Computing and Estimating Information Leakage with a Quantitative Point-to-Point Information Flow Model

by

Christopher Novakovic

A thesis submitted to the
University of Birmingham
for the degree of
Doctor of Philosophy

School of Computer Science
College of Engineering & Physical Sciences
University of Birmingham
October 2014

Abstract

Information leakage occurs when a system exposes its secret information to an unauthorised entity. Information flow analysis is concerned with tracking flows of information through systems to determine whether they process information securely or leak information.

We present a novel information flow model that permits an arbitrary amount of secret and publicly-observable information to occur at any point and in any order in a system. This is an improvement over previous models, which generally assume that systems process a single piece of secret information present before execution and produce a single piece of publicly-observable information upon termination. Our model precisely quantifies the information leakage from secret to publicly-observable values at user-defined points — hence, a "point-to-point" model — using the information-theoretic measures of mutual information and min-entropy leakage; it is ideal for analysing systems of low to moderate complexity.

We also present a relaxed version of our information flow model that estimates, rather than computes, the measures of mutual information and min-entropy leakage via sampling of a system. We use statistical techniques to bound the accuracy of the estimates this model provides. We demonstrate how our relaxed model is more suitable for analysing complex systems by implementing it in a quantitative information flow analysis tool for Java programs.

For my parents, Sue and Stevo,
without whose unwavering support
none of this would have been possible.

Acknowledgements

It's often said that a Ph.D. is a solitary endeavour. That certainly wasn't my experience: a great number of people made it enjoyable for me, and my only regret is that I can't thank them all in this limited space.

I owe an enormous debt of gratitude to Tom Chothia, my M.Sc. project and Ph.D. supervisor. For almost five years I've been fortunate to work closely with someone with such expertise, enthusiasm, and — perhaps most importantly — seemingly boundless patience. Thanks, Tom; I look forward to working with you again in the near future.

This thesis is the culmination of work performed not only with Tom, but also with my erstwhile co-authors Yusuke Kawamoto and Dave Parker; I'd like to thank them for their many valuable contributions. I'm also grateful to Eike Ritter and Marco Cova, the other members of my thesis group, for reviewing my progress every nine months and offering improvements and suggestions for future work. The Security & Privacy Group in the School of Computer Science at the University of Birmingham have attended many seminars based on this work and offered excellent feedback on the work itself and my presentation of it, both of which I appreciate very much.

The past and present occupants of Room 117 in the School of Computer Science have been the best officemates I could have hoped for; thanks to Sarah Al-Azzani, Olaf Klinke, Kat Samperi, B. J. Woolford-Lim, Olle Fredriksson, Mohammed Al Wanain, Mark Rowan, Ahmed Al-Ajeli, Abdessalam Elhabbash and Xiaodong Jia for four years' worth of lively and entertaining conversations, some of which were even about research. The School more generally has been an excellent place in which to study and begin my academic career. I'll never forget my time here.

Special mentions go to Ian Batten for being a valuable friend, and to Amber Wright for plying me with tea throughout the 2011–13 period.

Contents

III Estimating Information Leakage from Programs

IV Closing Statements

1

Introduction

What do the following events have in common?

(a) In June 2014, the American telecommunications company AT&T was legally obliged to reveal that the names, Social Security numbers, dates of birth and call activity of an undisclosed number of their customers had been accessed illicitly two months previously. AT&T claimed that the details had been exposed to three employees of a third-party contractor who were using an internal system for obtaining unlock codes for mobile devices connected to AT&T's network. The company claimed that the intrusion was intentional (Forbes, 2014).

(b) In March 2012, an individual breached a web server operated by the British Pregnancy Advisory Service (BPAS). The charity's web site was defaced, but BPAS stressed that the personal information of those who had contacted the charity seeking advice on contraception, pregnancy and abortion was not at risk during the attack. Two years later, in March 2014, it transpired that the names, telephone numbers and dates of birth of thousands of people who had telephoned the charity were retrieved by the individual, who threatened to publish the names before being arrested days later. BPAS claimed that it was unaware that a vulnerability in its web site code made it possible for the individual to recover the information surreptitiously (Information Commissioner's Office, 2014).

(c) In April 2011, an attacker gained unauthorised access to a web application server responsible for serving content to users of Sony's PlayStation Network (PSN). The attacker was able to retrieve the names, addresses, dates of birth, telephone numbers and potentially credit card information of subscribers to the service from a PSN database server, bypassing two firewalls in the process. A week later, Sony publicly confirmed the occurrence of the data breach. With the personal details of 77 million subscribers affected by the intrusion, it ranks as one of the largest data breaches in history (CNET, 2011).

The connection between these incidents is that they all feature significant *information leaks*.

1.1 What is Information Leakage?

Technology is ubiquitous in society; computers are consequently expected to store and process rapidly increasing amounts of data, and it is inevitable that some of this data will be divulged to other entities. It is important not only that this data is shared with explicitly *authorised* entities, but also that those entities are not able to access more data than their authorisation permits; the disclosure of information to entities that do not have explicit permission to access it is known as *information leakage*.

The consequences of information leakage can be severe. The most noticeable victims are the data subjects themselves: obvious potential outcomes include financial loss if electronic payment information is disclosed to those with the means of abusing it (as was potentially the case with the credit card details of 77 million PSN subscribers), and identity theft if personally identifiable information is revealed (as was the case with the Social Security numbers of AT&T's affected customers, in an era when Social Security numbers are *de facto* authenticators for US citizens). There are often less obvious, more sinister outcomes for affected data subjects. The individual who compromised BPAS's web server intended to publicise the names of women who

had contacted the charity seeking abortion advice; given the highly personal and sensitive nature of such a procedure, this information could have been used to blackmail them. Similarly, the employees of AT&T's third-party contractor could have used the telephone call metadata they illicitly obtained to infer much more information about the daily activities of the subscribers — one of the greatest revelations of the global surveillance agency disclosures of 2013 is the extent to which such metadata readily reveals details about the private lives of its subjects (Stanford Report, 2014).

It is easy to forget, however, that the data controllers themselves are often also victims. Again, financial loss is a concern: while Sony and the FBI investigated the PSN data breach, the service was temporarily taken offline and remained closed for almost a month, causing an enormous loss of revenue for Sony; in total, the episode cost the company approximately $171 million (CBS News, 2011). Similarly, AT&T offered to pay for one year of free credit monitoring for all subscribers affected by their data breach. Regulatory frameworks may also punish infringing data controllers: organisations are often compelled to report the occurrence of information leaks under threat of fines or legal action, and may yet be punished even if leaks *are* declared: two years after the attack on their server, BPAS were fined £200,000 by the Information Commissioner's Office, the UK's national data protection authority, for improper storage of personal data. A much longer-lasting risk is reputational damage: consumers may think carefully before conducting business with companies that have previously disclosed their customers' personal information to attackers, and women seeking abortion advice may be less inclined to contact a charity with a public track record of accidentally leaking information about those who ask it for help.

Clearly, then, it is in the best interests of all concerned that information leaks do not occur.

1.2 Preventing Information Leakage

Denning and Denning (1979) were amongst the earliest to recognise the importance of preventing or mitigating the effects of information leakage. They identify four methods of safeguarding sensitive data: access control, cryptographic control, inference control, and information flow control.

Access control is the most widely used method of safeguarding data. As the name implies, access to sensitive data is restricted to those entities (*i.e.* users or programs) authorised to access it; an entity must *authenticate* themselves (*i.e.* prove their identity), at which point their request to access the data is either accepted or rejected depending on whether they have the authorisation to access it. However, as Hedin and Sabelfeld (2011) argue, in many cases access control is not fine-grained enough to provide adequate information security guarantees: access to the data is either completely allowed or completely forbidden; entities that are granted access to the data must therefore be trusted not to leak any sensitive information.

Cryptographic control uses encryption to protect sensitive data during transmission or storage. This prevents unauthorised entities from being able to read the data by eavesdropping on or intercepting communications between the system protecting the data and its authorised entities. However, as with access control, it is not possible to restrict what entities do with the data after it has been revealed to them: trust must be placed in the entity not to leak sensitive information after it has been decrypted. Additionally, if cryptography is used incorrectly to protect sensitive data, it may be possible for an unauthorised entity to intercept the encrypted data and decrypt it, thus bypassing cryptographic controls entirely.

Inference control prevents unauthorised entities from consolidating and applying their prior knowledge of data they have the authorisation to access to infer information about more sensitive data that they do not (an *inference attack*). This is a particularly desirable safeguard for a database: the ideal is that wide-ranging statistics provided

by aggregate queries should not leak information about *any* sensitive data that was used to construct those statistics. In practice, however, perfect inference control is unachievable; Denning and Denning argue that a more appropriate use of inference control is to determine how expensive it would be for an unauthorised entity to launch a successful inference attack against sensitive data, and to design the system so that this expense is prohibitively great.

A stronger safeguard is *information flow control*, which monitors how sensitive data moves (or *flows*) through a system. Systems implementing information flow controls have a *security policy* imposed on them that specifies where and how sensitive data is permitted to flow; systems are considered insecure if sensitive data that should be inaccessible to an unauthorised entity flows into data that the entity is authorised to read. This is a fundamentally different safeguard to the previous three: as Smith (2007) states, they ensure that sensitive information is not *released*, whereas information flow control ensures that it is not *propagated*. Of the four safeguards we have discussed, it is perhaps the most satisfactory for achieving the goal of eliminating information leakage.

This thesis therefore focuses on detecting information leaks through *information flow analysis*.

1.3 Information Flow Analysis

Information flow analysis considers how systems (typically programs) can be statically or dynamically analysed to determine the flows of information that occur in them. For such an analysis to be meaningful, a formal definition of "information flow" is required. This involves defining an *information flow model* consisting of three components:

(a) a *system model*, a theoretical abstraction or encoding of the system being analysed;

(b) an *attacker model*, an encoding of the capabilities and goals of an unauthorised entity trying to violate the security of the system (as defined by the system model), typically specifying whether the entity has prior knowledge of the system's behaviour or its sensitive information and whether the entity is able to provide input to (or otherwise interact with) the system while it is executing; and

(c) a *security policy*, a precise definition of where sensitive information may flow when it is processed by the system.

Thus, we state that a system is secure if and only if the model of the system satisfies the given security policy, given the presence of an attacker with the capabilities defined by the attacker model. This also implies that the quality of an information flow analysis depends largely on how accurately the information flow model reflects the reality of the system and its attackers.

The task of encoding a system as a system model and performing an information flow analysis on it to verify its security could be performed manually, but given the complexity of modern systems it is much more desirable for the entire process to be automated. Finding the best way of achieving this is an active field of research in computer security; current work falls into one of two branches.

Qualitative information flow analysis typically focuses on the compartmentalisation of information: similarly to real-world governmental information classification systems, the system model sorts the processed information into categories and arranges these categories in a hierarchy, which can be represented as a lattice; the security policy defines the directions in which information may and may not flow. For instance, a system model may consist of the classifications "top secret", "confidential" and "unclassified" arranged in a linear hierarchy, and the accompanying security policy may state that information can only flow upwards in the hierarchy: no "top secret" information may flow into the "confidential" or "unclassified" categories, and no "confidential" information may flow into the "unclassified" category. (Systems that satisfy this security policy are said to have the *noninterference* property — but we shall show

in Chapter 3 that it is unrealistic to expect all but the most trivial of systems to have this property.)

A successful qualitative information flow analysis answers the question "does this system leak information?". While it is helpful to know the answer to this question, it does not give a particularly convincing security guarantee; as we shall see in Chapter 3, many programs *must* leak at least a small amount of information to provide their intended functionality. The canonical example is a password-checking program: even the most secure implementation of such a program will still leak a very small amount of its secret information (*i.e.* the correct password) to an attacker — specifically, whether or not the password they provided was valid. When discussing the security of a system, it is therefore often more useful to speak not in terms of *whether* it leaks information, but instead *how much* information it leaks.

This is the approach taken by *quantitative information flow analysis*; it provides bounds on the amount of sensitive information that is leaked to an attacker. A quantitative security policy, therefore, is not just a statement of *where* sensitive information may flow, but also *how much* of it may flow there before the system is deemed insecure. In quantitative information flow models, the severity of information leaks is usually measured using information theory (Shannon, 1948), a branch of mathematics and computer science concerned with quantifying information; one of the most common metrics for measuring information leakage is the *mutual information* of (*i.e.* the amount of information shared between) the system's secret data and the data accessible by an attacker when both types of data are treated as individual variables with their own *entropy* (*i.e.* uncertainty in their possible value from the perspective of an attacker). Many other metrics are available; for example, *min-entropy leakage* quantifies the vulnerability of a system's secret data to an attacker with the ability to make a single attempt at correctly guessing the secret data after observing the data they are permitted to access — this metric is becoming increasingly popular because of the security guarantees it provides in a setting where an unauthorised entity only has one opportunity to attack

the system, *e.g.* because a single incorrect guess triggers an alarm (Smith, 2011).

As far as the practical application of these information flow models is concerned, a particular weakness in most of the existing models in the literature — both qualitative and quantitative — is that they feature system models that make unrealistic assumptions about the manner in which programs typically process secret information and produce output visible to attackers. Many assume that all of the secret information processed by a program is present just before it begins executing, and that all of the information visible to an attacker is present just after it terminates. Additionally, some models (sometimes the *same* models) assume that only individual pieces of secret and observable information are processed by a program. In reality, programs often exhibit much more complex behaviour: some produce output before and after processing secret information, some may process multiple types of secret information in different stages, the secret information being processed may itself be modified during execution, *etc.* It is prohibitively difficult to model real and complex systems in these system models, and consequently many theoretically sound information flow models simply cannot realistically be used to analyse the types of programs that commonly exist in the real world.

A related secondary weakness is that few existing information flow models feature system models that can be integrated into real-world programming languages; this is problematic because a requirement for programmers to transform their programs into a formal model in order to perform an information flow analysis on them will in all likelihood simply lead to the analysis being considered too difficult or time-consuming and thus not being performed at all, leading to an ever-increasing number of programs being vulnerable to information leakage.

1.4 Contributions of this Thesis

This thesis makes two significant contributions to the information flow analysis literature.

The first is a novel "point-to-point" quantitative model of information flow, with a system model based upon CH-IMP, a probabilistic imperative language with basic features found in all modern programming languages: variable assignment and scoping, branching and looping. Provided that a system can be modelled in CH-IMP, our information flow model is able to analyse it. CH-IMP's information flow model differs from existing quantitative information flow models in that it does not restrict the occurrence of secret or publicly-observable information at all: either type of information may occur anywhere in a program, including inside complex code structures such as nested branches or loops or blocks of code that are only executed with a low probability.

CH-IMP's most novel feature is the addition of two commands, secret and observe, that signify the occurrence of secret and publicly-observable values respectively at specific points in the program code — hence a "point-to-point" model that answers the question "what does an attacker learn about these secret values at these specific points in the program by observing these publicly-observable values at these other specific points?". We show that CH-IMP has a sound basis by formally defining the execution of a CH-IMP program in terms of a *discrete-time Markov chain* (DTMC), a well-understood probabilistic model; the secret and publicly-observable information that occurs during execution of a program is defined in terms of a discrete *joint probability distribution*, from which a great number of information-theoretic leakage measures can be derived (although in this thesis we focus on mutual information and min-entropy leakage). With the aid of many motivating examples and two in-depth case studies, along with a software implementation of our information flow model, we demonstrate that CH-IMP is capable of analysing the flows of information that occur in typical sys-

tems and is particularly suited to analysing probabilistic programs and protocols of low to moderate complexity.

Due to the DTMC-based semantics of CH-IMP program execution, however, this information flow model is ill-suited to analysing complex systems that contain too many paths of execution to be analysed precisely in polynomial time — although this is a limitation of *any* model that explores the entire state space of a probabilistic program, rather than of ours specifically. Our second significant contribution is a relaxed form of this information flow model, in which there can be some margin of error in the joint probability distribution of the secret and publicly-observable information. Using results from the statistics literature, we show that the information-theoretic leakage measures of mutual information and min-entropy leakage can still be computed from this approximate joint probability distribution, and that good bounds on their accuracy can even be derived. Given that this distribution need only be an approximation of the true distribution, this frees us from analysing systems in CH-IMP's formal system model for precisely computing the true distribution; we show that this estimated joint probability distribution can be derived via the sampling of a program and recording of the secret and publicly-observable information that occurs during execution. We present new algorithms for determining when the sample size is large enough for mutual information and min-entropy leakage estimates to be computed, thus minimising the amount of time taken to perform a successful information flow analysis of the program.

Finally, with the help of three more detailed case studies, we demonstrate that this relaxed model and sampling method can be used to estimate the size of information flows in realistic programs written in Java, a real-world programming language, and present a second software tool for automatically performing quantitative information leakage analyses on Java programs.

1.5 Thesis Structure

The thesis is structured as follows.

Part I addresses existing research on information flow analysis. In Chapter 2 (p. 22) we provide definitions and theorems from probability theory and information theory that form the foundations of the research area. In Chapter 3 (p. 42) we review the information flow analysis literature, and inspect some existing information flow models and analysis tools. We identify features of existing models that preclude them from analysing flows of information that typically occur in complex systems, and thus motivate the creation of a new model.

Part II presents our new quantitative information flow model. In Chapter 4 (p. 77) we introduce the probabilistic CH-IMP language and define our point-to-point model of information flow; we show how the mutual information and min-entropy leakage measures of information leakage can be derived from the DTMC induced by the execution of a CH-IMP program. In Chapter 5 (p. 111) we demonstrate an implementation of the CH-IMP language and information flow model in a software tool, `chimp`, and in Chapter 6 (p. 144) we show how `chimp` can be used to verify information security policies in two realistic case studies of low to moderate complexity. The example programs we present in this chapter are easily expressed in CH-IMP but difficult to express in the system models of other information flow models, underlining one of our major contributions to the qualitative information flow analysis literature.

Part III investigates how a relaxed version of our information flow model can be used to analyse more complex systems. In Chapter 7 (p. 168) we demonstrate that, with the aid of the statistics literature, the information leakage measures that can be computed from CH-IMP's analysis of a probabilistic program can instead be estimated from samples of the secret and publicly-observable information that occurs during execution of the program. We present leakiEst, a tool and Java library for estimating information leakage measures from probabilistic system samples. We also show how

this technique can be automated in a real-world programming language by presenting LeakWatch, a quantitative information flow estimation tool for Java. In Chapter 8 (p. 199) we provide three further case studies demonstrating that our method is viable for quantifying the information leakage of complex Java programs.

Finally, we make our closing remarks in Part IV: in Chapter 9 (p. 219) we conclude and present ideas for future work in this research area.

1.6 List of Publications

This thesis is based upon our following publications in the literature:

- Tom Chothia, Yusuke Kawamoto, Chris Novakovic, and David Parker. Probabilistic Point-to-Point Information Leakage. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 193–205, New Orleans, Louisiana, USA, June 2013b. IEEE Computer Society

- Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. A Tool for Estimating Information Leakage. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 690–695, Saint Petersburg, Russia, July 2013a. Springer

- Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. LeakWatch: Estimating Information Leakage from Java Programs. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS 2014), Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 219–236, Wrocław, Poland, September 2014. Springer

The publications upon which each chapter is based are detailed in the introduction to the chapter.

I

Background

2

Definitions

2.1 Probability Theory

In Chapter 4 we define our new point-to-point information flow model for probabilistic programs, in Chapter 5 we compare this model with another that can also quantify the information leakage that occurs from probabilistic programs, and in Chapter 7 we present techniques for estimating quantitative measures of information leakage; all of these topics require knowledge of probability theory, and it is therefore a prerequisite for most of what follows in this thesis. The definitions provided in this section are all standard and can be found in any introductory text for probability theory (*e.g.* Freund, 1973).

Probability theory is concerned with *experiments* and the *outcomes* of those experiments; in this field, an experiment is typically the execution of a program to observe how it behaves, and the outcome of interest is typically the occurrence of a particular type of information during execution; *e.g.*, "when executing program $\mathcal{P}$, what secret information $\mathcal{S}$ does it process?".

The *sample space* Ω of an experiment is the set of all possible outcomes o of the experiment; in this thesis, we are concerned only with *discrete* sample spaces; *i.e.*, we assume that the outcomes from an experiment are countable. An *event* E is a subset of the sample space (*i.e.* a set of outcomes) to which a *probability* has been assigned;

the probability of an event occurring is denoted $P(E)$, and defines the likelihood of an experiment ending in a particular way. The probability of an event occurring is formally defined as follows:

Definition 2.1 (probability of an event)

If the event E in a discrete sample space is comprised of the outcomes $o_1, o_2, \ldots, o_n$, the probability of E occurring is

$$P(E) = \sum_{i=1}^{n} P(o_i).$$

We often need to combine multiple events; we use standard set theory notation and terminology when referring to the *subset* $E \subset F$, the *union* $E \cup F$ (*i.e.* the outcomes of Ω that are members of E or F), the *intersection* $E \cap F$ (*i.e.* the outcomes of Ω that are members of both E and F), and the *complement* $\overline{E}$ (*i.e.* the outcomes of Ω that are not members of E).

