

4 Foundations for Quantitative Information Flow

In the two previous sections, we considered information leaks from H to L associated with encryption and with nontermination, showing that secure information flow analyses can be permissive about such flows, while still preserving security guarantees. More generally, it would be valuable to develop a theory of “small” information leaks that is independent of any particular programming mechanism. To this end, in this section we consider the foundations of a *quantitative* theory of information flows. Such a quantitative theory has long been recognized as an important generalization of the theory of noninterference, and there has been quite a lot of recent work in this area, including the works of Clark, Hunt, and Malacaria [22,23,24], Köpf and Basin [25], Clarkson, Myers, and Schneider [26], Lowe [27], and Di Pierro, Hankin, and Wiklicky [21]. Also related is work in quantitative *anonymity*, such as that of Chatzikokolakis, Palamidessi, and Panangaden [28].

We can identify four main research steps required to develop a useful theory of quantitative information flow:

1. Define a quantitative notion of information flow.
2. Show that the notion gives appropriate security guarantees.

3. Devise static analyses to enforce a given quantitative flow policy.
4. Prove the soundness of the analyses.

In this paper, we limit our discussion to Steps 1 and 2.

Moreover, rather than trying to tackle the problem in full generality, we will consider important *special cases* in the hopes of better understanding what is going on. We therefore adopt the following conceptual framework:

- We assume that there is a single secret h , which is chosen from some space S according to some a priori, *publicly-known* probability distribution.
- We assume that c is a program that has *only* h as input and (maybe) leaks information about h to its unique public output l .
- We assume that c is *deterministic* and *total*.

Having made these assumptions, we can now follow Köpf and Basin [25] and observe that the public output l is a *function* of the secret h ; thus there exists a function f such that $l = f(h)$. Furthermore, f induces an *equivalence relation* \sim on S :

$$h_1 \sim h_2 \text{ iff } f(h_1) = f(h_2).$$

(In set theory, \sim is called the *kernel* of f .) Hence the program c *partitions* S into the equivalence classes of \sim .

So what information is leaked by c ? The observer \mathcal{O} sees the final value of l . This tells \mathcal{O} which equivalence class h belonged to. How bad is that? We can first explore that question by considering two extreme situations:

Extreme 1. If f is a *constant* function, then there is just one equivalence class, and noninterference holds.

Extreme 2. If f is *one-to-one*, then the equivalence classes are singletons, and we have *total leakage* of h (in principle).

The reason that we say “in principle” in Extreme 2 is that \mathcal{O} might be unable to *compute* the value of h efficiently from the value of l ; our framework here is thus *information-theoretic* rather than *computational*.

To assess situations between the two extremes considered above, we need appropriate quantitative measures. Here we review two common information-theoretic measures. Let X be a discrete random variable whose values have probabilities $p_1, p_2, p_3, \dots, p_n$, where we assume for convenience that $p_i \geq p_{i+1}$, for all i . The *Shannon entropy* of X is defined by

$$H(X) = \sum_{i=1}^n p_i \log \frac{1}{p_i}.$$

The Shannon entropy can be viewed informally as the “uncertainty” about X ; more precisely it can be understood as the expected number of bits required to transmit X optimally. The *Guessing entropy* of X is defined by

$$G(X) = \sum_{i=1}^n ip_i.$$

The Guessing entropy can be understood as the expected number of guesses required to guess X optimally.