Formally, the *probability measure* P must satisfy *Kolmogorov's axioms of probability*:

Theorem 2.1 (Kolmogorov's axioms of probability)

First axiom. The probability of an event occurring is a non-negative real number:

$$\forall E \subset \Omega \qquad P(E) \in \mathbb{R}, \quad P(E) \geq 0.$$

Second axiom. The probability that an event in the sample space will occur is equal to 1:

$$P(\Omega) = 1.$$

Third axiom. If $E_1, E_2, \ldots, E_n$ is a sequence of mutually exclusive events, the

probability of one of the events occurring is

$$P(E_1 \cup E_2 \cup \ldots \cup E_n) = \sum_{i=1}^{n} P(E_i).$$

Kolmogorov's third axiom requires that the events are mutually exclusive, but the *general addition rule* permits the addition of events that are not:

Theorem 2.2 (general addition rule)

If $E, F \subset \Omega$ are two events in a discrete sample space, the probability of one of these events occurring is

$$P(E \cup F) = P(E) + P(F) - P(E \cap F).$$

Kolmogorov's third axiom is a special case of the general addition rule, since, for mutually exclusive events, $P(E \cap F) = 0$.

Kolmogorov's axioms of probability have the following consequences:

Theorem 2.3 (consequences of Kolmogorov's axioms)

Monotonicity. If $E, F \subset \Omega$ and $E \subset F$, then $P(E) \leq P(F)$.

Probability of the empty set. $\forall \Omega \quad P(\emptyset) = 0$.

Bounds on probability. $\forall E \quad 0 \leq P(E) \leq 1$.

Complementary events. If E and $\overline{E}$ are complementary events in Ω, then $P(\overline{E}) = 1 - P(E)$.

In this field, we are often concerned with the *dependence* of events: if one event occurs, what is the probability that another event occurs? This is quantified by *conditional probability*.

Definition 2.2 (conditional probability of events)

If $E, F \subset \Omega$ are two events in a discrete sample space, and the probability of event E occurring is non-zero, the conditional probability of F given E is

$$P(F \mid E) = \frac{P(E \cap F)}{P(E)}.$$

A consequence of Definition 2.2 is the *multiplication rule of probability*.

Definition 2.3 (multiplication rule of probability)

If $E, F \subset \Omega$ are two events in a discrete sample space, and the probability of event E occurring is non-zero, the probability of an outcome in both E and F occurring is

$$P(E \cap F) = P(F \mid E) \cdot P(E)$$

or alternatively, if the probability of event F occurring is non-zero,

$$P(E \cap F) = P(E \mid F) \cdot P(F).$$

From Definition 2.2, we can derive *Bayes' theorem*, which relates $P(E \mid F)$ to $P(F \mid E)$.

Theorem 2.4 (Bayes' theorem)

If $E, F \subset \Omega$ are two events in a discrete sample space, and the probability of event E occurring is non-zero,

$$P(F \mid E) = \frac{P(F) \cdot P(E \mid F)}{P(E)}.$$

Two events are *independent* if the occurrence of one does not affect the occurrence of the other.

Definition 2.4 (independence of events)

If $E, F \subset \Omega$ are two events in a discrete sample space, they are independent if and only if

$$P(E \cap F) = P(E) \cdot P(F).$$

A *discrete random variable* is a variable whose value varies though chance; *i.e.*, the exact value it assumes from some countable set of possible values is subject to some probability. Formally, it is defined as a function:

Definition 2.5 (discrete random variable)

A discrete random variable X is a real-valued function defined over the observations of a sample space Ω.

It is conventional to denote a random variable with an upper-case letter, and one of its possible corresponding values with the equivalent lower-case letter; *i.e.*, "the random variable X assuming the value x" is denoted with $X = x$. We follow this convention in this thesis, denoting random variables with the letters X, Y and Z.

A *discrete probability distribution* is a function that describes the probabilities associated with the occurrence of possible values of a discrete random variable; it must satisfy Kolmogorov's first and second axioms (Theorem 2.1, p. 23) in a similar manner to probability measures.

Definition 2.6 (discrete probability distribution)

The probability distribution P_X of a discrete random variable X is a function such that $P_X(x) = P(X = x)$, provided that

$$\forall x \in \mathrm{dom}(X) \qquad P_X(x) > 0$$

and

$$\sum_{x \in \mathrm{dom}(X)} P_X(x) = 1.$$

In this thesis, to avoid confusion of probability measures with probability distributions, we denote probability distributions with a subscript signifying the random variable whose probabilities the distribution defines; *e.g.*, P_X is the probability distribution for the random variable X.

The *mean* of a probability distribution is its expected average value.

Definition 2.7 (mean of a discrete probability distribution)

The mean of a discrete probability distribution P_X is given by

$$\mu_X = \sum_{x \in \mathrm{dom}(P_X)} x \cdot P_X(x).$$

The *variance* of a probability distribution, informally, is a measure of how much the probability distribution diverges from its mean.

Definition 2.8 (variance of a discrete probability distribution)

The variance of a discrete probability distribution P_X is given by

$$\sigma_X^2 = \sum_{x \in \mathrm{dom}(P_X)} (x - \mu_X)^2 \cdot P_X(x).$$

There are two probability distributions that shall be of interest to us when we review related work from the literature on the statistical estimation of information leakage measures in Sections 3.3.1 and 3.3.2 (pp. 66 and 70) and present our own advances in the practical estimation of these measures in Chapter 7; they are the *normal distribution* and the χ^2 *distribution*. We will also encounter the *uniform distribution*, in which each outcome occurs with equal probability.

Definition 2.9 (uniform distribution)

A discrete probability distribution P_X with k outcomes that occur with a non-zero

probability is uniform if and only if

$$\forall x \in \mathrm{dom}(P_X) \qquad P_X(x) = \frac{1}{k}.$$

As mentioned earlier, we shall often be concerned with the dependence between events; a *joint probability distribution* defines the relationship between multiple random variables (although in this thesis we shall mostly concern ourselves with the relationship between two random variables). It defines the probability that n random variables will assume n given values simultaneously.

Definition 2.10 (joint probability distribution)

The joint probability distribution $P_{X_1,\ldots,X_n}$ of n discrete random variables $X_1,\ldots,X_n$ is a function such that

$$\forall x_1 \in \mathrm{dom}(X_1),\ \ldots,\ x_n \in \mathrm{dom}(X_n) \qquad P_{X_1\ldots X_n}(x_1,\ldots,x_n) = P(X_1 = x_1 \cap \ldots \cap X_n = x_n).$$

Again, to avoid confusion with probability distributions, we denote joint probability distributions with a subscript signifying the random variables whose joint probabilities the distribution defines; *e.g.*, P_{XY} is the joint probability distribution for the random variables X and Y.

Using Definition 2.10, we can derive P_{XY} from the probability distributions P_X and P_Y; we can remove the influence of either P_X or P_Y from P_{XY} through a process known as *marginalisation*:

Definition 2.11 (marginal probability distributions)

The joint probability distribution P_{XY} of two discrete random variables X, Y is

marginalised thus:

$$P_X(x) = \sum_{y \in \mathrm{dom}(Y)} P_{XY}(x, y);$$
$$P_Y(y) = \sum_{x \in \mathrm{dom}(X)} P_{XY}(x, y).$$

Note that this allows us to "recover" both P_X and P_Y from P_{XY}, a fact we rely upon heavily when we quantify information leakage in our model in Section 4.5 (p. 106).

A *conditional probability distribution* quantifies causality: it relates the occurrence of a past event with the probability of the occurrence of a future event, and is defined similarly to the conditional probability of two events (Definition 2.2, p. 25).

Definition 2.12 (conditional probability distributions)

If P_{XY} is the joint probability distribution of two discrete random variables X, Y, and P_X and P_Y are the marginal probability distributions of X and Y respectively (see Definition 2.11), the conditional probability distribution $P_{X|Y}$ of X given that $Y = y$ is given by

$$P_{X|Y}(x \mid y) = \frac{P_{XY}(x, y)}{P_Y(y)}$$

provided that $P_Y(y) > 0$, and the conditional probability distribution $P_{Y|X}$ of Y given that $X = x$ is given by

$$P_{Y|X}(y \mid x) = \frac{P_{XY}(x, y)}{P_X(x)}$$

provided that $P_X(x) > 0$.

Finally, we can use the definition of independence for events (Definition 2.4, p. 26) to formally state a definition of independence for random variables:

Theorem 2.5 (independence of discrete random variables)

If P_{XY} is the joint probability distribution of two discrete random variables X, Y, and P_X and P_Y are the marginal probability distributions of X and Y respectively (see Definition 2.11), X and Y are independent if and only if

$$\forall x \in \mathrm{dom}(X),\ y \in \mathrm{dom}(Y) \qquad P_{XY}(x, y) = P_X(x) \cdot P_Y(y).$$

Note that Theorem 2.5 is a qualitative measure: either two random variables are dependent, or they are independent. In computer security, it is often much more useful to consider *how much* two quantities are related — this is the case for information flow analysis, as we shall argue in Chapter 3. We will see how to quantify the dependence of two random variables when we define mutual information in Section 2.3.1 (p. 33).

2.2 Discrete-Time Markov Chains

In this thesis, we primarily concern ourselves with the execution of programs that exhibit probabilistic behaviour. In Chapter 4, we shall define the execution of a program written in our own probabilistic language in terms of a *discrete-time Markov chain* (DTMC). We shall also review related work that makes use of DTMCs in Section 3.2.8 (p. 60).

A discrete-time Markov chain is a process that transitions randomly between states. Because DTMCs are random processes, probability theory is used to describe their behaviour.

Definition 2.13 (Discrete-time Markov chain)

A discrete-time Markov chain is a tuple $\mathcal{D} = (S, \bar{s}, \mathbf{P})$ where:

- S is a (countable) set of states;
- $\bar{s} \in S$ is an initial state;

- $\mathbf{P} : S \times S \to [0,1]$ is a *transition probability matrix*, a discrete probability distribution such that $\sum_{s' \in S} \mathbf{P}(s,s') = 1$ for all $s \in S$.

All processes that can be modelled with a DTMC begin in some initial state $\bar{s}$, and transition into other states in S with some probability; the matrix $\mathbf{P}$ encodes the probability of transitioning from one state s into a succeeding state s'. Note that the probability of transitioning into a succeeding state depends only on the current state, and not on any previous states; this is known as the *Markov property*.

We use the term *accepting state* to denote a state in a DTMC in which it is not possible to transition into any other state (note that, given the definition of $\mathbf{P}$ in Definition 2.13, a self-transition must therefore exist).

Definition 2.14 (accepting state of a DTMC)

An *accepting state* $\underline{s}$ of a discrete-time Markov chain $\mathcal{D}$ is a state $s \in S$ such that $\mathbf{P}(s,s) = 1$, and $\mathbf{P}(s,t) = 0$ for all $t \in S$ such that $s \neq t$.

Thus, an accepting state in a DTMC is analogous to an accepting state in (*e.g.*) a finite-state automaton. The set of all accepting states in a DTMC is denoted with $\underline{S}$.

A *path* of a DTMC is a sequence of successive states from the initial state to some other state that occurs with a non-zero probability.

Definition 2.15 (path of a DTMC)

A *path* ω of a discrete-time Markov chain $\mathcal{D}$ is a finite sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = \bar{s}$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$.

The set of all paths that occur in a DTMC is denoted with $\Omega_{\mathcal{D}}$. Since a path is a finite sequence of states, and transitions occur between states with some probability, the probability of a particular path occurring can be computed by multiplying together the probabilities of each transition occurring:

Definition 2.16 (probability of a path of a DTMC occurring)

The probability $P(\omega)$ of a path $\omega = \langle s_0, s_1, \ldots, s_n \rangle$ of a discrete-time Markov chain $\mathcal{D}$ occurring is

$$P(\omega) = \prod_{i=1}^{n-1} P(s_i, s_{i+1}).$$

Thus, the probability of entering a given accepting state can be computed by multiplying the probability of each transition that occurs between the path's initial and accepting states:

Definition 2.17 (probability of entering an accepting state of a DTMC)

The probability $P(\underline{s})$ of a discrete-time Markov chain entering an accepting state $\underline{s}$ is

$$P(\underline{s}) = \sum_{\omega = \langle s_0, s_1, \ldots, s_n \rangle \in \{\Omega_{\mathcal{D}} | s_n = \underline{s}\}} P(\omega).$$

In this thesis, we shall assume that a DTMC always eventually enters an accepting state; *i.e.*, we assume that paths are not infinitely long and that the final element in every path $\omega \in \Omega_{\mathcal{D}}$ is an accepting state.

In Section 4.2 (p. 81) we shall define the execution of a terminating probabilistic program in terms of a DTMC, and in Section 4.5 (p. 106) we shall show how the probability of entering each accepting state of the DTMC can be used to quantify the program's information leakage in an intuitive manner.

2.3 Information Theory

In Section 3.2 (p. 51) we shall present some related publications from the literature on the topic of quantitative information flow analysis. Each publication uses one of several information-theoretic measures — mutual information, a derivative of mutual information, or min-entropy leakage — as a quantitative information flow measure. Therefore, to understand this work, some knowledge of information theory is required.

Information theory, developed by Shannon (1948), is a large field in its own right; it is primarily concerned with quantifying the reliability of data transmission along *communication channels* that accept some input, defined by a random variable, and produce some corresponding output, defined by another random variable. In this section, we restrict ourselves to defining the quantities required to understand both the related work presented in Section 3.2 and our own contributions to the literature.

2.3.1 Definitions for Mutual Information

We ended our introduction to probability theory with a qualitative definition of the independence of two random variables X and Y (Theorem 2.5, p. 30). We suggested that this definition is somewhat unsatisfactory: rather than simply stating that there exists a dependency between two random variables, it would be much more useful to state *how much* of a dependency exists. *Mutual information* is such a measure of the dependency between two random variables; it is a quantity that relates the amount of uncertainty that exists when the random variables assume their values separately to the amount of uncertainty that exists when they assume their values jointly.

As with the probability theory presented in Section 2.1, these definitions are all standard and can be found in any introductory text for information theory (*e.g.* Cover and Thomas, 2006).

Shannon entropy (or commonly just *entropy*) is a measure of the unpredictability associated with a random variable.

Definition 2.18 (Shannon entropy of a random variable)

The Shannon entropy of a random variable X is given by

$$H(X) = -\sum_{x \in \mathrm{dom}(X)} P_X(x) \cdot \log_2 P_X(x).$$

The entropy of a random variable is a measure of its expected information content,

and is often measured in bits.[1] For example, the entropy of a random variable describing the rolling of a fair die (where the probability of rolling each value is 1/6) is $\sum_{x\in\{1,\ldots,6\}} \frac{1}{6}\log_2\frac{1}{6} \approx 2.58$ bits. If the die were weighted heavily so that a six were guaranteed to be rolled (*i.e.* the probability of rolling a six were 1 and the probability of rolling any other number were 0), the entropy of its random variable would be $(\sum_{x\in\{1,\ldots,5\}} 0\log_2 0) + (\sum_{x\in\{6\}} 1\log_2 1) = 0$ bits:[2] it would be purely deterministic; there would be no unpredictability in its value.

Joint entropy is a measure of the unpredictability associated with *two* random variables; it is defined in terms of their entropies.

Definition 2.19 (joint entropy of random variables)

The joint entropy of two random variables X and Y is given by

$$H(X,Y) = -\sum_{\substack{x\in\mathrm{dom}(X)\\ y\in\mathrm{dom}(Y)}} P_{XY}(x,y)\cdot\log_2 P_{XY}(x,y).$$

The joint entropy of two random variables is therefore a measure of their combined expected information content. The joint entropy of two independent random variables is the sum of their entropies; since there cannot be more uncertainty in two random variables' values than there is when they occur independently, $H(X,Y) \leq H(X) + H(Y)$ for all random variables X and Y.

Conditional entropy measures the unpredictability of one random variable given that another random variable has already assumed a value. It is defined in terms of the joint entropy of both random variables and the entropy of the random variable whose value is known.

[1] If this is the case, logarithms are taken with respect to base 2, as in Definition 2.18 (p. 33). For the remainder of the thesis, we will measure Shannon entropy and other related information-theoretic quantities in bits.

[2] Strictly, this is false: $\log_2(k)$ is undefined at $k = 0$. However, given that $\lim_{k\to 0^+} k\cdot\log_2(k) = 0$, and that k is a probability (*i.e.*, $0 \leq k \leq 1$), we assume that $0\cdot\log_2(0) = 0$ so that entropy can be computed. Similar assumptions are made when computing the other information-theoretic measures defined in this thesis.

Definition 2.20 (conditional entropy of a random variable)

The conditional entropy of the random variable X given the random variable Y is given by

$$\begin{aligned} H(X \mid Y) &= H(X,Y) - H(Y) \\ &= \sum_{\substack{x \in \mathrm{dom}(X) \\ y \in \mathrm{dom}(Y)}} P_{XY}(x,y) \cdot \log_2 \frac{P_X(x)}{P_{XY}(x,y)}. \end{aligned}$$

The greater the conditional entropy of two random variables, the smaller the dependency between them. If the two random variables are independent, the conditional entropy $H(X \mid Y)$ is simply the entropy of X: $H(X \mid Y) = (H(X) + H(Y)) - H(Y) = H(X)$; *i.e.*, knowledge of the behaviour of Y reveals nothing about that of X.

Mutual information is a measure of the information gained about the behaviour of one random variable by observing the behaviour of another; it is defined in terms of the random variables' conditional entropy and the entropy of the random variable whose value is being observed.

Definition 2.21 (mutual information of random variables)

The mutual information of the random variables X and Y is given by

$$\begin{aligned} I(X;Y) &= H(X) - H(X \mid Y) \\ &= H(Y) - H(Y \mid X) \\ &= \sum_{\substack{x \in \mathrm{dom}(X) \\ y \in \mathrm{dom}(Y)}} P_{XY}(x,y) \cdot \log_2 \frac{P_{XY}(x,y)}{P_X(x) \cdot P_Y(y)}. \end{aligned}$$

As suggested by the fraction in the third equation in Definition 2.21, the mutual information of two random variables is the weighted ratio of the probability of them assuming two values simultaneously to the probability of them assuming the same two values independently. Therefore, if two random variables are independent, $\log_2 \frac{P_{XY}(x,y)}{P_X(x) \cdot P_Y(y)} = \log_2(1) = 0$ and they have 0 bits of mutual information; if they are dependent, their

mutual information will be a real number in the interval $(0, \min(H(X), H(Y))]$.

A related measure is *conditional mutual information*, the mutual information of two random variables given that a third random variable has already assumed a value.

Definition 2.22 (conditional mutual information of random variables)
The conditional mutual information of the random variables X and Y given the value of the random variable Z is given by

$$\begin{aligned} I(X;Y \mid Z) &= I(X;Y,Z) - I(X;Z) \\ &= \sum_{\substack{x \in \mathrm{dom}(X) \\ y \in \mathrm{dom}(Y) \\ z \in \mathrm{dom}(Z)}} P_{XYZ}(x,y,z) \cdot \log_2 \frac{P_Z(z) \cdot P_{XYZ}(x,y,z)}{P_{XZ}(x,z) \cdot P_{YZ}(y,z)}. \end{aligned}$$

Finally, the *capacity* of a communication channel is the maximum amount of information that can be transmitted along it.

Definition 2.23 (capacity of a communication channel)
The capacity of a communication channel whose inputs are defined by the random variable X and whose outputs are defined by the random variable Y is given by

$$C = \max_{P_X} I(X;Y)$$

where P_X is the probability distribution of X that maximises the mutual information of X and Y.

Mutual information is a commonly-used measure of information leakage (*e.g.* Wittbold and Johnson, 1990; Gray, 1991) because it can be used to quantify the correlation between the occurrence of a collection of values that an attacker may observe (*e.g.* the publicly-visible output from a program) and the occurrence of a collection of hidden values of interest to the attacker (*e.g.* some secrets processed by the program): if the program's output reveals nothing about its secrets, the mutual information is 0; if the program's output reveals everything about its secrets, the mutual information is the

Shannon entropy of the random variable whose value is a collection of the program's secrets.

As a measure of information leakage, mutual information does not distinguish between situations where a small amount of secret information is leaked with a high probability and those where a large amount of secret information is leaked with a small probability — instead, it measures the information leakage that occurs in the average case; consequently, mutual information is an appropriate measure for modelling an attacker with no prior knowledge of the secret information. We argue that this average-case leakage measure is sometimes preferable to a worst-case leakage measure that apportions a large amount of information leakage to programs that leak information about their secrets in unlikely situations.

For example, consider Algorithm 2.1, a simple function that checks whether a given four-digit PIN is the correct one associated with a given account number; assume that the secret information is the correct PIN for the account number, and that the publicly-observable information is whether the given combination of account number and PIN is correct. It is true that, if the attacker manages to correctly guess the PIN for a given account, the function leaks *all* $\log_2(4)$ bits of the information it was tasked with keeping secret. This appears catastrophic, but it overestimates the information leakage

Algorithm 2.1: checks whether a given PIN is valid for a given account number in a database

Input: db, a hash table mapping account numbers to correct PINs; $account$, the account number provided by the user; pin, the PIN provided by the user
Output: **true** if pin is the correct PIN for $account$; false otherwise

```
function IsCorrectPIN(db, account, pin)
    correct ← false
    for all a ∈ keys of db do
        if a = account and db(a) = pin then
            correct ← true
        end if
    end for
    return correct
end function
```

that occurs in the more likely scenario where the attacker has *no* prior knowledge of the PIN and is simply guessing the PIN for a given account at random. In this case, there is a $1/10000$ probability that the attacker's guess will be correct (resulting in total leakage of the secret), and a $9999/10000$ probability that the attacker's guess will be incorrect (resulting in a minute leakage — namely, that the correct PIN is not the one the attacker guessed), for an average-case leakage of just over 0.001 bits. Now consider the effect of increasing the length of PINs to five digits: the average-case leakage of the function decreases to just over 0.0001 bits, but the worst-case leakage *increases* to $\log_2(5)$ bits in the event that the attacker guesses the PIN correctly. We argue that the second function is no less secure than the first, in the sense that it is no more likely to leak information about its secret to an attacker lacking prior knowledge of the secret, and that a worst-case leakage measure would be unrepresentative of the security offered by the function.

We shall see some examples of mutual information and its derivatives being used as information leakage measures when we review related work in Section 3.2 (p. 51).

2.3.2 Definitions for Min-Entropy Leakage

There are many information-theoretic measures of information leakage other than those derived from mutual information. One of them — min-entropy leakage — is a comparatively recent invention, but has amassed a strong following because of the good operational security guarantees it provides (Smith, 2011). Given two random variables X and Y, it quantifies the increase in the vulnerability of X to having its value correctly guessed in a single attempt through the process of observing the value assumed by Y; unlike mutual information, this permits the modelling of an attacker with prior knowledge of a system's secrets, and is particularly well-suited to modelling an attacker with the ability to make a single guess at the program's secret values by observing its publicly-visible outputs.

The *vulnerability* of a random variable is the probability that the value of the ran-

dom variable is guessed correctly in a single attempt.

Definition 2.24 (vulnerability of a random variable)

The vulnerability of a random variable X is given by

$$V(X) = \max_{x \in \mathrm{dom}(X)} P_X(x).$$

This is intuitive: with no other knowledge of how (or even whether) X's behaviour relates to that of other random variables, an attacker's best strategy to correctly guess the value of X is simply to select the value that it assumes with the highest probability.

$P_X(x)$ is a probability, and therefore $0 \leq V(X) \leq 1$; *min-entropy* represents this probability as an entropy measure that captures X's unpredictability.

Definition 2.25 (min-entropy of a random variable)

The min-entropy of a random variable X is given by

$$\begin{aligned} H_\infty(X) &= \log_2 \frac{1}{V(X)} \\ &= -\log_2 \max_{x \in \mathrm{dom}(X)} P_X(x). \end{aligned}$$

In cases where the behaviour of X is influenced by some other random variable Y, the *conditional vulnerability* of X is the probability that the value of the random variable is guessed correctly in a single attempt, given that the behaviour of Y is known.

Definition 2.26 (conditional vulnerability of a random variable)

The conditional vulnerability of the random variable X given the random variable Y is given by

$$\begin{aligned} V(X \mid Y) &= \sum_{y \in \mathrm{dom}(Y)} P_Y(y) \cdot \max_{x \in \mathrm{dom}(X)} P_{X|Y}(x \mid y) \\ &= \sum_{y \in \mathrm{dom}(Y)} \max_{x \in \mathrm{dom}(X)} P_{XY}(x, y). \end{aligned}$$

Again, this is intuitive: if X and Y are correlated, an attacker's knowledge of Y will influence their best strategy to correctly guess the value of X; certain values of Y (which themselves will be assumed with different probabilities) may indicate a greater likelihood of X assuming certain values. Thus, conditional vulnerability represents the weighted average probability of correctly guessing X's value from a particular value of Y across all possible values of Y.

As with the definition of vulnerability in Definition 2.24, $0 \leq V(X \mid Y) \leq 1$; *conditional min-entropy* is the analogue of min-entropy, and quantifies the amount of unpredictability that remains in X given that Y's behaviour is known.

Definition 2.27 (conditional min-entropy of a random variable)

The conditional min-entropy of the random variable X given the random variable Y is given by

$$\begin{aligned} H_\infty(X \mid Y) &= \log_2 \frac{1}{V(X \mid Y)} \\ &= -\log_2 \sum_{y \in \mathrm{dom}(Y)} \max_{x \in \mathrm{dom}(X)} P_{XY}(x, y). \end{aligned}$$

Finally, *min-entropy leakage* relates the two min-entropies: it is the difference between the *a priori* and *a posteriori* uncertainty about X with respect to Y.

Definition 2.28 (min-entropy leakage of a random variable)

The min-entropy leakage from the random variable X to the random variable Y is given by

$$\begin{aligned} \mathcal{L}_{XY} &= H_\infty(X) - H_\infty(X \mid Y) \\ &= -\left(\log_2 \max_{x \in \mathrm{dom}(X)} P_X(x)\right) + \left(\log_2 \sum_{y \in \mathrm{dom}(Y)} \max_{x \in \mathrm{dom}(X)} P_{XY}(x, y)\right). \end{aligned}$$

Thus, the min-entropy leakage from a program's collection of secrets to its publicly-observable outputs quantifies the average amount of additional information about the

program's secrets that an attacker is able to guess correctly in one attempt by observing the program's outputs. Ideally, an attacker would learn no additional information about the secrets by observing the outputs, and therefore the most secure program would have a min-entropy leakage of 0 bits; insecure programs would reveal information about the secrets such that a min-entropy leakage of n bits would increase the vulnerability of program's secrets to single-attempt guessing attacks by a factor of 2^n.

Smith (2009, 2011) provides further details about min-entropy leakage and the quantities that define it, and presents a more detailed comparison of min-entropy leakage and mutual information as information leakage measures.

3

Previous & Related Work

In Chapter 2, we presented the basic definitions and theorems of probability theory and information theory on which all of the work in the information flow analysis literature is based.

In this chapter, we introduce the two branches of information flow analysis — *qualitative* and *quantitative* — and present an overview of both the historical and the state-of-the-art work in each field. Most of the publications we review in these sections present a novel formal model of information flow in a hypothetical or real-world programming language; some of them additionally feature an implementation of their model. We argue that quantitative information flow models provide better security guarantees than qualitative ones, and provide examples of insecure programs where quantitative information flow analysis provides a much more satisfactory and convincing analysis of the program's security. We discuss the shortcomings of the publications we review, and in doing so we identify where our own contributions to the literature can be made; we motivate the creation of a new quantitative information flow model, which we shall formally define in Chapter 4. We also review more recent work on the statistical estimation of the information-theoretic measures we introduced in Section 2.3 (p. 32), which we shall utilise when we develop an approximation of our information flow model in Chapter 7.

3.1 Qualitative Information Flow Analysis

The information flow analysis literature can be broadly grouped into two branches. The first (and older) of these branches is *qualitative information flow analysis*. Qualitative information flow models determine whether *any* secret information processed by a system is ever revealed as publicly-accessible information in an unauthorised manner; if it is, the system is deemed to be insecure. Note that this is a binary property: either a system is secure or it is not; most of this work therefore focuses on defining the precise properties a system requires to be deemed secure.

3.1.1 Security Lattices

Denning (1976) presents some of the earliest work on qualitative information flow analysis; she considers the information security of programs that store and process pieces of information of varying sensitivity in different memory locations. Her system model consists of a set of *security classes*, discrete compartments to which these memory locations belong. The security classes are analogous to the real-world clearance levels restricting classified information, with some security classes containing more sensitive information than others. These security classes are arranged in a hierarchy, thus defining a relationship between them; this hierarchy can be represented as a lattice. Attackers are assumed to have access to some security classes but not others (it is implied that they typically have access to the lowest security class in the hierarchy), and the model's security policy stipulates that secure information flow is achieved if and only if information flows in directions permitted by the hierarchy.

Two examples presented by Denning are shown in Figure 3.1. In the simple case depicted in Figure 3.1(a), the hierarchy is linear, and each security class — H (high-security), M (medium-security), and L (low-security) — is a subset of the next highest security class; this results in a linear ordered lattice, in which information may only flow upwards into higher security classes.

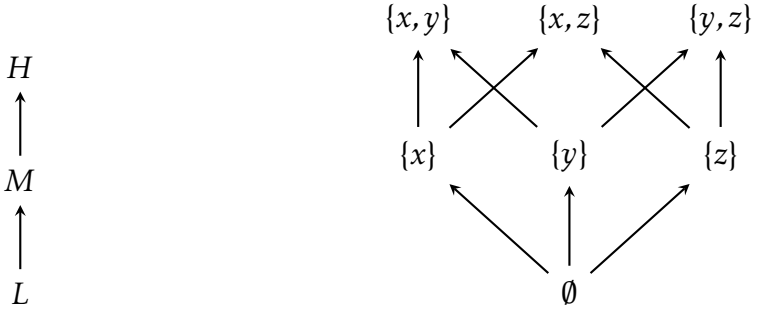
A more complex policy is depicted in Figure 3.1(b): here, the set of security classes is the powerset of $X = \{x, y, z\}$. Information from one of these security classes A is permitted to flow into another security class B if and only if $A \subset B$ (so information may flow from $\{x\}$ to $\{x, z\}$ but not to $\{y, z\}$). Denning argues that this is an effective method of modelling attribute-based flow control: if X were perceived as a set of attributes and the security classes as combinations of attributes, any information flow from a to b would be disallowed by the security policy unless b were to have at least the same attributes as a.

In any case, information flows that violate the hierarchy described by the lattice are considered by the security policy to be unauthorised, and the program is thus deemed to process information insecurely.

3.1.2 Noninterference

Goguen and Meseguer (1982) develop Denning's information flow model further. They present a requirement language for enforcing security policies on systems modelled as

Figure 3.1: two qualitative security policies represented as lattices



(a) a simple lattice representing a linear hierarchy of security classes

(b) a more complex lattice of subsets of $\{x, y, z\}$ representing a non-linear hierarchy of security classes

finite-state automata. Their attacker model assumes that an attacker is able to observe execution traces of processes, and their security policy is based on the concept of *noninterference*: it should not be possible for an attacker to distinguish between two execution traces of a terminating process based on their low-security outputs if their low-security inputs are identical, even if their high-security inputs differ.

Definition 3.1 (noninterference)

A process has the noninterference property if, for all initial states σ and terminating states σ',

$$\sigma =_L \sigma' \quad \Rightarrow \quad [\![P]\!]\sigma =_L [\![P]\!]\sigma'$$

where $=_L$ is a (binary) equality operator for the low-security information in its operands.

Goguen and Meseguer additionally show that the noninterference property can be modelled as a security lattice.

Volpano et al. (1996) formulate Denning's security lattice as a type system and prove its correctness using a hypothetical simple, deterministic language. It can be verified statically that a program written in their language satisfies the noninterference property: security classes in Denning's lattice correspond to "security level" types in Volpano et al.'s language, and a program is badly-typed if an unauthorised information flow occurs between two security levels. Their type system also detects information leaks that arise due to *implicit flow*, where subtleties in the execution of a sequence of instructions can cause information about their behaviour to be inadvertently leaked over a *covert channel*. The canonical example is a variable assignment following a branch operation, as shown in Algorithm 3.1: if it is assumed that y is stored in low-security memory, and x is stored in high-security memory and may only contain the value 0 or 1, there is total leakage of the value of x even though no direct assignment from x to y occurs. Volpano et al.'s type system defines different security levels for the language's expressions, variables and commands; detection of implicit flows is ensured

by guaranteeing that a command C with security level $\mathsf{cmd}{:}\tau$ is only well-typed if every assignment that occurs in C is made to a variable of security level $\mathsf{var}{:}\tau$ or higher.

Ideally, all programs would satisfy the noninterference property, because no high-security information may be leaked to low-security memory without violating the security policy. Practically, however, noninterference is all but unachievable, as it is excessively strict: even the most trivial programs violate it, as shown by the password-checking function in Algorithm 3.2 (p. 46). Assuming that p is stored in high-security memory and that the function's return value is written into low-security memory, it could be argued that this function leaks the minimum possible amount of information about p while still retaining its usefulness: an attacker observing the return value learns either that p equals the string `"mysecret"` in the worst case (*i.e.* a total leakage of p), or that p does not equal `"mysecret"` in every other case (*i.e.* a very small leakage of p each time the function is called; the precise amount depends on the size of the password space). This subtlety is not captured by noninterference.

Algorithm 3.1: assignment based on the value of another variable, demonstrating an implicit flow of information from x to y

```
if x = 0 then
    y ← 0
else
    y ← 1
end if
```

Algorithm 3.2: a simple login function that checks whether a given password p matches a string, and returns an appropriate Boolean value; even this straightforward example violates the noninterference property (Definition 3.1, p. 45)

```
function LOGIN(p)
    if p = "mysecret" then
        return true
    else
        return false
    end if
end function
```

Algorithm 3.2 indicates that for most programs to function as intended, *some* information must be leaked — it is therefore inappropriate to require programs to have the noninterference property for them to be considered secure. Instead, it is usually acceptable for the program to leak only the information necessary for its correct operation, and no more. This raises questions about how to identify which information may be leaked, how it is marked as such, and how it can be verified that a program does not cause undesirable information leaks.

3.1.3 Controlled Interference

A number of alternative, less strict information flow properties have been defined. Bevier et al. (1995) define a weaker property, *controlled interference*, that permits entities to modify each others' internal state, but only over explicitly-defined channels. An example is shown in Figure 3.2: entities A, B and C are only permitted to communicate across the channels specified by the arrows connecting them, so A may communicate with C either by sending information from a_4 over a channel that modifies internal state c_1, or by sending information from a_3 over a channel that modifies c_3. Notice that channels are not implicitly bidirectional: C may not communicate with A over the same channels.

Ferrari et al. (1997) propose a similar technique rooted in object-oriented systems, with security policies containing restrictions and exemptions from noninterference (known as *waivers*) at the method level within objects; they argue that this naturally coincides with the core principles of object-oriented design (*i.e.* encapsulation and modularity). Indeed, it could be argued that their technique suits (for example) Java's *annotation* framework, in which programmer-specified metadata can be affixed to constructs of the language (classes, methods, variables, *etc.*) and is made available for inspection at compile-time; the compiler could grant exemptions to methods to which an appropriate annotation has been affixed. Their model does not, however, account for information flows that may occur between local variables within methods,

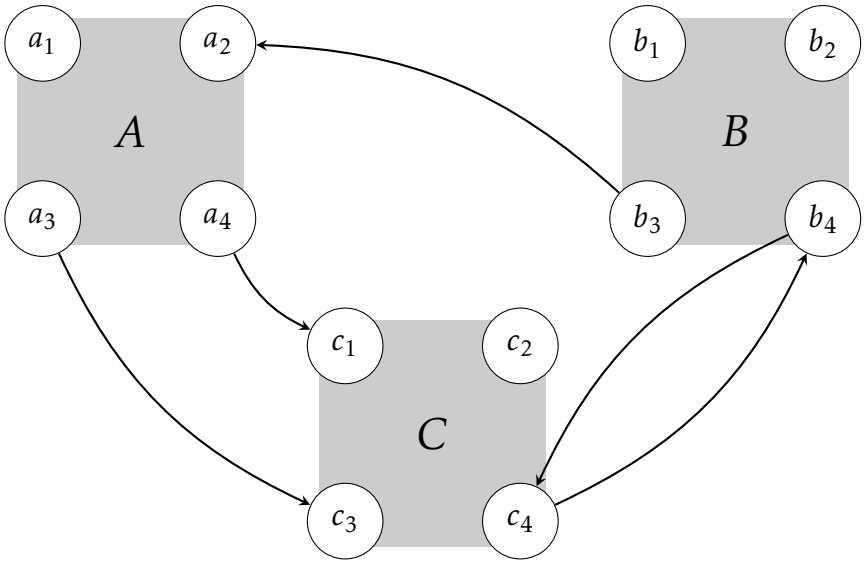
so information leaks may still occur.

3.1.4 Waivers & Decentralised Labels

A different strategy, the *decentralised label model*, is employed by Myers and Liskov (1998). All variables and channels in a system are annotated with labels that define their *owners* (the sources of the data) and *readers* (entities that the owner permits to access that data). Labels are acquired and forgotten as variables are passed over channels, with the security policy prohibiting flows to a variable or channel with a less restrictive label than the original label. The model also permits *selective declassification*, which allows a variable or channel to be relabelled with a less restrictive label by its owner if necessary.

This model is implemented by Myers (1999) as an extension of the Java programming language, in the form of Jif ("Java + Information Flow"), a compiler that acts as a preprocessor for a standard Java compiler: if a Jif program satisfies the security policy by passing the type-checking stage, Jif will convert its input into standard Java syntax

Figure 3.2: a system that satisfies the controlled interference property, showing the channels across which the entities A, B and C are permitted to communicate; this example is adapted from Bevier et al. (1995, Figure 1)



suitable for compilation; otherwise, it will return an error. Jif is notable as it has been used to analyse the information security of large software projects (Hicks et al., 2006).

A real-world example similar to the password-checking function in Algorithm 3.2 is shown in Listing 3.1. A class `passwordFile` with the owner root contains two member variables: `names` (storing user names) and `passes` (storing passwords for the respective user names, and also owned by root). The class contains a single method, `check()`, that checks the validity of a given user name and password and returns a Boolean value indicating whether the combination is valid. The return value of the method has the implied label {user; password}, as these are the method's parameters and are assumed to be known to an attacker. After execution of the `for` loop, the Boolean variable `match` is set to `true` if there is a matching combination, and `false` if there is not; since its value will depend on both the user name and password passed as parameters to the method *and* the values in `passes`, it is implicitly given the label {user; password; root:}. This is a stricter label than the one the method is permitted to return, and if `match` were to be returned by the method, Jif would identify it as the cause of an information leak. The variable must therefore be *declassified* to the weaker policy before it can be returned; accordingly, Jif finds no leaks in this class.

Since Jif is a type system built on Java, it necessarily introduces new syntax into an existing programming language. This raises two important usability issues: in order to use Jif, programmers must (*a*) commit to understanding the underlying information flow model, and (*b*) heavily refactor any existing code to comply with the type system; both of these factors potentially discourage its use. This issue is partially addressed by Jifclipse (Hicks et al., 2007), an integrated development environment that supports Jif's syntax, although it is not developed alongside Jif and is not always compatible with the latest version.

3.1.5 Summary

In this section, we have reviewed a number of qualitative information flow models; they primarily ensure that systems satisfy the noninterference property, or a relaxed variant of it. One of these models — the decentralised label model — has a practical implementation in a real-world programming language, meaning that programmers are not required to formulate abstract representations of their programs in order to analyse their information security.

A key weakness of the literature reviewed in this section is that it is only able to confirm — with varying degrees of success — that information leakage occurs. McCamant and Ernst (2008) consider the challenge of information flow analysis to be not simply the detection of information flows, but the ability to correctly distinguish between unacceptable and acceptable flows. The boundary between unacceptable and acceptable is unclear, especially in complex systems: the output from a program that processes secret information often consists of an abbreviated or summarised version of the secret information, which places a restriction on the assumptions an attacker

Listing 3.1: a Jif password-checking class, adapted from Myers (1999, Figure 4)

```
1  public class passwordFile authority(root) {
2    private String[] names;
3    private String{root:}[] passes;
4
5    public boolean check(String user, String password) where authority(root) {
6      boolean match = false;
7
8      for (int i = 0; i < names.length; i++) {
9        if (names[i].equals(user) && passes[i].equals(password)) {
10         match = true;
11         break;
12       }
13     }
14
15     return declassify(match, {user; password});
16   }
17 }
```

can make about it. McCamant and Ernst's examples include a web site that reveals the last four digits of a customer's 16-digit credit card number, and a censored document that replaces sensitive text with a black rectangle. In the latter example, the black rectangle may superficially appear to leak no information about the sensitive text, but an attacker with knowledge of the font in which the censored text is set may be able to guess the length of the text based on the width of the rectangle and the width of the font's glyphs; furthermore, if the font provides glyphs of differing widths, the attacker may also be able to reconstruct some letters or words in the text.

Thus, rather than considering *whether* a system leaks information about its secrets, it is perhaps more useful to consider *how much* information it leaks, a problem addressed by quantitative information flow analysis.

3.2 Quantitative Information Flow Analysis

Quantitative information flow analysis is the second, newer branch of information flow analysis. Quantitative information flow models measure the amount of secret information leaked by a system, and, instead of indiscriminately permitting or disallowing communication between entities in a system (as is the case with qualitative information flow analysis), a quantitative security policy imposes an upper bound on the amount of information that may be leaked before it is deemed to be unsafe. We have already seen one example where this would be desirable: the password-checking function in Algorithm 3.2 (p. 46). The "false alarm" problem that occurs with this example would be solved, since the programmer would be able to specify in the security policy the maximum amount of information that *should* be leaked by the function, and then confirm that the function satisfies the security policy by leaking no more than that amount.