Let us now apply Shannon entropy to the partitions induced by program c . For simplicity, let us consider the case where h is *uniformly distributed* over space S , and $|S| = n$. Suppose that the partition induced by c consists of r equivalence classes C_1, C_2, \dots, C_r , where $|C_i| = n_i$, for all i . Then the Shannon entropy of h is

$$H(h) = \sum_{i=1}^r \frac{1}{n} \log n = \log n.$$

This can be viewed as the “initial uncertainty about h ”. And the Shannon entropy of l is

$$H(l) = \sum_{i=1}^r \frac{n_i}{n} \log \frac{n}{n_i}.$$

Plausibly, this can be viewed as the “amount of information leaked”. This view is supported by the two extreme cases discussed above. In Extreme 1, there is just one equivalence class, of size n , so

$$H(l) = \sum_{i=1}^1 \frac{n}{n} \log \frac{n}{n} = 0$$

and in Extreme 2, there are n equivalence classes, each of size 1, so

$$H(l) = \sum_{i=1}^n \frac{1}{n} \log n = \log n.$$

We can also ask another question, which is more crucial from the point of view of security: how much uncertainty about h remains *after* the attack? This quantity can be calculated as a *conditional Shannon entropy*:

$$H(h|l) = \sum_{i=1}^r \frac{n_i}{n} H(C_i) = \sum_{i=1}^r \frac{n_i}{n} \log n_i.$$

Quite reasonably, in Extreme 1 we get $H(h|l) = \log n$ and in Extreme 2 we get $H(h|l) = 0$. Finally, there is a pretty equation relating these three quantities:

$$H(h) = H(l) + H(h|l)$$

which can be read as

“initial uncertainty = information leaked + remaining uncertainty”.

So is Step 1 (“Define a quantitative notion of information flow”) finished? In the restricted framework that we are considering, it certainly seems promising to *define* the amount of information leaked to be $H(l)$, and the remaining uncertainty to be $H(h|l)$. And in fact this seems to be the literature consensus: it

is the approach taken by Clarke, Hunt, and Malacaria [22,23,24] and by Köpf and Basin [25] (although they also consider $G(l)$ and $G(h|l)$). The approach of Clarkson, Myers, and Schneider [26] is more general, because they consider the case when the observer \mathcal{O} has (possibly mistaken) *beliefs* about the probability distribution of h . But in the special case when \mathcal{O} 's beliefs match the a priori distribution, and when the expected flow over all experiments is considered [26, Section 4.4], their approach then reduces to using $H(l)$ and $H(h|l)$.

So we might turn our attention next to Step 2 (“Show that the notion gives appropriate security guarantees”). A first step that can be taken here is to show that $H(l)$ (“the amount of information leaked”) is 0 iff c satisfies noninterference. This is good, of course, but it is just a first step—it establishes only that the zero/nonzero distinction is meaningful. A more interesting result is the *Fano Inequality*, which gives lower bounds, in terms of $H(h|l)$, on the probability that observer \mathcal{O} will *fail* to guess the value of h correctly, given l . Unfortunately these bounds are extremely weak in many cases.

Really the key question for Step 2 is whether the value of $H(h|l)$ (“the remaining uncertainty”) accurately reflects the threat to h . Let us consider some example attacks to answer this question.

First consider a program c that simply copies 1/10 of the bits of h into l ; this could be done with a program like this:

```
l = h & 0177777;
```

Assuming as before that h is uniformly distributed over S , where $|S| = n$, this attack partitions S into $2^{0.1 \log n} = n^{0.1}$ equivalence classes, each of size $2^{0.9 \log n} = n^{0.9}$. Hence we get $H(l) = 0.1 \log n$ and $H(h|l) = 0.9 \log n$, which seems quite reasonable since 9/10 of the bits of h are completely unknown to \mathcal{O} after the attack.

But now suppose that the possible values of h range from 0 to $n - 1$ and consider the following program:

```
if (h < n/10)
  l = h;
else
  l = -1;
```

This program puts 90% of the possible values of h into one big equivalence class, and puts each of the remaining 10% into *singleton* classes. Hence we get

$$H(l) = 0.9 \log \frac{1}{0.9} + 0.1 \log n \approx 0.1 \log n + 0.14$$

and

$$H(h|l) = 0.9 \log(0.9n) \approx 0.9 \log n - 0.14$$

These quantities are essentially identical to those for the previous attack! But now observer \mathcal{O} can guess h with probability 1/10.