This raises the additional question of *how* information flows should be quantified; this is one of the topics addressed by the quantitative information flow analysis lit-

erature. In Section 2.3 (p. 32) we described how information theory can be used to quantify the uncertainty in probabilistic systems; the literature we review here makes heavy use of the definitions introduced in Section 2.3, although there is little consensus in the literature about which information-theoretic measure is best for quantifying information leakage. Indeed, it is suggested that the measure chosen should reflect the attacker model: Clark et al. (2005) argue that mutual information (Section 2.3.1, p. 33) best models an attacker with an interest in the system's hidden secrets and the ability to observe a system's publicly-visible information, while Smith (2011) argues that min-entropy leakage (Section 2.3.2, p. 38) best models an attacker with the ability to make only a single attempt at guessing a system's secret information after observing its publicly-visible information. Others argue in favour of conditional mutual information (*e.g.* Mu and Clark, 2009a) or channel capacity (*e.g.* Chen and Malacaria, 2009).

In this section we review the information flow models presented in a range of publications from the quantitative information flow analysis literature. Several of them culminate in the development of information flow analysis tools, which we also briefly review.

3.2.1 Entropy and Channel Capacity of State Transition Systems

Denning (1982, Section 5.1.3) was the first to suggest that information theory could be used to quantify information flow; she describes the relationship between information flow and state transitions in programs in terms of information theory. The fundamental observation is that the amount of information leaked by the flow of information from a private variable x to a public variable y can be characterised as the reduction of x's entropy in successive states, and therefore that the maximum amount of information leaked by the program can be characterised as the capacity of a communication channel with a probability mass function describing values of x as the channel's input and a probability mass function describing values of y as the channel's output.

Denning provides examples that illustrate how the information-theoretic defini-

tions of Section 2.3.1 can be applied to single lines or small blocks of pseudocode to detect undesirable information flows; one of the examples is similar to the implicit flow scenario in Algorithm 3.1 (p. 46). Rather than requiring the programmer to merely acknowledge the existence of this information flow and accept or reject it (as is the case when performing a qualitative analysis of the program's security), Denning shows that information theory can be used to measure the leakage caused by this flow; the programmer may then decide for themselves whether an information flow of the given magnitude constitutes a leak.

3.2.2 The Lattice of Information

Landauer and Redmond (1993) define the *lattice of information* as the complete lattice over the equivalence relations on a set. Higher equivalence relations in the lattice correspond to refinements of lower equivalence relations; *i.e.*, the information provided by the higher equivalence relations is finer-grained than the information provided by the lower ones. If the set is the state space of a deterministic system containing a secret value and an observable low-security value, and the equivalence relation is one that equates all states in which the secret value cannot be distinguished solely from the corresponding observable value, the lattice of information can be regarded as an information flow model: the identity equivalence relation at one extreme of the lattice relates every state to itself, thus indicating total leakage of the secret value to an attacker observing the system's low-security value, while the equivalence relation at the other extreme relates every state to *all* of the system's states, thus indicating that the system possesses the noninterference property.

Of particular interest are the equivalence relations that fall between these extremes in the lattice of information: while the extremes characterise a qualitative notion of information flow, the ordering over the remaining equivalence relations characterises a *quantitative* notion of information flow. Landauer and Redmond establish the relationship between the lattice of information and the capacity of a communication

channel whose input is the secret value and whose output is the observable value. Relationships have also been established between the lattice of information and many other information-theoretic measures, including mutual information, conditional mutual information and min-entropy leakage; Malacaria (2015) provides a more detailed survey of this literature.

3.2.3 Quantitative Security Policies as Satisfiability Problems

Using the notion of quantitative information flow provided by Landauer and Redmond's lattice of information, Heusser and Malacaria (2010) consider the information flow that occurs in systems with high-security and low-security inputs in their initial states and low-security outputs in their final states. In their information flow model, a quantitative security policy is represented as an upper bound on the number of classes in the equivalence relation that equates all states in which the value of the high-security input cannot be distinguished solely from the corresponding low-security outputs; a system is deemed to be insecure if the number of equivalence classes exceeds some positive integer k, in which case the system is equivalent to an information-theoretic communication channel with a capacity greater than $\log_2 k$ bits.

Heusser and Malacaria encode the definition of a system as a function and a quantitative security policy as a *driver function* — consisting in part of a set of assumptions and assertions encoded in Boolean logic — whose purpose is to attempt to prove that the number of resulting equivalence classes exceeds k. The driver function has a standardised structure and can be generated automatically: the function defining the system is called k times (each time consuming as its high-security input a different output from a non-deterministic choice function), and it is assumed that the outputs from the function defining the system are all unique; the function defining the system is then called once more, and it is asserted that this final output is identical to one of the previous outputs.

Verifying whether the driver function's assumptions and assertions hold can be

construed as a satisfiability problem, and so a symbolic model checker can be used to efficiently verify whether the system violates the security policy. The model checker is executed on the driver function, and it attempts to find a counterexample that violates either one of the assumptions *or* the negation of the final assertion; if a counterexample of either type is found, then by definition the number of classes in the equivalence relation exceeds k, the system violates the security policy, and it is therefore deemed insecure.

Heusser and Malacaria adapt their system model to functions written in the C programming language, where a function's arguments may be high-security or low-security inputs and the return value and any pointer arguments to the function are low-security outputs; driver functions are represented using separate code fragments. They use a bounded model checker that encodes both the driver function and the C function being checked for information leakage as a propositional formula that is only satisfiable if there exists an execution trace that violates the security policy. A benefit of the bit-level reasoning facilitated by the model checker is the ability to report detailed information about a violation of the policy, *e.g.* the specific input values that triggered it; this allows the authors to model attacker-controlled low-security input. They show that their technique has practical value by uncovering information leakage vulnerabilities in code from individual modules of the Linux kernel.

3.2.4 Min-Entropy Leakage of Java Bytecode via Symbolic Analysis

Phan et al. (2012) propose a practical method based on the work of Meng and Smith (2011). Their information flow model provides an upper bound on the min-entropy leakage from a single high-security value that occurs before a program begins executing to a single low-security value that exists when the program terminates. They adopt a symbolic approach, in which the output of a program is represented as a sequence of Boolean values corresponding to a bit vector; this allows multiple outputs with similar bit-level representations (named *bit patterns*) to be grouped together for more efficient

model-checking. The model counts the number of possible bit patterns a program can produce as output, thus confirming whether or not the program leaks more than a given number of bits of information. The authors note that this model is designed to detect the presence of and verify the acceptability of small information leaks, rather than quantify large ones.

Phan et al. develop their information flow model in a tool based on a model checker that facilitates the verification of Java bytecode. The authors' experimental tests of the tool's performance suggest that it provides more accurate upper bounds for min-entropy leakage than Meng and Smith's earlier work when analysing programs that contain implicit flows — although this comes at a cost to the speed of the analysis — and programs in which the output space is contiguous (*e.g.* where the range of possible outputs is a list of consecutive integers).

3.2.5 Channel Capacity via Karush-Kuhn-Tucker Conditions

Chen and Malacaria (2009) present a method of quantifying information flow in probabilistic systems. These systems are represented as communication channels, and, following Denning (1982, Section 5.1.3), they define information leakage in terms of the capacity of the channel; this provides an upper bound on the amount of information a system can leak, and therefore models its worst-case behaviour. A novel feature of their work is the derivation of channel capacity using *Karush-Kuhn-Tucker conditions*, which find the extrema of a function subject to some inequality constraints; this enables them to analyse the security of systems in which the attacker's *a priori* knowledge of the secret information can be stated in terms of an inequality, *e.g.* "the secret password processed by the system is n times more likely to be a dictionary word than a meaningless string of letters". The authors prove that finding the capacity of the channel representing the system can be reduced to solving a system of equations.

Although they do not provide a tool that automatically analyses a representation of a probabilistic system, Chen and Malacaria note that compatible techniques for deriv-

ing the relationship between the system's secret input and observable output exist in the literature (*e.g.* Heusser and Malacaria, 2007; Backes et al., 2009), and that existing numerical analysis tools are able to solve the systems of equations that define their information leakage measure. However, a disadvantage of Heusser and Malacaria's technique is its assumption that the system is a program that contains a single secret declared before execution and produces a single output; a similar disadvantage of Backes et al.'s technique is the assumption that a program's secret information is present in its initial state.

Chen and Malacaria present two examples of their technique being used to quantify the information leakage that occurs in probabilistic systems. The first models a simple multithreaded program in which an information leak occurs depending on the order in which the scheduler chooses to schedule the threads for execution; the second models the high-level routing behaviour of an onion routing protocol and quantifies the loss of anonymity that occurs in an onion network when the attacker controls one of its nodes.

3.2.6 Conditional Mutual Information of Probabilistic Programs

Mu and Clark (2009a) analyse flows of information in a simple probabilistic language with assignment, branching, looping and sequential composition constructs. They combine Clark et al.'s (2007) static quantitative information flow analysis technique with Malacaria's (2007) looping construct semantics. This enables them to place tighter bounds on the information leakage from loops that do not terminate, without requiring the manual analysis of the loop's behaviour beforehand.

Mu and Clark's model allows for a single high-security and low-security variable to be defined at the start of a program; there must also be a single low-security variable that is treated as the program's output. Following Clark et al. (2005), the information leakage from the program is defined as the conditional mutual information of the random variable describing the high-security variable's value and the random vari-

able describing the low-security output variable's value when the program terminates, given that the random variable describing the low-security input variable's value is known by the attacker.

Execution of programs with probabilistic semantics is known to become intractable when the probability distributions on the variables' values become too large. To address this, Mu and Clark (2009b) present an abstraction technique that trades the accuracy of the information leakage measure for the ability to analyse programs whose execution would otherwise make such analysis intractable.

Mu and Clark (2011) present an implementation of both the concrete and abstract models, with the same restrictions on high-security and low-security inputs and outputs. They acknowledge that both the language and the design of the tool make it impractical for real-world use, and stress that its purpose is to demonstrate how their models could be implemented.

3.2.7 Network Flow Capacity via Static and Dynamic Analysis

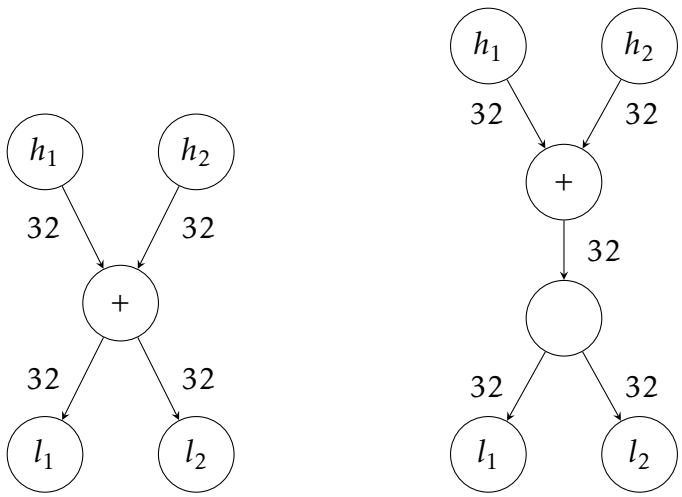
McCamant and Ernst (2008) take a different approach: rather than focusing solely on the static analysis of the source code of programs, the authors combine static and dynamic analysis to measure flows of information. They model a program as a directed acyclic graph and calculate the flow of information from input values to output values as statements are executed — nodes of the graph are operations that can be performed on these values; edges of the graph represent the values themselves and are weighted according to the amount of information about these values that can be transferred by executing statements. The maximum amount of information leaked by the program is the maximum flow from the graph's "input nodes" to its "output nodes" (*i.e.* the *network flow capacity*), which is equal to the weight of the graph's minimum cut (the smallest set of edges separating the input nodes from the output nodes), per the max-flow min-cut theorem (Elias et al., 1956).

Accurate construction of the graph is complex, as demonstrated by the example

in Figure 3.3 (p. 59); this shows two possible graphs that could be generated from $l_1 = l_2 = h_1 + h_2$ — a simple statement that adds together the values of two high-security 32-bit integer variables, assigns the result to a low-security variable, then assigns the value of that variable to another low-security variable. Figure 3.3(a) overestimates the worst-case information flow in this statement, as it allows 32 bits of information to flow from h_1 to l_1 and another 32 bits to flow from h_2 to l_2; in fact there is an information flow of at most 32 bits, as that is the size of the value returned by the addition operator. Figure 3.3(b) correctly models the resulting information flow by inserting a dummy node into the graph after the addition operation, thus clarifying that the statement performs three separate operations in a particular order.

More complex blocks of code, such as loops, must be separated into *enclosure regions* for the analysis to provide accurate bounds on the leakage; flows inside these enclosure regions are measured as if they occur inside separate programs with their own inputs and outputs. Although the enclosure regions can often be inferred automatically as part of a preprocessing stage, it is sometimes necessary to annotate the

Figure 3.3: two possible graph representations of the statement $l_1 = l_2 = h_1 + h_2$, resulting in different quantities of information leakage; this example is adapted from McCamant and Ernst (2008, Figure 1)



(a) leakage of 64 bits (b) leakage of 32 bits

program's source code with the correct enclosure regions manually; the authors estimate that this is necessary in approximately 1 in 4 cases, and is more often required when analysing complex programs that optimise their use of data structures for performance reasons. McCamant and Ernst have implemented their model in a tool that constructs graphs for programs written in C, C++ and Objective-C, and the authors have tested their model with and found information leaks in major software projects written in these languages.

3.2.8 Entropy of Discrimination Relations over Markov Chains

Biondi et al. (2013a) present a model that quantifies flows of information in a program containing a single high-security variable, declared before execution, written in a simple imperative language featuring deterministic and non-deterministic variable assignment, branching and looping constructs.

A program written in the language is modelled as a Markov decision process (MDP) whose states consist of the values of the high-security and low-security variables at that moment. The attacker is modelled as an entity that potentially has prior information about the value of the high-security variable and the ability to distinguish between certain states in the MDP; the prior information about the high-security variable constitutes the *action* in the definition of the MDP. The non-determinism in the MDP is resolved by computing the probability distribution of the possible values of the high-security variable in each state given the attacker's prior knowledge, and the MDP is thus reduced to a Markov chain (Section 2.2, p. 30).

Attackers are not assumed to be able to observe every state in the Markov chain: states that the attacker cannot observe are termed "internal states", and an *observable reduction* is constructed in which these states are hidden. Three *discrimination relations* are then constructed from the attacker definition and the observable reduction: the *observer* discrimination relation (in which two states are in the same class if the attacker cannot distinguish between them), the *secret* discrimination relation (in which two

states are in the same class if the attacker cannot distinguish between the values of the high-security variable in those states), and the *joint* discrimination relation (the intersection of these two discrimination relations).

Given the Markov chains $\mathcal{D}_o, \mathcal{D}_s, \mathcal{D}_{o \cap s}$ induced by these three discrimination relations respectively, the information leakage of the program is defined in terms of the entropies of the Markov chains: $H(\mathcal{D}_o) + H(\mathcal{D}_s) - H(\mathcal{D}_{o \cap s})$.

Biondi et al. (2013b) present a tool, QUAIL, that quantifies the flow of information from the high-security variable to the low-security variables in the program's terminating states in programs written in this imperative language. The authors provide examples that demonstrate how the tool can be used to quantify leaks in small probabilistic algorithms and security protocols.

3.2.9 Summary

The quantitative information flow models we have reviewed in this section improve on the qualitative models of Section 3.1 by providing precise bounds on the amount of information leaked by systems. There are, however, some limitations that prevent them from being used to analyse arbitrary systems and programs.

Several of the models we have presented (Denning, 1982; Mu and Clark, 2009a; Biondi et al., 2013a; Phan et al., 2012) assume that the system processes a single high-security value, produces a single low-security value, or both. These models are theoretically sound, but are uncharacteristic of the behaviour of most systems: it is more often the case that multiple high-security values are processed, and multiple low-security values are revealed to an attacker by the time the system finishes executing. It is unclear how these models can be used to quantify the information leakage that occurs in such systems.

Furthermore, all of these models directly or indirectly assume that systems either process some secret information they are given before execution, produce some publicly-observable information after they terminate, or both. Again, these models

are sound, but they only characterise rather simple programs: in reality, secret and observable information often occurs throughout the execution of a program, rather than simply before execution and upon termination. Although it is possible to reformulate any such program as one where all of its secrets are present before execution and all of its outputs are produced upon termination, it is prohibitively difficult to do so in the general case — and if programmers were required to perform this task manually, it would certainly discourage them from analysing the security of their programs.

As a result of these restrictions, few quantitative information flow models have implementations that can be used to analyse the security of real-world programs. The notable exceptions reviewed in this section are Heusser and Malacaria's and McCamant and Ernst's tools for analysing C programs — these are robust tools that have been used to detect and quantify information leaks in existing code bases, although Heusser and Malacaria's tool focuses on computing the information leakage from functions and McCamant and Ernst's tool requires extensive annotation of large bodies of code for accurate results.

One possible solution to the first two problems is to create a model that examines the entire collections of secret and publicly-observable information that occur during a program's execution, disregarding all other aspects of a program's state as irrelevant to the goal of quantifying the flow of information in the program; thus, the collection of pieces of secret information that occur throughout a program's execution can be viewed as a single random variable, and the collection of pieces of publicly-observable information that occur can be viewed as another random variable. Using these random variables and an information-theoretic measure defined in Section 2.3 (p. 32), we could then quantify the leakage from a program's secrets to its observations — effectively answering the question "what does an attacker learn about the secret information that occurs at all of these specific points during the program's execution by inspecting the publicly-observable values that occur at all of these other specific points?". This would also free us from making restrictive assumptions about the attacker's abilities, because

all of the data required to compute multiple information-theoretic measures would be available; *e.g.*, we could compute the mutual information of these two random variables to quantify the information leakage to an attacker with an interest in the program's secrets and the ability to observe the program's public outputs, or the min-entropy leakage from the former random variable to the latter to quantify the leakage to an attacker with the ability to make only a single attempt at guessing the program's secrets after observing its public outputs.

Using the definitions of Section 2.3 requires that the probability distributions describing the behaviour of the two random variables are known. It is not difficult for a formal model to compute these distributions if the program's state space is small. However, as Mu and Clark (2009b) indicate, precisely computing these probability distributions becomes intractable when the program's state space becomes too large, essentially preventing us from using this model to quantify the information leakage from complex systems. For complex systems with larger state spaces, rather than defining these random variables by computing the probabilities in the distributions describing their behaviour, we could instead *estimate* the probabilities; this would affect the accuracy of the information leakage measure (and we would require some method of quantifying that accuracy), but it would enable us to analyse the security of more complex systems if the estimation process were computationally less expensive.

Estimating information-theoretic measures is therefore the final topic of our literature review.

3.3 Estimation of Information-Theoretic Measures via Sampling

Recall that mutual information and min-entropy leakage are both defined in terms of the random variables X and Y, or the discrete probability distributions P_X and P_Y describing the values those random variables may take (Definitions 2.21 and 2.28, pp. 35

and 40). For many probabilistic systems, one possible technique for estimating these information-theoretic measures is to empirically determine either P_X or P_Y by fixing the value of X or Y (whichever is known) and repeatedly *sampling* the system to determine the most likely distribution describing the corresponding unknown random variable. This process produces an *estimated probability distribution*, denoted $\hat{P}_X$ or $\hat{P}_Y$ respectively. The same process can be followed if neither P_X nor P_Y are known, instead giving the estimated joint probability distribution $\hat{P}_{XY}$, which in turn can be marginalised to give the equivalent $\hat{P}_X$ and $\hat{P}_Y$ (Definition 2.11, p. 28). The estimated probability distribution can be used to estimate the unknown random variable, and both the known and estimated random variables can be used in turn to estimate the mutual information measure $\hat{I}(X;Y)$ or the min-entropy leakage measure $\widehat{\mathcal{L}}_{XY}$. Provided that the system has been sampled a sufficient number of times, we will obtain a good approximation of the true measure $I(X;Y)$ or $\mathcal{L}_{XY}$.

This process is feasible in the context of analysing information flows in probabilistic programs: by assigning the sample space of X to be the secret information that can be processed by a program and the sample space of Y to be the publicly-observable information that can be produced by the program, we can fix the secret information processed by the program for a particular execution — according to the probability of each particular piece of secret information occurring, if those probabilities are known — and sample repeatedly to determine the possible corresponding observable information that the program produces. We can then use the estimated measure as an approximation of the amount of information that leaks from the program.

Chatzikokolakis et al. (2010) identify the many advantages of this technique for quantifying the information that leaks from complex programs, rather than, for instance, building a formal model of the program and analysing it using a model-checker, as in Section 3.2.3 (p. 54): many complex systems have an enormous number of internal states that produce no observable output visible to an external attacker, so much of the effort spent building the model satisfies the means and not the end; and even if

it is possible to produce a formal model of the program within a reasonable time — which is not always the case, given that the state space of the system often explodes exponentially — there is also no guarantee that existing model-checkers are capable of exploring the state space within a similarly reasonable time in order to compute the program's information leakage. Additionally, one must also assume that the translation of the program into a formal model has been performed correctly, and even if it has, the resulting model cannot be used to detect information leaks that occur as a result of faults in the implementation itself (which is often the case).

There is, however, no guarantee that an estimate of one of these measures is an accurate reflection of the true measure: as Chatzikokolakis et al. indicate, performing a numerical analysis on data derived from sampling of a system does not necessarily produce results that are meaningful. It is possible that the numerical analysis could produce a false result because of statistical errors present in the sample, and it would be unclear whether any purported information leakage arose because information actually flowed between X and Y, or simply because the probability distributions describing the random variables insufficiently modelled the behaviour of the system. There is therefore a need to be able to (*a*) distinguish between scenarios where the sample indicates the existence of a statistically significant flow of information in the system and scenarios where, for example, the system has been sampled an insufficient number of times to draw any meaningful conclusions about its behaviour, and (*b*) state the accuracy of the numerical analysis, *i.e.* state with a given confidence how far the estimated measure falls from the true measure, and thus place bounds on the true measure in terms of the estimated measure.

In this section, we review results from the literature on estimating and quantifying the accuracy of these information-theoretic estimates. The need to analyse complex probabilistic systems does not only occur in the field of computer security: in the case of mutual information, finding dependencies between two potentially unknown random variables has applications elsewhere, including machine learning, statistics,

signal processing and neuroscience, and many of the results we summarise in this section are drawn directly or indirectly from the literature in these fields.

3.3.1 Estimating Mutual Information

The literature contains relevant results that address how to compute the accuracy of a mutual information estimate; they define the estimated mutual information in terms of the probability distribution from which the estimate is drawn, and the mean of that distribution (Definition 2.7, p. 27) is defined in terms of the true mutual information. We begin by considering scenarios where neither X nor Y are known, and both have been estimated through sampling. In cases where there is information shared between X and Y, Moddemeijer (1989) shows the following:

Theorem 3.1 (distribution of the mutual information of two related, unknown random variables)

When the random variables X and Y are both unknown and the true mutual information of X and Y is non-zero (*i.e.* $I(X;Y) > 0$), then, for a sufficiently large value of n, the value $2n \cdot \hat{I}(X;Y)$ is drawn from a distribution with mean approximately

$$I(X;Y) + \frac{(\#X - 1) \cdot (\#Y - 1)}{2n} + \mathcal{O}\left(\frac{1}{n^2}\right)$$

and variance approximately

$$\frac{1}{n}\left(\sum_{\substack{x \in \mathrm{dom}(\hat{P}_X) \\ y \in \mathrm{dom}(\hat{P}_Y)}} \hat{P}_{XY}(x,y) \log_2^2\left(\frac{\hat{P}_{XY}(x,y)}{\hat{P}_X(x) \cdot \hat{P}_Y(y)}\right) - \left(\sum_{\substack{x \in \mathrm{dom}(\hat{P}_X) \\ y \in \mathrm{dom}(\hat{P}_Y)}} \hat{P}_{XY}(x,y) \log_2\left(\frac{\hat{P}_{XY}(x,y)}{\hat{P}_X(x) \cdot \hat{P}_Y(y)}\right)\right)^2\right) + \mathcal{O}\left(\frac{1}{n^2}\right).$$

Brillinger (2004) also states the following:

Theorem 3.2 (distribution type of the mutual information of two related, unknown random variables)

The probability distribution from which the value $2n \cdot \hat{I}(X;Y)$ is drawn in Theorem 3.1 is an approximate normal distribution.

Brillinger additionally shows the following in cases where there is *no* information shared between X and Y:

Theorem 3.3 (distribution of the mutual information of two unrelated, unknown random variables)

When the random variables X and Y are both unknown and the true mutual information of X and Y is zero (*i.e.* $I(X;Y) = 0$), then, for a sufficiently large value of n, the value $2n \cdot \hat{I}(X;Y)$ is drawn from an approximate χ^2 distribution with $(\#X - 1) \cdot (\#Y - 1)$ degrees of freedom, *i.e.* $\hat{I}(X;Y)$ is drawn from a probability distribution with mean approximately

$$\frac{(\#X - 1) \cdot (\#Y - 1)}{2n}$$

and variance approximately

$$\frac{(\#X - 1) \cdot (\#Y - 1)}{2n^2}.$$

We now consider scenarios where one of X or Y is known, and the other has been estimated through sampling. Using Moddemeijer's and Brillinger's results, Chatzikokolakis et al. (2010) show the following:

Theorem 3.4 (distribution of the mutual information of related random variables, one known and one unknown)

When the random variable X is known, the random variable Y is unknown and the true mutual information of X and Y is non-zero (*i.e.* $I(X;Y) > 0$), then, for

a sufficiently large value of n, the value $\hat{I}(X;Y)$ is drawn from an approximate normal distribution with mean approximately

$$I(X;Y) + \frac{(\#X-1)\cdot(\#Y-1)}{2n} + \mathcal{O}\left(\frac{1}{n^2}\right)$$

and variance approximately

$$\frac{1}{n}\sum_{x\in\mathrm{dom}(P_X)} P_X(x) \cdot\left(\sum_{y\in\mathrm{dom}(\hat{P}_Y)}\frac{\hat{P}_{XY}(x,y)}{P_X(x)}\log_2^2\left(\frac{\hat{P}_{XY}(x,y)}{\hat{P}_Y(y)}\right)-\left(\sum_{y\in\mathrm{dom}(\hat{P}_Y)}\frac{\hat{P}_{XY}(x,y)}{P_X(x)}\log_2\left(\frac{\hat{P}_{XY}(x,y)}{\hat{P}_Y(y)}\right)\right)^2\right) + \mathcal{O}\left(\frac{1}{n^2}\right).$$

Theorem 3.5 (distribution of the mutual information of unrelated random variables, one known and one unknown)

When the random variable X is known, the random variable Y is unknown and the true mutual information of X and Y is zero (*i.e.* $I(X;Y) = 0$), then, for a sufficiently large value of n, the value $2n\cdot\hat{I}(X;Y)$ is drawn from an approximate χ^2 distribution with $(\#X-1)\cdot(\#Y-1)$ degrees of freedom, *i.e.* $\hat{I}(X;Y)$ is drawn from a probability distribution with mean approximately

$$\frac{(\#X-1)\cdot(\#Y-1)}{2n}$$

and variance approximately

$$\frac{(\#X-1)\cdot(\#Y-1)}{2n^2}.$$

The authors demonstrate that these results collectively provide a framework for quantifying the accuracy of any given mutual information estimate. For any combination

of known or estimated random variables X and Y, it is possible to estimate the mutual information between them and check whether this estimate is consistent with the true existence of mutual information by testing for goodness of fit with either the χ^2 or normal distributions, and, in cases where the estimate is found to fit the χ^2 distribution and not the normal distribution, use the confidence interval of the χ^2 distribution to place bounds on the accuracy of the estimate.

Chatzikokolakis et al. indicate that there are situations where their technique for estimating mutual information is inappropriate.

One of the largest drawbacks is the requirement for a large number of data points[1] n to produce meaningful statistics; this is necessary because the variance of the normal distributions in Theorems 3.1 and 3.4 is of order $\mathcal{O}(1/n)$ on decreasing powers of n, and this variance must be minimised in order to produce a sufficiently narrow confidence interval. The required number of data points is in all cases a function of the size of the sample spaces of X and Y (*i.e.* $\#X \cdot \#Y$), and, depending on the complexity of the system, it may not be possible to perform enough sampling of the system in a reasonable time to produce an estimate of any unknown random variables. Chatzikokolakis et al. argue that one advantage is that, notwithstanding the requirement for a large amount of data, the results become progressively more certain as n increases; *i.e.*, the estimated mutual information approaches the true mutual information as the number of data points tends to infinity.

The statistical methods employed by Chatzikokolakis et al. require that the outputs from the system for a given input are independent and identically distributed (i.i.d.): the output of the system must only depend on the provided input each time the system is sampled. No other factor may have a statistically significant impact on the output if it is not accounted for in the output of the system (and, as explained in the sample size discussion above, such factors will cause an exponential increase in the

[1] We use the term "data point" when referring to the secret and publicly-observable information gathered from a single execution rather than the more appropriate and widely-used "observation", to avoid confusion with other meanings of "observation" in this thesis.

required number of samples if they are treated as outputs). It is worth noting that it *is* permissible for external sources of randomness not to be accounted for in the output, provided that they do not have a statistically significant effect on the distribution on the possible outputs, *i.e.* they are truly random.

3.3.2 Estimating Min-Entropy Leakage

Chothia et al. (2014) present a method for calculating a confidence interval for min-entropy leakage estimates obtained via sampling of a probabilistic system.[2] They begin by defining the min-entropy leakage estimate in terms of the estimated joint probability distribution $\hat{P}_{XY}$ and the marginal probability distributions $\hat{P}_X$ and $\hat{P}_Y$ in the expected way:

$$\widehat{\mathcal{L}}_{XY} = \log_2 \sum_{y \in \hat{P}_Y} \max_{x \in \hat{P}_X} \hat{P}_{XY}(x,y) - \log_2 \max_{x \in \hat{P}_X} \sum_{y \in \hat{P}_Y} \hat{P}_{XY}(x,y).$$

They also define the *empirical joint frequency distribution* $\hat{s}$ and the *empirical input frequency distribution* $\hat{u}$ from $\hat{P}_{XY}$ and the sample size n:

$$\hat{s}(x,y) = \hat{P}_{XY}(x,y) \cdot n;$$
$$\hat{u}(x) = \sum_{y \in \hat{P}_Y} \hat{s}(x,y).$$

The authors then compute a confidence interval for the min-entropy leakage estimate from the confidence intervals of the min-entropy $H_\infty(X)$ (Definition 2.25, p. 39) and conditional min-entropy $H_\infty(X \mid Y)$ (Definition 2.27, p. 40) — recall from Definition 2.28 (p. 40) that min-entropy leakage itself is defined in terms of these two quantities. The computation of these confidence intervals requires the experimental discovery of various empirical frequency distributions that fit the true frequency

[2] The author of this thesis is a co-author of this publication, but did not contribute significantly to the min-entropy leakage estimation proofs; hence, they are presented here as previous work, rather than as a contribution of the author as part of a later chapter.

distributions with greater than 95% confidence; the authors therefore use Pearson's (1900) χ^2 tests to test for goodness of fit (below, $\chi^2_{\alpha,k}$ denotes the χ^2 test statistic with significance level α and k degrees of freedom).

Firstly, a 95% confidence interval for conditional min-entropy is computed as follows. From the empirical joint frequency distribution $\hat{s}$, find the joint frequency distribution $s_{\max}$ that maximises the conditional vulnerability (Definition 2.26, p. 39) amongst the joint frequency distributions that satisfy the following χ^2 test statistic:

$$\chi^2_{(0.05,\#X\cdot\#Y-1)} = \sum_{\substack{x\in\mathrm{dom}(X)\\ y\in\mathrm{dom}(Y)}} \frac{(\hat{s}(x,y)-s_{\max}(x,y))^2}{s_{\max}(x,y)}.$$

After finding $s_{\max}$, use it to compute the equivalent joint probability distribution $P^{\mathrm{post}}_{\max}$:

$$P^{\mathrm{post}}_{\max}(x,y) = \frac{s_{\max}(x,y)}{n}.$$

Next, from the empirical joint frequency distribution $\hat{s}$, find the joint frequency distribution $s_{\min}$ that minimises the conditional vulnerability amongst the joint frequency distributions that satisfy the following χ^2 test statistic:

$$\chi^2_{(0.05,\#X\cdot\#Y-1)} = \sum_{\substack{x\in\mathrm{dom}(X)\\ y\in\mathrm{dom}(Y)}} \frac{(\hat{s}(x,y)-s_{\min}(x,y))^2}{s_{\min}(x,y)}.$$

After finding $s_{\min}$, use it to compute the equivalent joint probability distribution $P^{\mathrm{post}}_{\min}$:

$$P^{\mathrm{post}}_{\min}(x,y) = \frac{s_{\min}(x,y)}{n}.$$

The lower and upper bounds for the 95% confidence interval for conditional min-

entropy can then be computed from $P^{\text{post}}_{\max}$ and $P^{\text{post}}_{\min}$:

$$H^{\text{low}}_{\infty}(X \mid Y) = -\log_2 \sum_{y \in \hat{P}_Y} \max_{x \in \hat{P}_X} P^{\text{post}}_{\max}(x, y);$$
$$H^{\text{up}}_{\infty}(X \mid Y) = -\log_2 \sum_{y \in \hat{P}_Y} \max_{x \in \hat{P}_X} P^{\text{post}}_{\min}(x, y).$$

Secondly, a 95% confidence interval for min-entropy is computed as follows. From the empirical input frequency distribution $\hat{u}$, find the input frequency distribution $u_{\max}$ that maximises the vulnerability (Definition 2.24, p. 39) amongst the input frequency distributions that satisfy the following χ^2 test statistic:

$$\chi^2_{(0.05,\#X-1)} = \sum_{x \in \text{dom}(X)} \frac{(\hat{u}(x) - u_{\max}(x))^2}{u_{\max}(x)}.$$

After finding $u_{\max}$, use it to compute the equivalent probability distribution $P^{\text{prior}}_{\max}$:

$$P^{\text{prior}}_{\max}(x) = \frac{u_{\max}(x)}{n}.$$

Next, from the empirical input frequency distribution $\hat{u}$, find the input frequency distribution $u_{\min}$ that minimises the vulnerability amongst the input frequency distributions that satisfy the following χ^2 test statistic:

$$\chi^2_{(0.05,\#X-1)} = \sum_{x \in \text{dom}(X)} \frac{(\hat{u}(x) - u_{\min}(x))^2}{u_{\min}(x)}.$$

After finding $u_{\min}$, use it to compute the equivalent probability distribution $P^{\text{prior}}_{\min}$:

$$P^{\text{prior}}_{\min}(x) = \frac{u_{\min}(x)}{n}.$$

The lower and upper bounds for the 95% confidence interval for min-entropy can then

be computed from $P^{\text{prior}}_{\max}$ and $P^{\text{prior}}_{\min}$:

$$H^{\text{low}}_{\infty}(X) = -\log_2 \max_{x \in \hat{P}_X} P^{\text{prior}}_{\max}(x);$$
$$H^{\text{up}}_{\infty}(X) = -\log_2 \max_{x \in \hat{P}_X} P^{\text{prior}}_{\min}(x).$$

Finally, the lower and upper bounds of the confidence intervals for conditional min-entropy and min-entropy can be used to derive bounds for a greater than 95% confidence interval for min-entropy leakage:

$$\widehat{\mathcal{L}}^{\text{low}}_{XY} = H^{\text{low}}_{\infty}(X) - H^{\text{up}}_{\infty}(X \mid Y);$$
$$\widehat{\mathcal{L}}^{\text{up}}_{XY} = H^{\text{up}}_{\infty}(X) - H^{\text{low}}_{\infty}(X \mid Y).$$

Chothia et al. find that, in practice, both the min-entropy leakage estimate and the confidence interval computed by this method have a higher variance than their mutual information counterparts; this is because Moddemeijer's (1989) and Brillinger's (2004) earlier work determines the precise distribution from which mutual information estimates are drawn and so a precise confidence interval can be computed, but this is not the case when estimating min-entropy leakage. The authors also experimentally identify a small negative bias in the min-entropy leakage estimate when compared with the true min-entropy leakage measure, but for the same reason cannot correct for it in the same way that Chatzikokolakis et al. correct their mutual information estimate. The authors indicate that the variance of the min-entropy leakage estimate depends greatly on the system being sampled: some systems, particularly those containing unique maximum probabilities in the estimated joint probability distribution $\hat{P}_{XY}$, generate samples that produce estimates with a wider variance with this method than others.

3.3.3 Summary

Our review of the literature in this section indicates that the estimation of information-theoretic measures is practical. The results of Moddemeijer (1989), Brillinger (2004) and Chatzikokolakis et al. (2010) show that excellent confidence intervals can be derived for mutual estimation estimates, and the results of Chothia et al. (2014) show that acceptable confidence intervals can be derived for min-entropy leakage estimates. This is a viable technique we can use to overcome the problem of being unable to analyse flows of information in complex probabilistic systems, provided that the amount of sampling required to produce the estimate relative to the amount of secret and publicly-observable information that occurs in the program is not prohibitively large. Clearly, this would also limit the types of system we would be able to analyse, but it would enable us to partially achieve our goal of analysing more complex systems.

3.4 Summary of the Literature

In this chapter, we have reviewed publications in both branches of the information flow analysis literature — qualitative and quantitative. We found that, while all of the information flow models we reviewed are sound, few of them are equipped to analyse complex or real-world systems: qualitative models either have security policies requiring compliance with excessively restrictive properties (*e.g.* noninterference), or give less satisfactory guarantees about the security of the system; quantitative models, while providing bounds on the amount of information leaked by a system, generally do not model the types of interaction between secret and publicly-observable information seen in typical systems (*e.g.*, they assume that secret or publicly-observable information only occurs before the system executes or after it terminates, when in reality both types of information occur throughout execution).

We have identified one possible solution: a new information flow model that disregards most of a system's behaviour and instead examines only the collections of secret

and publicly-observable information that occur during its execution, computing the discrete probability distributions on the collections of secret and publicly-observable information that occur. This model would be able to analyse more complex probabilistic systems and use the information-theoretic measures of mutual information and min-entropy leakage to answer questions of the form "what does an attacker learn about the secret information that occurs at all of these specific points during the program's execution by inspecting the publicly-observable values that occur at all of these other specific points?". However, this "point-to-point" information flow model would eventually encounter the same problem as all information flow models that measure information flow from discrete probability distributions: precisely computing these probability distributions becomes intractable when the program's state space becomes too large.

To solve this problem, we could estimate information leakage measures via sampling of the system; this would affect the accuracy of the measure, but would allow us to analyse more complex systems with larger state spaces. We have reviewed statistical results from the literature that derive confidence intervals for estimates of both mutual information and min-entropy leakage, allowing us to place bounds on the accuracy of the estimates (and, in the case of mutual information, apply a correction to the bias in the estimate). The main drawback is the large sample size required relative to the number of unique pieces of secret and publicly-observable information that occur for acceptable mutual information confidence intervals to be derived, a size that is even larger when deriving min-entropy leakage confidence intervals; however, provided that a sufficient sample can be assembled, good results can be achieved.

II

Computing Information Leakage from Programs

4

A Probabilistic Point-to-Point Information Flow Model

In our review of the quantitative information flow analysis literature in Chapter 3, we identified two particular weaknesses of existing information flow models. The first concerns the *amount* of secret or publicly-observable information the models permit: all of the models we reviewed restricted programs to either containing a single secret value, a single publicly-observable value, or both. The second concerns the *locations* where this information may occur: the models assume either that secret information is present before programs execute, that publicly-observable information occurs only when the program terminates, or both. We argued that neither of these restrictions are appropriate when modelling the behaviour of most typical programs — it is often the case that both types of information occur throughout the program's execution. In order to analyse information flows that occur in these programs, a new model is required that accurately represents how they store and process secret and publicly-observable information.

In this chapter, we present CH-IMP, a programming language with a probabilistic semantics; the execution of a CH-IMP program is formally defined in terms of a discrete-time Markov chain (Section 2.2, p. 30). A novel feature of CH-IMP is the addition of two commands, secret and observe, that record the occurrence of secret and

publicly-observable information respectively. The purpose of these commands is to identify that the values stored in certain variables at particular moments are "special": the secret V command indicates that the value stored in the variable V at this instant is secret, and the observe V command indicates that the value stored in V at this instant can be observed by the attacker. The use of either command is unrestricted — they may occur in complex structures such as loops and branches, or in blocks of code that are only executed with a small probability; the model quantifies the amount of information that flows from the secret values at certain arbitrary points to the publicly-observable values at other arbitrary points — hence a *point-to-point* information flow model. This could be considered a generalisation of Alvim et al.'s (2010) *interactive system* model, in which programs may contain interleaving secret and observable labels, but only in alternating order along each possible path of execution through the program.

The CH-IMP attacker model assumes that the behaviour of a CH-IMP program is known to a passive attacker (*e.g.*, it is assumed that the attacker knows the program's source code); because CH-IMP assigns values to variables probabilistically, and because secret information arises from variables at defined points, this implies that the attacker knows the *range* of possible secret values that could occur at each point, but cannot necessarily correlate particular secret values and particular publicly-observable values that arise from the program.

If the attacker's goal is simply to learn as much information as possible about the specific secret values that occur during execution, the Shannon entropy of the secret information is a good measure of the attacker's uncertainty about the program's secret information, and therefore the mutual information of the secret and observable information that occurs in the program is a good measure of the amount of information it leaks; if the attacker's goal is instead to correctly guess the program's secret values *en masse* in a single attempt, the min-entropy of the secret information is a better measure of the attacker's uncertainty about the program's secret information, and therefore the

min-entropy leakage from the secret information to the publicly-observable information is a more appropriate measure of the program's information leakage. In either case, the model does not constrain the selection of information leakage measure, and in fact *any* measure that is a function of the joint probability distribution of the secret and observable information can be computed, adding flexibility to CH-IMP's attacker model.

Initially, we present a version of CH-IMP *without* an information flow semantics; *i.e.*, we formally define the behaviour of all commands in the language except for secret and observe. We then consider a number of insecure CH-IMP programs, and contemplate how much secret information they leak; at the end of this process, we find that a particular semantics for secret and observe naturally arises, as well as formal definitions of "secret information" and "publicly-observable information". We then present a version of CH-IMP *with* an information flow semantics based on these intuitions, and show how the two information leakage measures we highlight can be derived from the secret and publicly-observable information that is recorded during execution of the program.

The contents of this chapter are based primarily on one of our contributions to the literature (Chothia et al., 2013b).