The conclusion is that if $H(h|l)$ is used as the measure of remaining uncertainty, then Step 2 cannot be done well, because $H(h|l)$ fails to distinguish

between two attacks that are completely different from the point of view of their threat to the secrecy of h .

We might now revisit Step 1, in the hopes of finding a measure that works out better with respect to Step 2. But why not use Step 2 to guide Step 1? Why not define a measure of remaining uncertainty directly in terms of the desired security guarantees? Here is a very simple and basic measure that we can consider: let us define $V(h|l)$, the *vulnerability of h given l* , to be the probability that observer \mathcal{O} can guess h correctly in one try, given l .

Let us now explore the value of $V(h|l)$ in the case when h is uniformly distributed, with n possible values. If the partition induced by c consists of r equivalence classes C_1, C_2, \dots, C_r , where $|C_i| = n_i$ for all i , then the probability of ending in class C_i is n_i/n and the probability that \mathcal{O} can guess h in one try, given that h is in C_i , is $1/n_i$. Remarkably, the n_i 's cancel out and we get

$$V(h|l) = \sum_{i=1}^r \frac{n_i}{n} \frac{1}{n_i} = \frac{r}{n}.$$

So in this case all that matters is the *number* of equivalence classes, not their sizes!

Let us now consider some examples to assess the reasonableness of $V(h|l)$:

- a. Noninterference case: $r = 1$, $V(h|l) = 1/n$
- b. Total leakage case: $r = n$, $V(h|l) = 1$
- c. Copy 1/10 of bits: $r = n^{0.1}$, $V(h|l) = 1/n^{0.9}$
- d. Put 1/10 of h 's values into singleton classes: $r = 1 + n/10$, $V(h|l) \approx 1/10$
- e. Put h 's values into classes, each of size 10: $r = n/10$, $V(h|l) = 1/10$
- f. A password checker, that tests whether h is equal to some particular value:
 $r = 2$, $V(h|l) = 2/n$

All of these values seem reasonable, suggesting that maybe maybe $V(h|l)$ is a better foundation for quantitative information flow.

However it is clear that using a *single number* to represent a complex partition is necessarily crude. Compare examples **d** and **e**, which both have $V(h|l) \approx 1/10$. In example **d**, 1/10 of the time \mathcal{O} will *know* the value of h , since it ends up in a singleton class, and 9/10 of the time \mathcal{O} will have no idea about the value of h , since it ends up in the big equivalence class. In contrast, in example **e** we find that \mathcal{O} never *knows* the exact value of h , but always knows it to within 10 possible values. Hence giving \mathcal{O} a *second* guess would be essentially useless in example **d**, but would double \mathcal{O} 's chance of success in example **e**. Nevertheless, it seems that $V(h|l) \approx 1/10$ is a reasonable (though crude) measure of the threat to the secrecy of h in both of these examples.

We conclude by remarking that $V(h|l)$ is unfortunately not so good with respect to *compositionality*. This will be important when Steps 3 and 4 are considered—ideally, a static analysis should determine the threat associated with a sequential composition $c_1; c_2$ from the threats associated with c_1 and with c_2 . But this does not seem possible for $V(h|l)$. Another challenging question is whether the information-theoretic approach of this section could somehow

be integrated with the computational complexity approach of Section 2. These remain topics for future study.

Acknowledgments

I am grateful to Rafael Alpízar and Ziyuan Meng for helpful discussions of this work. This work was partially supported by the National Science Foundation under grant HRD-0317692.