4.1 CH-IMP: A Probabilistic Language

We begin by defining the syntax of CH-IMP, a probabilistic language featuring arithmetic and Boolean expressions and the ability to store integer values in variables. It is syntactically similar to Winskel's (1993, Chapter 2) IMP, a non-probabilistic imperative language that forms part of an introduction to operational semantics, and is semantically similar to Kozen's (1981) probabilistic language (on which Mu and Clark (2009a) also base their own probabilistic language for information flow analysis).

The syntax of CH-IMP is defined as follows:

Definition 4.1 (CH-IMP grammar)

A CH-IMP program is a command C conforming to the grammar

$$
\begin{array}{rcl}
C & ::= & \text{new } V := \rho \\
 & | & V := \rho \\
 & | & \text{if } (B) \{ C \} \text{ else } \{ C \} \\
 & | & \text{while } (B) \{ C \} \\
 & | & C; C \\
 & | & \text{start; } C\text{; end} \\
 & | & \text{secret } V \\
 & | & \text{observe } V
\end{array}
\qquad
\begin{array}{rcl}
B & ::= & \text{true} \\
 & | & \text{false} \\
 & | & A == A \\
 & | & A < A \\
 & | & A > A \\
 & | & \text{not } B \\
 & | & B \text{ and } B \\
 & | & B \text{ or } B
\end{array}
$$

$$
\begin{array}{rcl}
\rho & ::= & \{ F \}
\end{array}
\qquad
\begin{array}{rcl}
A & ::= & n \\
 & | & V \\
 & | & A + A \\
 & | & A - A \\
 & | & A * A \\
 & | & A / A \\
 & | & A \text{ mod } A
\end{array}
$$

$$
\begin{array}{rcl}
F & ::= & A \to p \\
 & | & F, F
\end{array}
$$

where V ranges over the variable names $\mathbb{V}$, n ranges over the integers $\mathbb{Z}$, p is a probability (*i.e.* a real number such that $0 \leq p \leq 1$), and ρ is a discrete probability distribution (Definition 2.6, p. 26).

As is the case in other probabilistic languages, variables in CH-IMP are not assigned values using the declaration and assignment commands according to the evaluation of a single arithmetic expression, but are instead assigned values according to a discrete probability distribution ρ on arithmetic expressions; thus, the command

$$\text{new } i := \{ 0 \to 0.5, 1 \to 0.5 \}$$

declares a new variable i and assigns to it the value 0 with probability ½, 1 with probability ½, and all other integers with probability 0. For ρ to be a discrete probability distribution, the probabilities must sum to 1 (Definition 2.6, p. 26); we consider programs that do not meet this requirement to be badly-formed. To simplify the formal definition of variable scoping in Section 4.2 (p. 81), we also require that variables are declared with unique names for a CH-IMP program to be well-formed.

Although the grammar requires declarations and assignments to be made according to a probability distribution on arithmetic expressions, we will nevertheless use new $i := A$ as syntactic sugar for new $i := \{ A \to 1 \}$ for brevity.

4.2 A Basic Semantics for the CH-IMP Language

We shall now formalise the execution of a CH-IMP program *without* an information flow semantics — *i.e.* with the secret and observe commands omitted — in terms of a discrete-time Markov chain, following Definition 2.13 (p. 30).[1] This allows us to focus initially on defining the behaviour of the core language. The secret and observe commands will be motivated and formally defined later, and thus provide a full information flow semantics for CH-IMP.

We begin by defining three operations on sequences and functions.

Definition 4.2 (function and sequence operations)

Sequence partition. $i_0 :: I_r$ partitions a sequence $I = \langle i_0, i_1, \ldots, i_n \rangle$ into its *first* element i_0 and another sequence consisting of its remaining elements $I_r = \langle i_1, \ldots, i_n \rangle$; $I_r :: i_n$ partitions I into its *last* element i_n and another sequence consisting of its remaining elements $I_r = \langle i_0, \ldots, i_{n-1} \rangle$.

Function comprehension. $f_p \cup f_q$ produces a new function f by merging the functions f_p and f_q; if i appears in the domain of both functions, $f(i) = f_p(i)$.

Function sequence remapping. $f \oplus \langle f_0, f_1, \ldots, f_n \rangle$ produces a new sequence of functions $\langle f_0', f_1', \ldots, f_n' \rangle$, where f_i' is defined as follows for all i functions

[1] The definition of information flow that we provide later in this chapter can only be computed if each path in the DTMC induced by a CH-IMP program's execution ends in an accepting state; therefore, we shall assume that CH-IMP programs always terminate.

in the new sequence:

$$f_i' = \begin{cases} f \cup f_i & \text{if } \mathrm{dom}(f) \subset \mathrm{dom}(f_i); \\ f_i & \text{otherwise.} \end{cases}$$

These operators simplify the semantic rules governing the behaviour of CH-IMP commands: as we shall see shortly, the sequence partition operator will be used to both insert and remove elements in sequences between states of a CH-IMP program, and the function comprehension and function sequence operators simplify the formal definitions of variable scoping.

The semantic rules for evaluating the arithmetic and Boolean expressions in Definition 4.1 are shown in Figure 4.1. Variable scope is maintained using a finite sequence σ of *scope frames* o, functions mapping variable names to values; the narrowest scope frame is the first element in the sequence, and the global scope frame is the last. The operators are conventional, and their semantics are unsurprising; the only pitfall is the division operator, which returns the integer quotient of its operands (since all variable values and evaluations of arithmetic expressions in CH-IMP are integers).

Using the rules from Figure 4.1 and the definition of a DTMC from Definition 2.13, we define the semantics of the version of CH-IMP without the secret and observe commands as follows.

Definition 4.3 (CH-IMP execution without information flow semantics)
The execution of a CH-IMP program $\mathcal{P}$ *without an information flow semantics* is formalised as a discrete-time Markov chain (DTMC) $\mathcal{D} = (S, \bar{s}, \mathbf{P})$ with states $S : C \times \sigma$, where

- C is a finite sequence of commands to be executed (*i.e.* an expression derived from the grammar in Definition 4.1 (p. 80), except for secret V and observe V),

Figure 4.1: the semantics of arithmetic and Boolean expressions in CH-IMP

$$\textit{ConstInt}\ \frac{}{[\![\, n \,]\!](\sigma) \to n}$$

$$\textit{ConstTrue}\ \frac{}{[\![\, \mathsf{true} \,]\!](\sigma) \to \mathsf{true}} \qquad \textit{ConstFalse}\ \frac{}{[\![\, \mathsf{false} \,]\!](\sigma) \to \mathsf{false}}$$

$$\textit{VarInScope}\ \frac{V \in \mathrm{dom}(o)}{[\![\, V \,]\!](o :: \sigma) \to [\![\, V \,]\!](o)} \qquad \textit{VarNotInScope}\ \frac{V \notin \mathrm{dom}(o)}{[\![\, V \,]\!](o :: \sigma) \to [\![\, V \,]\!](\sigma)}$$

$$\textit{Add}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2}{[\![\, A_1 + A_2 \,]\!](\sigma) \to n_1 + n_2} \qquad \textit{Mult}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2}{[\![\, A_1 * A_2 \,]\!](\sigma) \to n_1 \times n_2}$$

$$\textit{Sub}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2}{[\![\, A_1 - A_2 \,]\!](\sigma) \to n_1 - n_2} \qquad \textit{Div}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2}{[\![\, A_1 / A_2 \,]\!](\sigma) \to \lfloor n_1/n_2 \rfloor}$$

$$\textit{Mod}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2}{[\![\, A_1\ \mathsf{mod}\ A_2 \,]\!](\sigma) \to n_1\ \mathrm{mod}\ n_2}$$

$$\textit{EquTrue}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2 \qquad n_1 = n_2}{[\![\, A_1 == A_2 \,]\!](\sigma) \to \mathsf{true}}$$

$$\textit{EquFalse}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2 \qquad n_1 \neq n_2}{[\![\, A_1 == A_2 \,]\!](\sigma) \to \mathsf{false}}$$

$$\textit{LessTrue}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2 \qquad n_1 < n_2}{[\![\, A_1 < A_2 \,]\!](\sigma) \to \mathsf{true}}$$

$$\textit{LessFalse}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2 \qquad n_1 \not< n_2}{[\![\, A_1 < A_2 \,]\!](\sigma) \to \mathsf{false}}$$

$$\textit{GreatTrue}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2 \qquad n_1 > n_2}{[\![\, A_1 > A_2 \,]\!](\sigma) \to \mathsf{true}}$$

$$\textit{GreatFalse}\ \frac{[\![\, A_1 \,]\!](\sigma) \to n_1 \qquad [\![\, A_2 \,]\!](\sigma) \to n_2 \qquad n_1 \not> n_2}{[\![\, A_1 > A_2 \,]\!](\sigma) \to \mathsf{false}}$$

$$\textit{And}\ \frac{[\![\, B_1 \,]\!](\sigma) \to t_1 \qquad [\![\, B_2 \,]\!](\sigma) \to t_2}{[\![\, B_1\ \mathsf{and}\ B_2 \,]\!](\sigma) \to t_1 \wedge t_2} \qquad \textit{Or}\ \frac{[\![\, B_1 \,]\!](\sigma) \to t_1 \qquad [\![\, B_2 \,]\!](\sigma) \to t_2}{[\![\, B_1\ \mathsf{or}\ B_2 \,]\!](\sigma) \to t_1 \vee t_2}$$

$$\textit{Not}\ \frac{[\![\, B \,]\!](\sigma) \to t}{[\![\, \mathsf{not}\ B \,]\!](\sigma) \to \neg t}$$

- $\sigma : \mathbb{N} \to o$ is a finite sequence of variable scope frames (with the narrowest scope frame at the start of the sequence), and
- a variable scope frame $o : \mathbb{V} \to \mathbb{Z}$ is a function mapping variable names to values;

the initial state $\bar{s} = (\mathcal{P}, \langle\{\}\rangle)$; and the transition probability matrix $\mathbf{P}$ defined according to the following semantic rules:

$$\textit{Decl}\ \frac{}{(\mathsf{new}\ V := \rho;\ C, o :: \sigma) \xrightarrow{\rho(n)} (C, (\{V \to n\} \cup o) :: \sigma)}$$

$$\textit{Assign}\ \frac{}{(V := \rho;\ C, \sigma) \xrightarrow{\rho(n)} (C, \{V \to n\} \oplus \sigma)}$$

$$\textit{IfTrue}\ \frac{[\![\ B\ ]\!](\sigma) \to \mathrm{true}}{(\mathsf{if}\ (B)\ \{\ C_T\ \}\ \mathsf{else}\ \{\ C_F\ \};\ C, \sigma) \xrightarrow{1} (\mathsf{start};\ C_T;\ \mathsf{end};\ C, \sigma)}$$

$$\textit{IfFalse}\ \frac{[\![\ B\ ]\!](\sigma) \to \mathrm{false}}{(\mathsf{if}\ (B)\ \{\ C_T\ \}\ \mathsf{else}\ \{\ C_F\ \};\ C, \sigma) \xrightarrow{1} (\mathsf{start};\ C_F;\ \mathsf{end};\ C, \sigma)}$$

$$\textit{WhileTrue}\ \frac{[\![\ B\ ]\!](\sigma) \to \mathrm{true}}{(\mathsf{while}\ (B)\ \{\ C_W\ \};\ C, \sigma) \xrightarrow{1} (\mathsf{start};\ C_W;\ \mathsf{end};\ \mathsf{while}\ (B)\ \{\ C_W\ \};\ C, \sigma)}$$

$$\textit{WhileFalse}\ \frac{[\![\ B\ ]\!](\sigma) \to \mathrm{false}}{(\mathsf{while}\ (B)\ \{\ C_W\ \};\ C, \sigma) \xrightarrow{1} (C, \sigma)}$$

$$\textit{ScopeIn}\ \frac{}{(\mathsf{start};\ C, \sigma) \xrightarrow{1} (C, \{\} :: \sigma)} \qquad \textit{ScopeOut}\ \frac{}{(\mathsf{end};\ C, o :: \sigma) \xrightarrow{1} (C, \sigma)}$$

$$\textit{Accept}\ \frac{}{(\langle\rangle, \sigma) \xrightarrow{1} (\langle\rangle, \sigma)}$$

The sequence σ in the tuple defining a CH-IMP program's state is responsible for storing the variables that are in scope at any given moment during execution, along with the values they have been assigned. In the initial state $\bar{s}$, this sequence contains a single empty scope frame, *i.e.* a function whose domain is the empty set. This scope

frame represents the program's global scope; it will never be destroyed. The start and end commands maintain the program's variable scope per the *ScopeIn* and *ScopeOut* rules. These commands are not to be used directly: they are inserted into the list of commands C by the *IfTrue*, *IfFalse* and *WhileTrue* rules to create a new, empty narrowest scope frame and destroy the current narrowest scope frame in σ respectively. This ensures that variables declared in C_T, C_F and C_W go out of scope when their commands have finished executing. Although variable scoping is not a strictly necessary feature of the language (since any CH-IMP program could be rewritten using only globally-scoped variables), it minimises the number of bound variables in any given state of a program, thus minimising the time complexity of a test for the equality of two instances of the sequence σ. We shall see why this is valuable when we describe our implementation of the semantics of CH-IMP in Chapter 5.

The *Decl* and *Assign* rules define the behaviour of variable declaration and assignment. A new variable V is declared with a name and a probability distribution ρ on its value; for each integer n, with probability $\rho(n)$ the DTMC transitions into a new state where the narrowest scope frame in σ contains a new mapping from the variable name to the value n. The *Assign* rule is similar, but replaces the mapping for V in the scope frame in which it was declared rather than inserting a new mapping for V into the narrowest scope frame. *Decl* and *Assign* are the only rules whose behaviour is probabilistic: all other rules cause the DTMC to transition into a single succeeding state with probability 1.

The remaining rules cause state transitions in a typical manner: *IfTrue* evaluates an if command if its guard — a Boolean expression — evaluates to true according to the rules in Figure 4.1, thus executing C_T; otherwise, the command is evaluated with the *IfFalse* rule, thus executing C_F. If the guard in a while command evaluates to true, the body of the while command C_W is executed once and the guard is re-evaluated; if it evaluates to false, execution continues from C. Finally, the *Accept* rule defines the DTMC's behaviour when there are no more commands remaining to be

executed: to satisfy the formal definition of a DTMC's probability transition matrix (Definition 2.13, p. 30), it enters an accepting state, *i.e.* a self-transition with probability 1.

4.3 Toward An Information Flow Semantics for CH-IMP

We must complete the definition of CH-IMP by defining the behaviour of the secret and observe commands; in doing so, we shall define its information flow model.

Recall from the introduction to this chapter (p. 77) our intuitions about these two commands: secret V indicates that the value of V at this point should be considered a secret, and observe V indicates that an attacker will be able to observe the value of V at this point. In both cases, we do not concern ourselves with the value of V before or after the command executes, even though its value may not change: we are interested in the value stored inside a particular memory cell at a given moment, rather than the memory cell itself. Additionally, we do not wish to restrict occurrences of either command to particular locations within the program: it should be possible for both secret and observe to occur any number of times anywhere in the program, including inside branches of if commands and bodies of while loops. Permitting both secret and observable information to occur at arbitrary points in programs complicates the information flow model: unlike in previous work, where the same secret information exists throughout the execution of a program and the publicly-observable information is present upon termination, it may not be immediately clear when — or even whether — secret and observable information occurs, and how the amounts of each type of information should be quantified.

CH-IMP's attacker model assumes that the attacker (*a*) is passive, (*b*) has prior knowledge of the *behaviour* of the program (although not necessarily the specific path of execution it takes), and (*c*) has the ability, after termination, to observe the values of all variables that were operands of an observe command at the points at which

the observe commands were evaluated. Note that, since CH-IMP is a probabilistic language, prior knowledge of the program's behaviour confers knowledge about the *range* of values that could be assigned to a particular variable at the point at which its value is marked as secret, but does not guarantee that the attacker is able to correlate the occurrence of particular observable values with the occurrence of particular secret values. However, if the program is insecure, the observable information revealed to the attacker potentially narrows the range of possible values that could be assigned to variables whose values are marked as secret; this is therefore the source of information leakage in a CH-IMP program.

If the attacker's goal is simply to learn as much information as possible about the secret values that occur during execution, the Shannon entropy (Definition 2.18, p. 33) of the secrets is a good measure of the attacker's uncertainty about those secrets; this suggests that the mutual information (Definition 2.21, p. 35) of the secret and observable values that occur during execution is a good measure of the amount of information leaked by the program. (In the examples that follow in this section, we shall use Shannon entropy and mutual information as uncertainty and information leakage measures respectively to motivate our information flow model.) However, many other leakage measures can be computed using the same information about the program's execution, and different ones can be chosen to model attackers with different capabilities and goals; for instance, if the attacker's goal is instead to correctly guess the program's secret values *en masse* in a single attempt, the min-entropy (Definition 2.25, p. 39) of the secret values is a better measure of the attacker's uncertainty, and therefore the min-entropy leakage (Definition 2.28, p. 40) from the secret values to the observable values is a more appropriate measure of the program's information leakage.

To gain further intuition about the desired behaviour of secret and observe, we now consider six CH-IMP programs that contain secret and observable information; we argue that each example contains an information leak, reason about the attacker's knowledge of the secret information, quantify each leak, and use the example to inform our

formal definition of CH-IMP with an information flow semantics.

4.3.1 Reversing a Bitwise XOR Operation

Listing 4.1 shows a simple CH-IMP program in which a bitwise XOR operation occurs. The value of *rand* at the point of declaration on line 1 is chosen from a uniform probability distribution, as is the value of *sec* on line 3. The value of a third variable on line 5, *out*, is the bitwise XOR of these two values.

The program produces two values that may be observed by an attacker, indicated by the use of the observe command: the value of *rand* on line 2, and the value of *out* on line 6. The use of the secret command on line 4 indicates that the value of *sec* is sensitive at that particular point; its value on line 4 is 0 or 1 with equal probability, so there is 1 bit of Shannon entropy in its value and therefore 1 bit of secret information present.

Table 4.1 shows the joint probability distributions of various subsets of secret values and observable values that occur in this program. The mutual information of *sec* and *rand* (Table 4.1(a)) is 0 bits: the attacker cannot distinguish between the two values of *sec* by observing *rand* alone. The same is true for *sec* and *out* (Table 4.1(b)). However, information *is* shared between the value of *sec* and the values of *rand* and *out* collectively (Table 4.1(c)): the attacker is able to reverse the xor operation on line 5 by observing both of these values, since bitwise XOR is a reversible function given the function's output and one of its inputs. The mutual information in this case is 1 bit, which represents a reduction in entropy of *sec*'s value by 1 bit and thus a complete

Listing 4.1: a CH-IMP example in which an attacker observes the output and half of the input of a bitwise XOR operation; the second half of the input is secret

```
1  new rand := { 0 → 0.5, 1 → 0.5 };
2  observe rand;
3  new sec := { 0 → 0.5, 1 → 0.5 };
4  secret sec;
5  new out := sec xor rand;
6  observe out
```

leakage of the program's secret information.

To detect information leaks such as the one described above, the "observable information" that occurs in a program should therefore be defined as the collection of *all* values that are observed along a particular path of execution, including those that are observed before other values are marked as secret.

4.3.2 Equality of Secrets

Another bitwise XOR operation occurs in the program in Listing 4.2 (p. 90). This time, there are two pieces of secret information: on line 3 the value of *sec1* (which at this point is 0 or 1 with equal probability) and on line 4 the value of *sec2* (which, again, is 0 or 1 with equal probability) are both marked as secret; there are therefore four possible combinations for the values of *sec1* and *sec2* at these points. These combinations all occur with the same probability; there are therefore 2 bits of Shannon entropy in the program's secret information.

An attacker may observe a single value in this program: the value of *out* on line 6,

Table 4.1: three possible joint probability distributions of secret values and observable values for the program shown in Listing 4.1

(a) *sec* and *rand*

	rand	
sec	0	1
0	0.25	0.25
1	0.25	0.25

(b) *sec* and *out*

	out	
sec	0	1
0	0.25	0.25
1	0.25	0.25

(c) *sec*, and *rand* and *out* collectively

	rand, *out*			
sec	0, 0	0, 1	1, 0	1, 1
0	0.25	0	0	0.25
1	0	0.25	0.25	0

which is the bitwise XOR of the values of *sec1* and *sec2* on line 5.

As before, Table 4.2 shows the joint probability distributions of various subsets of the secret values and observable values that occur. An attacker with the ability to observe *out* is unable to directly distinguish between the two possible values of *sec1* (Table 4.2(a)) or *sec2* (Table 4.2(b)). The mutual information of the values of *sec1* and *out*, and of *sec2* and *out*, is therefore 0 bits. However, by observing the value of *out*, the attacker learns *some* information about the values of *sec1* and *sec2*: because the XOR of any integer with itself equals 0, the attacker learns whether the values of *sec1* and *sec2* are equal. This narrows down the combinations of the values of *sec1* and

Listing 4.2: a CH-IMP example in which an attacker observes the bitwise XOR of two secrets

```
1 new sec1 := { 0 → 0.5, 1 → 0.5 };
2 new sec2 := { 0 → 0.5, 1 → 0.5 };
3 secret sec1;
4 secret sec2;
5 new out := sec1 xor sec2;
6 observe out
```

Table 4.2: three possible joint probability distributions of secret values and observable values for the program shown in Listing 4.2

(a) *sec1* and *out*

	out	
sec1	0	1
0	0.25	0.25
1	0.25	0.25

(b) *sec2* and *out*

	out	
sec2	0	1
0	0.25	0.25
1	0.25	0.25

(c) *sec1* and *sec2* collectively, and *out*

	out	
sec1, *sec2*	0	1
0, 0	0.25	0
0, 1	0	0.25
1, 0	0	0.25
1, 1	0.25	0

sec2 from four to two, the precise combinations depending on the observation that the attacker makes (Table 4.2(c)). This equates to a reduction of 1 bit in the entropy of the program's secret information, and there is therefore 1 bit of mutual information between it and the program's observable information.

This example is a counterpart to the program in Listing 4.1, but for secret information: it demonstrates that to detect such information leaks, the "secret information" that occurs in a program should be defined as the collection of *all* values that are marked as secret along a particular path of execution, including those that are marked as secret before other values are observed.

4.3.3 Secret Information inside a Loop

So far, we have considered only secret and publicly-observable information that occurs in the global scope frame. Listing 4.3 (p. 92) shows a more complex program, where secret and observe commands occur inside a while loop whose body is executed three times. A new value — an integer chosen uniformly from the interval $[0, 2]$ — is assigned to the variable *sec* once on every iteration of the loop, on line 6; therefore, from Definition 2.18 (p. 33), there are $-3 \cdot 1/3 \log_2 1/3 \approx 1.58$ bits of Shannon entropy in each value of *sec*. Each value is marked as a secret on line 7. The three assignments to *sec* happen unconditionally and the probability distribution on the possible values remains constant, so there is no information shared between the values of *sec* that may occur during execution of the program; consequently, there are $3 \cdot (-3 \cdot 1/3 \log_2 1/3) \approx 4.75$ bits of secret information in this program.

There is also an observe command in the loop body, on line 5; an attacker may therefore observe three values of the variable *out*, one during each iteration of the loop. The value of *out* is always 0 at the first and second points at which it is observed; however, an if command that evaluates to true in the second iteration of the loop causes the execution of code that copies the current value of *sec* into *out*. Therefore, on the final iteration of the loop, the value of *out* entirely leaks the value of *sec* that was

assigned in the previous iteration.

So far, we have established that a program's "secret information" and "observable information" should be defined as all of the secret and observable values that are encountered along a particular path of execution respectively; now that we are considering more complex flows of information, we must be more precise about these definitions.

Table 4.3 shows the joint probability distributions of various subsets of the secret values and observable values that occur in the program in Listing 4.3. By limiting the domains of the joint probability distributions to the values that occur inside a single iteration of a while loop (Table 4.3(a)), the program does not appear to leak information: because the value of *out* is unrelated to that of *sec* in any given iteration, each value of *out* is equally likely to occur given a particular value of *sec*, and the mutual information of *sec* and *out* is 0 bits. However, when considering the *entire* collection of values previously marked as secret on line 7 and the *entire* collection of values previously marked as observable on line 5 (*i.e.* the secret and observable information that occurs over all iterations collectively, as in Table 4.3(b)), the joint probability distribution is no longer uniform: there is a correlation between the second value of *sec* that is marked as secret and the third value of *out* that is marked as observable, which corresponds with the leak that occurs on line 9 during the second iteration of the loop.

Listing 4.3: a CH-IMP example containing a while loop with secret and publicly-observable information in its body

```
1  new i := 0;
2  new out := 0;
3  new sec := 0;
4  while (i < 3) {
5      observe out;
6      sec := { 0 → 1/3, 1 → 1/3, 2 → 1/3 };
7      secret sec;
8      if (i == 1) {
9          out := sec
10     };
11     i := i + 1
12 }
```

Table 4.3: two possible joint probability distributions of secret values and observable values for the program shown in Listing 4.3

(a) *sec* and *out* in each iteration of the while loop

	out		
sec	0	1	2
0	1/9	1/9	1/9
1	1/9	1/9	1/9
2	1/9	1/9	1/9

(b) *sec* and *out* across all iterations of the while loop

	out		
sec	0, 0, 0	0, 0, 1	0, 0, 2
0, 0, 0	1/27	0	0
0, 0, 1	1/27	0	0
0, 0, 2	1/27	0	0
0, 1, 0	0	1/27	0
0, 1, 1	0	1/27	0
0, 1, 2	0	1/27	0
0, 2, 0	0	0	1/27
0, 2, 1	0	0	1/27
0, 2, 2	0	0	1/27
1, 0, 0	1/27	0	0
1, 0, 1	1/27	0	0
1, 0, 2	1/27	0	0
1, 1, 0	0	1/27	0
1, 1, 1	0	1/27	0
1, 1, 2	0	1/27	0
1, 2, 0	0	0	1/27
1, 2, 1	0	0	1/27
1, 2, 2	0	0	1/27
2, 0, 0	1/27	0	0
2, 0, 1	1/27	0	0
2, 0, 2	1/27	0	0
2, 1, 0	0	1/27	0
2, 1, 1	0	1/27	0
2, 1, 2	0	1/27	0
2, 2, 0	0	0	1/27
2, 2, 1	0	0	1/27
2, 2, 2	0	0	1/27

Here, the mutual information of *sec* and *out* is approximately 1.58 bits, or the Shannon entropy of the second value of *sec* that is leaked by the program.

As with the example in Listing 4.2, this example demonstrates that, because information leaks may occur across iterations of a loop, the "secret information" and "observable information" that occurs in a program should be defined as the collections of *all* values marked as secret and observable respectively along a particular path of execution, including those stored in variables defined in non-global scope frames and after the variables that once stored their values are destroyed.

4.3.4 Different Secret Information in Different Paths of Execution

The previous examples only consider situations in which the program processes secret information stored inside a single variable, or in which the secret information arises from a sequence of variables that is identical along each path of execution of the program. We have not considered information leaks that occur when the path of execution taken determines either the number of variables whose values are treated as secret information, the variables from which secret information arises, or the order in which the values are marked as secret information — or, indeed, whether these situations should be considered information leaks at all.

Some programs may only process secret information under certain conditions. The true branch of the if command in the example shown in Listing 4.4 is executed with probability $1/2$, and marks the value of the variable *sec* as secret and then immediately

Listing 4.4: a CH-IMP example that may contain secret information, depending on which branch of an if command is executed

```
1  new rand := { 0 → 0.5, 1 → 0.5 };
2  if (rand == 1) {
3      new sec := { 0 → 0.5, 1 → 0.5 };
4      secret sec;
5      observe sec
6  } else {
7      observe rand
8  }
```

discloses it to the attacker (on lines 4 and 5 respectively). The false branch contains no occurrences of secret information, and instead always discloses the value 0 on line 7. Thus, the program may or may not contain secret information depending on which path of execution is followed.

Table 4.4 (p. 95) shows two possible joint probability distributions of the secret and observable values that could be used to quantify the information leakage from this program. By only considering paths of execution containing secret information (Table 4.4(a)), we conclude that the program contains 1 bit of secret information (*i.e.* the value of *sec* on line 4) and there is 1 bit of mutual information between it and the program's observable values, resulting in a total leakage of the program's secret information. However, this is an overstatement of the amount of information the program leaks: an attacker observing the value 0 in this program cannot be sure whether it was caused by the true branch being executed and the value of *sec* being 0 (an event that occurs with probability $1/3$ when the attacker observes the value 0), or the false

Table 4.4: two possible joint probability distributions of secret values and observable values for the program shown in Listing 4.4

(a) the secret and observable values that occurred along paths of execution containing secret information

	Observable values	
Secret values	0	1
$sec = 0$	1	0
$sec = 1$	0	1

(b) the secret and observable values that occurred along all paths of execution; $\perp$ denotes that no secret values occurred

	Observable values	
Secret values	0	1
$sec = 0$	0.25	0
$sec = 1$	0	0.25
$\perp$	0.5	0

branch being executed and no secret information occurring at all (an event that occurs with probability ⅔ when the attacker observes the value 0); these events are not distinguishable from each other in Table 4.4(a), which assumes that an attacker observing the value 0 *always* learns the value of *sec*. To make them distinguishable and thus avoid the overstatement, we must consider *all* paths of execution through the program, and not just those containing secret information; by doing this (Table 4.4(b)), we instead find from Definition 2.18 (p. 33) that there are 1.5 bits of secret information processed by the program, and from Definition 2.21 (p. 35) that there are approximately 0.81 bits of information shared between it and the observable information.

Some programs may process different types of secret information depending on the path of execution taken through the program; Listing 4.5 contains an example of this scenario. The true and false branches of the if command beginning on line 5 each declare a variable and mark the occurrence of secret information in that variable, but the variables have different names in each branch (and, given that CH-IMP variable names must be unique at declaration, are therefore different pieces of secret information); the branch that is executed depends on the value of *rand* that is assigned on line 1. An attacker is able to observe the value of *rand* on line 2 and, because *rand*'s value does not change between this observation and the evaluation of the Boolean expression in the if command on line 5, also learns which branch of the if command is executed. The attacker does not directly learn any other information about either of the variables

Listing 4.5: a CH-IMP example containing different secret information in each branch of an if command

```
1  new rand := { 0 → 0.5, 1 → 0.5 };
2  observe rand;
3  new sec1 := { 0 → 0.5, 1 → 0.5 };
4  new sec2 := { 0 → 0.5, 1 → 0.5 };
5  if (rand == 0) {
6      secret sec1
7  } else {
8      secret sec2
9  }
```

whose values are marked as secret (*sec1* on line 6, and *sec2* on line 8).

Table 4.5 shows some possible joint probability distributions of the secret and observable information in the program given in Listing 4.5. Table 4.5(a) shows the joint probability distribution that occurs when only the *values* of variables marked with the secret command are treated as secret information; from this distribution, we see that the program contains 1 bit of secret information (*i.e.* whether the secret value that occurred was either 0 or 1), and that the observable information reveals nothing about this secret information: the value of *rand* reveals nothing to the attacker about the value of either *sec1* or *sec2*. However, this is an understatement of what the attacker learns: although the program reveals nothing about the *value* of either variable, it does reveal the *name* of the variable that contained the value that was marked as secret, and so the attacker learns which piece of secret information was processed by the program. To correct this understatement, we must also consider the names of variables from which the secret values arise to be secret information in their own right by including

Table 4.5: two possible joint probability distributions of secret values and observable values for the program shown in Listing 4.5

(a) the secret values that occurred in each if branch, and *rand*

	Observable values	
Secret values	0	1
0	0.25	0.25
1	0.25	0.25

(b) all secret values that occurred (including the names of the variables containing their values), and *rand*

	Observable values	
Secret values	0	1
sec1 = 0	0.25	0
sec1 = 1	0.25	0
sec2 = 0	0	0.25
sec2 = 1	0	0.25

them in the joint probability distribution (Table 4.5(b)): now we see that, although the attacker cannot distinguish between situations where the secret value equals 0 or 1, the attacker *can* distinguish between situations where the secret value arose from the variable *sec1* or *sec2*. This introduces an extra 1 bit of secret information into the program, and from Definition 2.21 (p. 35) we find that there is 1 bit of information shared between it and the observable information (specifically, the name of the variable that stored the secret value); the understatement of the program's information leakage is therefore corrected.

4.4 An Information Flow Semantics for the CH-IMP Language

In Section 4.3, we provided five examples of CH-IMP programs that we consider to leak information, listed plausible joint probability distributions of the secret and observable information for each program, and argued which best quantified the information leakage from the program (using mutual information as our leakage measure). In this way, we provided intuitions about the nature of a program's "secret information" and "observable information" and the intended behaviour of CH-IMP's secret and observe commands. To summarise:

(a) A program's "observable information" is the collection of *all* values that are observed along a particular path of execution of the program, including those that are observed before other values are marked as secret (Section 4.3.1, p. 88).

(b) A program's "secret information" is the collection of *all* values that are marked as secret along a particular path of execution of the program, including those that are marked as secret before other values are observed (Section 4.3.2, p. 89).

(c) To quantify information flows from secret to observable values that occur as a result of executing control flow commands, a program's "secret information" and

"observable information" are the collections of *all* secret and observable values respectively along a particular path of execution of the program, including those stored in variables defined in non-global scope frames and after the variables that once stored their values are destroyed (Section 4.3.3, p. 91).

(d) To avoid overstating information flows from secret to observable values when the branching behaviour of a program causes a varying number of values to be marked as secret, *all* paths of execution through the program should be factored into the joint probability distribution, rather than just those containing a specific number of secret values (Section 4.3.4, p. 94).

(e) To avoid understating information flows from secret to observable values when the branching behaviour of a program causes secret information to arise from different variables, the program's "secret information" should include the names of variables whose values are marked as secret (Section 4.3.4, p. 94).

We now formalise these intuitions by providing a concrete definition for a program's "secret information" (which we denote with $\mathcal{S}$) and its "observable information" (which we denote with $\mathcal{O}$).

4.4.1 Formal Definitions for Secret and Observable Information

In Section 4.3.2 we informally defined a program's "secret information" as the collection of *all* secrets that occur along a particular path of execution, regardless of whether they occurred before other observable values. This naturally gives rise to the following definition of secret information:

Definition 4.4 (secret information in a CH-IMP program)

A program's *secret information* $\mathcal{S}$ is a finite sequence of variable name/value tuples that occur along a path of execution:

$$\mathcal{S} : \mathbb{N} \to (\mathbb{V} \times \mathbb{Z}).$$

Similarly, in Section 4.3.3 we emphasised the need to collect *all* secret values that occur in a CH-IMP program, regardless of whether the variables that stored those values are still in scope. The same principle is applied when formally defining a program's "observable information", per the informal definition in Section 4.3.1, with one important difference. The second example in Section 4.3.4 demonstrates the importance of tracking variable names associated with particular secret values to avoid understating information flows; it is not necessary to treat observable values in the same way, and we argue that it would be unrealistic to do so: real-world programs simply output information (*e.g.* by displaying the final four digits of a credit card number on a web page) without explicitly revealing anything about how that information was stored (*e.g.* the memory address where the final four digits of the credit card number were stored just before they were displayed). Thus, our attacker model also assumes that the attacker is able to inspect observable values that occur in a program, but *not* the names of the variables from which they arise:

Definition 4.5 (observable information in a CH-IMP program)

A program's *observable information* $\mathcal{O}$ is a finite sequence of variable values that occur along a path of execution:

$$\mathcal{O} : \mathbb{N} \to \mathbb{Z}.$$

Note that Definition 4.4 models an attacker with the ability to distinguish between scenarios where the same secret values arise from the same variables, but in a different order. This is demonstrated by the example in Listing 4.6: the variables *sec1* and *sec2*

are marked as containing secret information in a different order in each branch of an if command, and the attacker is able to observe the value of *rand*, which determines which branch of the if command is executed. Using the above definitions of "secret information" and "observable information", we obtain the joint probability distribution shown in Table 4.6; it shows that the program contains $\log_2(8) = 3$ bits of secret information and that, when the value of *rand* is disclosed to the attacker, the secret and observable values contain 1 bit of mutual information. Thus, the attacker learns which of the two variables *sec1* and *sec2* had its value marked as secret first, but does not learn anything about either of the variables' values.

Listing 4.6: a CH-IMP example where secret information occurs in a different order in each branch of an if command

```
1  new rand := { 0 → 0.5, 1 → 0.5 };
2  observe rand;
3  new sec1 := { 0 → 0.5, 1 → 0.5 };
4  new sec2 := { 0 → 0.5, 1 → 0.5 };
5  if (rand == 0) {
6      secret sec1;
7      secret sec2
8  } else {
9      secret sec2;
10     secret sec1
11 }
```

Table 4.6: the joint probability distribution of secret values and observable values for the program shown in Listing 4.6, given the definition of "secret information" in Definition 4.4

	Observable values	
Secret values	0	1
$sec1 = 0, sec2 = 0$	1/8	0
$sec1 = 0, sec2 = 1$	1/8	0
$sec1 = 1, sec2 = 0$	1/8	0
$sec1 = 1, sec2 = 1$	1/8	0
$sec2 = 0, sec1 = 0$	0	1/8
$sec2 = 0, sec1 = 1$	0	1/8
$sec2 = 1, sec1 = 0$	0	1/8
$sec2 = 1, sec1 = 1$	0	1/8

This is a strong attacker model, and it could be argued that it is *too* strong for simple programs such as the one in Listing 4.6. However, we argue that real-world programs are significantly more complex than this, and the order in which secret information occurs may be useful information for an attacker to learn. Nevertheless, if so desired, minor modifications can be made to the definition of $\mathcal{S}$ to weaken the attacker model; for instance, by defining $\mathcal{S}$ as a function mapping a variable name to the finite sequence of secret values that have arisen from that variable (*i.e.* $\mathcal{S} : \mathbb{V} \to (\mathbb{N} \to \mathbb{Z})$), it is possible to model an attacker that is unable to distinguish between scenarios that only differ by the order in which secret variable names occur.

4.4.2 Secret and Observable Information in CH-IMP's Basic Semantics

In Section 4.3.3 (p. 91), we emphasised the importance of tracking *all* secret and observable values that occur along a particular path of execution of a program. Now that we have established formal definitions of a program's "secret information" and "observable information", we must modify our formal definition of a CH-IMP program (Definition 4.3, p. 82) to ensure that this information is correctly recorded during the program's execution.

Because the secret and observable information that occurs in a program may differ along different paths of execution, each state of the DTMC must record the secret and observable information that has occurred so far along the current path. This requires the definition of DTMC nodes representing a CH-IMP program's state to be modified to

$$S : C \times \sigma \times \mathcal{S} \times \mathcal{O}$$

where $\mathcal{S}$ and $\mathcal{O}$ are the definitions of secret and observable information in Definitions 4.4 and 4.5 (p. 100) respectively. Empty sequences for $\mathcal{S}$ and $\mathcal{O}$ must also be added to the DTMC's initial state $\bar{s}$, since a program contains no secret or observable

information when it begins executing:

$$\bar{s} = (\mathcal{P}, \langle\{\}\rangle, \langle\rangle, \langle\rangle).$$

Definition 4.3 contains nine semantic rules that define a DTMC's transition probability matrix **P**; each rule modifies at least one element in the tuple that constitutes a state in the DTMC in order to produce a succeeding state with a given probability. We must therefore also redefine the nine semantic rules listed in Definition 4.3 to account for the addition of secret and observable information to states, but because none of these rules process secret or observable information, it is sufficient for the rules to simply "carry forward" both $\mathcal{S}$ and $\mathcal{O}$ into the succeeding states without modification. For example, the *Decl* rule, which declares a new variable in the narrowest scope frame, would be rewritten in the following way:

$$\textit{Decl}\ \frac{}{(\mathsf{new}\ V := \rho;\ C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, (\{V \to n\} \cup o) :: \sigma, \mathcal{S}, \mathcal{O})}$$

Since declaration of a variable has no direct effect on a program's secret or observable information, it is unnecessary to modify either $\mathcal{S}$ or $\mathcal{O}$ in the succeeding states. The same principle holds for the other eight rules.

Now that we have modified the semantic rules to track secret and observable information that occurs during a program's execution, we are ready to complete the definition of CH-IMP by formally stating the behaviour of its secret and observe commands.

4.4.3 Formalising CH-IMP's Information Flow Commands

Recall:

(a) from Section 4.3.2 (p. 89) that *all* secret values that occur along a particular path of execution of a program must be recorded;

(b) from Section 4.3.4 (p. 94) that the variable names associated with particular secret values and the order in which secret values occur are both secret information in their own right; and

(c) from our informal definition of the secret command in the introduction to Section 4.3 (p. 86) that a value marked as secret with the secret command should be considered secret at the point at which the secret command is evaluated.

The semantic rule defining the behaviour of the secret command therefore evaluates the given variable as an arithmetic expression and appends a tuple consisting of the variable name and the result of the evaluation to the sequence of tuples that constitute the program's secret information along this path of execution:

$$Sec \quad \frac{}{(\mathsf{secret}\ V; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S} :: (V, [\![V]\!](\sigma)), \mathcal{O})}$$

Similarly, recall:

(a) from Section 4.3.1 (p. 88) that *all* observable values that occur along a particular path of execution of a program must be recorded, and

(b) from our informal definition of the observe command in the introduction to Section 4.3 that a value marked as observable with the observe command should be considered observable at the point at which the observe command is evaluated.

The semantic rule defining the behaviour of the observe command therefore simply evaluates the given variable as an arithmetic expression and appends the result to the sequence of values that have been observed along this path of execution:

$$Obs \quad \frac{}{(\mathsf{observe}\ V; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O} :: [\![V]\!](\sigma))}$$

These rules ensure that, in any given state in the DTMC, *any* secret or observable information that has occurred along that path of the program's execution is recorded.