References

1. Sabelfeld, A., Myers, A.C.: Language-based information flow security. *IEEE Journal on Selected Areas in Communications* 21(1), 5–19 (2003)
2. Denning, D., Denning, P.: Certification of programs for secure information flow. *Communications of the ACM* 20(7), 504–513 (1977)
3. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. *Journal of Computer Security* 4(2,3), 167–187 (1996)
4. Smith, G., Alpízar, R.: Secure information flow with random assignment and encryption. In: *Proc. 4th ACM Workshop on Formal Methods in Security Engineering*, Fairfax, Virginia, pp. 33–43 (November 2006)
5. Smith, G., Alpízar, R.: Fast probabilistic simulation, nontermination, and secure information flow. In: *Proc. 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, pp. 67–71 (June 2007)
6. Bellare, M., Rogaway, P.: Introduction to modern cryptography (2005), <http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html>
7. Courant, J., Ene, C., Lakhnech, Y.: Computationally sound typing for non-interference: The case of deterministic encryption. In: Arvind, V., Prasad, S. (eds.) *FSTTCS 2007*. LNCS, vol. 4855, pp. 364–375. Springer, Heidelberg (2007)
8. Laud, P.: Semantics and program analysis of computationally secure information flow. In: *Proceedings 10th ESOP (European Symposium on Programming)*, pp. 77–91 (2001)
9. Laud, P.: Handling encryption in an analysis for secure information flow. In: *Proceedings 12th ESOP (European Symposium on Programming)*, pp. 159–173 (2003)
10. Laud, P., Vene, V.: A type system for computationally secure information flow. In: Liškiewicz, M., Reischuk, R. (eds.) *FCT 2005*. LNCS, vol. 3623, pp. 365–377. Springer, Heidelberg (2005)
11. Askarov, A., Hedin, D., Sabelfeld, A.: Cryptographically-masked flows. In: *Proceedings of the 13th International Static Analysis Symposium*, Seoul, Korea, pp. 353–369 (2006)
12. Laud, P.: On the computational soundness of cryptographically masked flows. In: *Proceedings 35th Symposium on Principles of Programming Languages*, San Francisco, California (January 2008)
13. Vaughan, J., Zdancewic, S.: A cryptographic decentralized label model. In: *IEEE Symposium on Security and Privacy*, Oakland, California, pp. 192–206 (2007)
14. Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: *Proceedings 35th Symposium on Principles of Programming Languages*, San Francisco, California (January 2008)

15. Backes, M., Pfitzmann, B.: Relating symbolic and cryptographic secrecy. In: Proceeding 26th IEEE Symposium on Security and Privacy, Oakland, California (2005)
16. Laud, P.: Secrecy types for a simulatable cryptographic library. In: Proceedings 12th CCS (ACM Conference on Computer and Communications Security), pp. 26–35 (2005)
17. Feller, W.: An Introduction to Probability Theory and Its Applications, 3rd edn., vol. I. John Wiley & Sons, Inc., Chichester (1968)
18. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings 13th IEEE Computer Security Foundations Workshop, Cambridge, UK, pp. 200–214 (July 2000)
19. Jonsson, B., Larsen, K.: Specification and refinement of probabilistic processes. In: Proc. 6th IEEE Symposium on Logic in Computer Science, pp. 266–277 (1991)
20. Baier, C., Katoen, J.-P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for Markov chains. *Information and Computation* 200(2), 149–214 (2005)
21. Di Pierro, A., Hankin, C., Wiklicky, H.: Approximate non-interference. In: Proceedings 15th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada, pp. 1–17 (June 2002)
22. Clark, D., Hunt, S., Malacaria, P.: Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science* 59(3) (2002)
23. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation* 18(2), 181–199 (2005)
24. Malacaria, P.: Assessing security threats of looping constructs. In: Proceedings 34th Symposium on Principles of Programming Languages, Nice, France, pp. 225–235 (January 2007)
25. Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: Proceedings 14th ACM Conference on Computer and Communications Security, Alexandria, Virginia (2007)
26. Clarkson, M., Myers, A., Schneider, F.: Belief in information flow. In: Proceedings 18th IEEE Computer Security Foundations Workshop, Aix-en-Provence, France, pp. 31–45 (June 2005)
27. Lowe, G.: Quantifying information flow. In: Proceedings 15th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada, pp. 18–31 (June 2002)
28. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. *Information and Computation* (to appear, 2008)