Finally, by combining the now-modified semantic rules in Definition 4.3 (p. 82) and the semantic rules for the secret and observe commands above, we arrive at the following definition of CH-IMP with information flow semantics:

Definition 4.6 (CH-IMP execution with information flow semantics)
The execution of a CH-IMP program $\mathcal{P}$ *with an information flow semantics* is formalised as a discrete-time Markov chain (DTMC) $\mathcal{D} = (S, \bar{s}, \mathbf{P})$ with states $S : C \times \sigma \times \mathcal{S} \times \mathcal{O}$, where

- C is a finite sequence of commands to be executed (*i.e.* an expression derived from the grammar in Definition 4.1 (p. 80)),
- $\sigma : \mathbb{N} \to o$ is a finite sequence of variable scope frames (with the narrowest scope frame at the start of the sequence),
- a variable scope frame $o : \mathbb{V} \to \mathbb{Z}$ is a function mapping variable names to values,
- the secret information $\mathcal{S} : \mathbb{N} \to (\mathbb{V} \times \mathbb{Z})$ is a finite sequence of secret variable name/value tuples that have occurred, and
- the observable information $\mathcal{O} : \mathbb{N} \to \mathbb{Z}$ is a finite sequence of observable values that have occurred;

the initial state $\bar{s} = (\mathcal{P}, \langle\{\}\rangle, \langle\rangle, \langle\rangle)$; and the transition probability matrix $\mathbf{P}$ defined according to the following semantic rules:

$$\textit{Decl}\ \frac{}{(\mathsf{new}\ V := \rho;\ C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, (\{V \to n\} \cup o) :: \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{Assign}\ \frac{}{(V := \rho;\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, \{V \to n\} \oplus \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{IfTrue}\ \frac{[\![B]\!](\sigma) \to \text{true}}{(\text{if } (B) \{ C_T \} \text{ else } \{ C_F \}; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\text{start}; C_T; \text{end}; C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{IfFalse}\ \frac{[\![B]\!](\sigma) \to \text{false}}{(\text{if } (B) \{ C_T \} \text{ else } \{ C_F \}; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\text{start}; C_F; \text{end}; C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{WhileTrue}\ \frac{[\![B]\!](\sigma) \to \text{true}}{(\text{while } (B) \{ C_W \}; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\text{start}; C_W; \text{end}; \text{while } (B) \{ C_W \}; C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{WhileFalse}\ \frac{[\![B]\!](\sigma) \to \text{false}}{(\text{while } (B) \{ C_W \}; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{ScopeIn}\ \frac{}{(\text{start}; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \{\} :: \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{ScopeOut}\ \frac{}{(\text{end}; C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O})}$$

$$\textit{Sec}\ \frac{}{(\text{secret } V; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S} :: (V, [\![V]\!](\sigma)), \mathcal{O})}$$

$$\textit{Obs}\ \frac{}{(\text{observe } V; C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O} :: [\![V]\!](\sigma))}$$

$$\textit{Accept}\ \frac{}{(\langle\rangle, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (\langle\rangle, \sigma, \mathcal{S}, \mathcal{O})}$$

4.5 Quantitative Security Policies for CH-IMP Programs

Now that we have a formal definition of CH-IMP with an information flow semantics, we can consider how to use a DTMC induced by the semantics to quantify the information leakage that occurs from a CH-IMP program and how to define a security policy for a CH-IMP program.

Recall that the purpose of a security policy is to determine whether the amount of information leaked by a system is acceptable. In this section we define the information leakage from a CH-IMP program as the mutual information of its secret and observable values, or alternatively as the min-entropy leakage from its secret values

to its observable values. A CH-IMP security policy, then, states an upper bound on the acceptable amount of information a program may leak, given a particular leakage measure; *e.g.*, "the mutual information of the secret and publicly-observable information in the program must not exceed 2 bits". (This definition of "security policy" is similar to Heusser and Malacaria's (2010), in that the security policy is a numerical upper bound on a particular leakage measure beyond which the program is deemed insecure.)

By executing a terminating CH-IMP program according to Definition 4.6, we obtain a DTMC with a structure similar to that depicted in Figure 4.2; this is the DTMC induced by executing the bitwise XOR-reversing example shown in Listing 4.1 (p. 88). The branching behaviour in the DTMC is caused by the new commands: the first by the declaration of *rand* (generating two succeeding states), and the second by the declaration of *sec* (generating two succeeding states for each of the previous two succeeding states). As explained in Section 4.4.3 (p. 103), $\mathcal{S}$ and $\mathcal{O}$ are either populated by the evaluation of the secret and observe commands respectively, or "carried forward" to succeeding states by the evaluation of all other commands. After the final observe command is executed, an accepting state is entered, signifying that that particular path of execution has ended.

Note that each of the accepting states in the DTMC is of the form $(\langle\rangle, \sigma, \mathcal{S}, \mathcal{O})$, and that $\mathcal{S}$ and $\mathcal{O}$ contain the secret and observable information that has occurred along the path of execution from the initial state to that accepting state. We denote the set of all sequences of secret information that occur in the accepting states of a DTMC with **S**, and the equivalent for all sequences of observable information with **O**.

From Definition 2.16 (p. 32), the probability of the DTMC entering a given accepting state can be computed by multiplying the probability of each transition that occurs between the path's initial and accepting states; this allows us to compute the probability of some secret information $\mathcal{S}$ and some observable information $\mathcal{O}$ occurring simultaneously — thus defining a joint probability distribution, which we denote with $P_{\mathbf{SO}}$.

Because multiple accepting states may contain the same secret and observable information, the probability of particular sequences of secret and observable information occurring jointly must be computed over all possible accepting states:

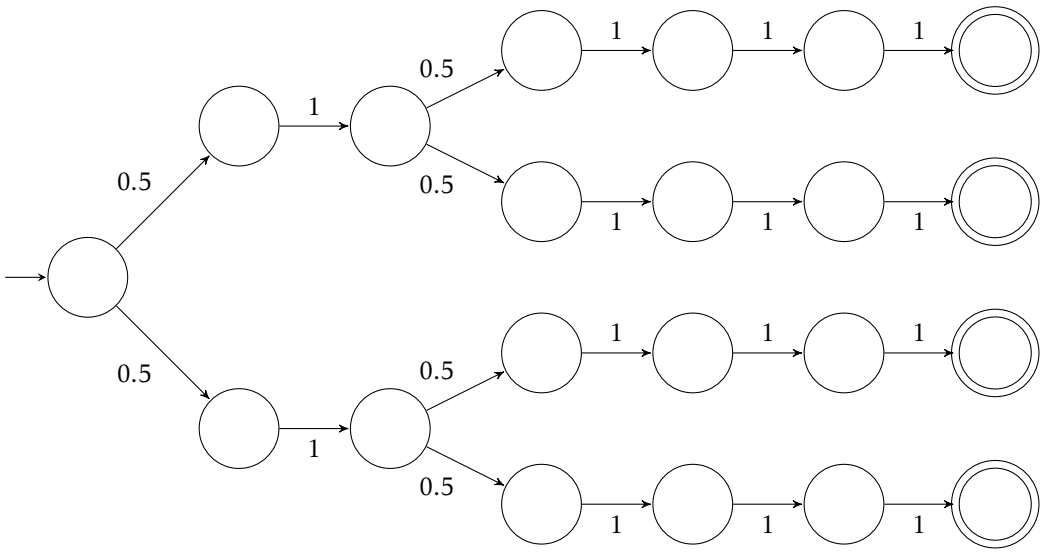
Definition 4.7 (joint probability distribution of secret and observable values in a terminating CH-IMP program)

The joint probability distribution $P_{\mathbf{SO}}$ of the secret and observable values that occur in a terminating CH-IMP program is given by

$$P_{\mathbf{SO}}(\mathcal{S},\mathcal{O}) = \sum_{\underline{s}=(C,\sigma,s',o')\in\{\underline{S}|s'=\mathcal{S}\wedge o'=\mathcal{O}\}} P(\underline{s}).$$

From the joint probability distribution $P_{\mathbf{SO}}$, we can compute the probability distributions of the secret information $P_{\mathbf{S}}$ and of the observable information $P_{\mathbf{O}}$ by marginalising $P_{\mathbf{SO}}$, per Definition 2.11 (p. 28):

Figure 4.2: an overview of the DTMC induced by Definition 4.6 (p. 105) by executing the program shown in Listing 4.1 (p. 88)



Definition 4.8 (probability distributions of the secret values and observable values in a terminating CH-IMP program)

The probability distributions $P_{\mathbf{S}}$ of the secret values and $P_{\mathbf{O}}$ of the observable values that occur in a terminating CH-IMP program are given by

$$P_{\mathbf{S}}(\mathcal{S}) = \sum_{\mathcal{O} \in \mathbf{O}} P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O});$$
$$P_{\mathbf{O}}(\mathcal{O}) = \sum_{\mathcal{S} \in \mathbf{S}} P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O}).$$

Now that $P_{\mathbf{S}}$, $P_{\mathbf{O}}$ and $P_{\mathbf{SO}}$ are defined, we are able to compute the mutual information of the secret and observable information using Definition 2.21 (p. 35):

$$I(\mathbf{S};\mathbf{O}) = \sum_{\mathcal{S} \in \mathbf{S}} \sum_{\mathcal{O} \in \mathbf{O}} P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O}) \log_2 \frac{P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O})}{P_{\mathbf{S}}(\mathcal{S}) \cdot P_{\mathbf{O}}(\mathcal{O})}.$$

Using the same probability distributions, we are able to compute the min-entropy leakage from the secret information to the observable information using Definition 2.28 (p. 40):

$$\mathcal{L}_{\mathbf{SO}} = -\log_2 \max_{\mathcal{S} \in \mathbf{S}} \sum_{\mathcal{O} \in \mathbf{O}} P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O}) + \log_2 \sum_{\mathcal{O} \in \mathbf{O}} \max_{\mathcal{S} \in \mathbf{S}} P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O}).$$

Alternatively, as mentioned in the introduction to this chapter, any other information leakage measure that is a function of $P_{\mathbf{SO}}$ can also be computed.

Note that, ultimately, the purpose of executing a CH-IMP program is to obtain the joint probability distribution $P_{\mathbf{SO}}$ — which is derived from the secret and observable information that occurs along each path of execution of the program, and the probability of each of these paths occurring — in order to compute an information leakage measure that is a function of $P_{\mathbf{SO}}$. Our approach is therefore comparable to executing a CH-IMP program using a probabilistic trace semantics, where the traces are restricted to the distinct sequences of secret and observable information that occur during execution, and computing the information leakage measure from the probability distri-

bution over the resulting traces.

4.6 Summary

This chapter presents a formal definition of a novel quantitative point-to-point information flow model in CH-IMP, a probabilistic language; it quantifies the amount of information that flows from secret information that occurs at defined points to publicly-observable information that occurs at other points, with an attacker model that assumes a passive attacker with prior knowledge of the program's behaviour. We have demonstrated how the information collected from the execution of a terminating CH-IMP program can be used to formulate a quantitative security policy; any information leakage measure that is a function of the joint probability distribution of the secret and publicly-observable information, including mutual information and min-entropy leakage, may be used.

The novelty of this information flow model is twofold: it permits the unrestricted occurrence of secret and publicly-observable information in programs, and permits any (positive) number of pieces of secret and publicly-observable information to occur. This is an advance in the field of quantitative information flow analysis, as existing models heavily restrict where and how often each type of information may occur in the program. Ours is the first such model to relax these common restrictions.

5

An Implementation of the Information Flow Model

In Chapter 4 we presented a quantitative point-to-point information flow model for CH-IMP, a probabilistic language, and showed that the information flow model can be used to quantify the information leakage that occurs from CH-IMP programs in terms of the information-theoretic measures of mutual information and min-entropy leakage. In this chapter we shall demonstrate that the CH-IMP information flow model is practical, and can be used to analyse the security of real-world systems and protocols (albeit of limited complexity), by presenting `chimp`: a concrete implementation of both the CH-IMP language and its accompanying information flow model.

`chimp`'s implementation of CH-IMP's information flow semantics (Definition 4.6, p. 105) does not naively store the entire discrete-time Markov chain representation of the program's states in memory; most of the information in the DTMC is irrelevant to the final information flow analysis stage where the information flow from the secret values to the observable values is quantified, and can in fact be discarded while the DTMC is still being explored. Instead, `chimp` operates on a data structure that we term an *environment function*, which tracks only the states of the discrete-time Markov chain that are or could become accepting states at a given moment. By discarding information that is not necessary for the final computation of the program's information

flow, chimp is able to analyse programs efficiently, in terms of both analysis time and memory usage.

We evaluate the performance of chimp and QUAIL, another quantitative information flow analysis tool from the literature. We shall see that chimp's performance is comparable to QUAIL's in terms of both total analysis time and peak memory usage when they are tasked with analysing the same probabilistic system; we shall also see that chimp scales slightly better than QUAIL when analysing probabilistic systems of increasing complexity. Nevertheless, there are limitations on our tool's (and, by implication, CH-IMP's) ability to efficiently analyse programs; we shall consider what causes these limitations to arise.

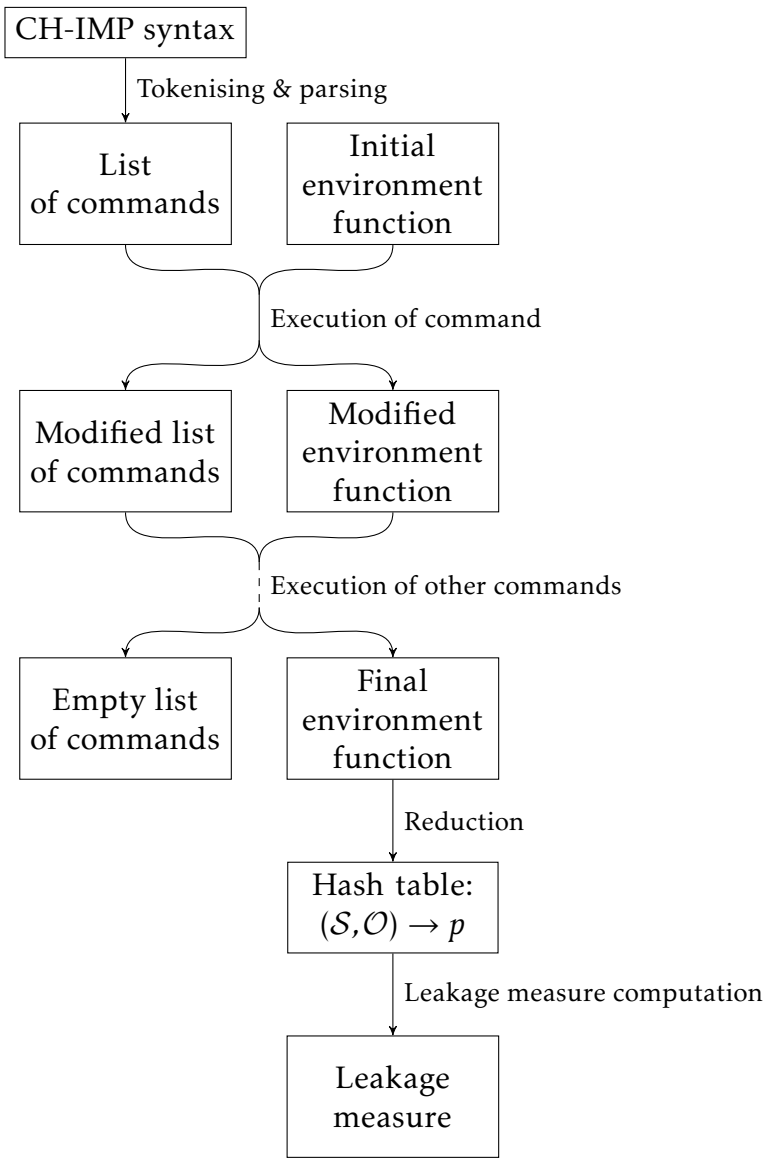
As with Chapter 4, the contents of this chapter are based primarily on one of our contributions to the literature (Chothia et al., 2013b).

5.1 An Overview of the chimp Tool

chimp is an implementation of the CH-IMP language and its information flow semantics, per Definition 4.6 (p. 105); it is written in the OCaml programming language.

An overview of chimp's structure is shown in Figure 5.1. A tokeniser and parser for the CH-IMP syntax allows programs to be written in the grammar defined in Definition 4.6, and produces an abstract syntax tree representing the sequence of commands C. The state space of the DTMC that is induced by executing C is then explored by chimp, and the values of $\mathcal{S}$ and $\mathcal{O}$ in each accepting state are used to compute the joint probability distribution P_{SO} (Definition 4.7, p. 108) when all of the accepting states have been discovered. Afterwards, chimp can use P_{SO} to compute a number of information leakage measures, including mutual information and min-entropy leakage, as outlined in Section 4.5 (p. 106).

Figure 5.1: an overview of the `chimp` tool



5.1.1 Executing a CH-IMP Program

Although the execution of a CH-IMP program is formally defined in terms of a DTMC, `chimp` does not actually store the DTMC's entire state space for two reasons:

(a) The commands in C are executed sequentially, and since the CH-IMP grammar does not contain a command ordering the program to terminate, either a command in C will eventually be executed along a particular path of execution of the program, or there will be no accepting state along that path of execution (*i.e.* because an infinite loop occurs).

(b) As noted in Section 4.5, only the secret information $\mathcal{S}$ and observable information $\mathcal{O}$ that occur when each path of execution ends are required to compute $P_{\mathbf{SO}}$. It is therefore only necessary for `chimp` to track the states that could be accepting states at any given time: because DTMCs have the Markov property (Section 2.2, p. 30) — for any given state, the probability of transitioning into a succeeding state is determined only by the current state — as soon as a state transitions into a state that is not itself, the previous state can be discarded.

Thus, instead of operating on states of a DTMC, `chimp` operates on the sequence of commands C and an *environment function* $E : e \to p$ mapping an *environment* $e : (\sigma, \mathcal{S}, \mathcal{O})$ to the probability p of the environment occurring as part of an accepting state in the DTMC. Informally, an environment function can be viewed as a collection of the information stored by states that are or could potentially become accepting states, along with the probability of each of these collections of information occurring were the program to terminate at that instant. The advantage of exploring the state space in this manner is its improved efficiency over a naive exploration of all of the DTMC's states: memory is not wasted storing information about states that are not required for the goal of computing $P_{\mathbf{SO}}$, and processor time is not wasted needlessly iterating over them.

After tokenising and parsing, `chimp` proceeds to execute a CH-IMP program as follows. Beginning with an *initial environment* — consisting of a sequence of scope frames σ containing a single scope frame with no bound variables, and no secret or observable information — that occurs with probability 1, `chimp` removes the first command c from the sequence of commands C and executes it in the following manner:

- If c is any command other than an `if` or `while` command, `chimp` performs the necessary modifications to σ, $\mathcal{S}$ and $\mathcal{O}$ in each environment in the environment function. If the command performs a variable declaration or assignment, previously unseen environments may occur with a non-zero probability, and the probabilities of the environment function are modified accordingly.

- If c is an `if` command, environments in the environment function are partitioned according to whether the guard B in the `if` command evaluates to true or false; the partitions are referred to as E_B and $E_{\neg B}$ respectively. Execution of the program proceeds in parallel: the sequence of commands in the true branch C_T is executed using E_B as the environment function, and the sequence of commands in the false branch C_F is executed using $E_{\neg B}$ as the environment function. When both branches finish executing, their terminating environment functions are merged, and execution of the sequence of commands C continues using the merged environment function.

- If c is a `while` command, environments in the environment function are partitioned according to whether the guard B in the `while` command evaluates to true or false, as with the `if` command; again, the partitions are referred to as E_B and $E_{\neg B}$ respectively. The sequence of commands in the body of the `while` command C_W is executed using E_B as the environment function, and when it finishes executing, its terminating environment function is merged with $E_{\neg B}$ and execution of the sequence of commands C continues using the merged environment function.

The same procedure is followed for the next command in C, and so on until C is an empty sequence (at which point, each path of execution of the program has ended, the DTMC's state space has been fully explored, and all of its accepting states have been discovered).

`chimp` implements environment functions using the following data structures:

- a scope frame is a list of pairs whose first component is the name of a variable declared within this level of scope and whose second component is the (integer) value of that variable,
- a sequence of scope frames σ is a list,
- the secret information $\mathcal{S}$ is a list of pairs whose first component is the name of a variable with a value that has previously been identified as secret and whose second component is the variable's (integer) value at that moment (*i.e.*, it has same structure as a scope frame),
- the observable information $\mathcal{O}$ is a list of (integer) values that have previously been identified as observable,
- an environment e is a 3-tuple, and
- an environment function E is a pair whose first component is the 3-tuple representing an environment e and whose second component is a floating-point number representing the probability of e occurring as part of an accepting state in the DTMC.

`chimp` must ensure that each environment in the domain of E (*i.e.* the first component in each element of the list representing E) is unique. Executing a command may cause several previously distinct environments to become identical (*e.g.* when they differ only by the value of a single variable V and a new value is assigned to V with probability 1); therefore, when constructing the succeeding environment function while

executing a command that may cause such behaviour, the probabilities to which duplicate environments are mapped must be summed. Given the data structures outlined above, this summing operation must perform a number of list, pair and tuple comparisons to determine which environments are identical; the speed of this operation, and therefore the execution speed of commands causing this behaviour in `chimp`, is a function of the number of variables currently in scope (as well as of the number of values that have been identified as secret or observable). CH-IMP's variable scoping feature improves the efficiency of this operation: by declaring variables inside an if branch or while loop when those variables are only accessed within the scope of that branch or loop (as opposed to declaring them inside a broader scope frame), the list representing σ remains as small as possible throughout execution of the program, thus avoiding unnecessarily large list comparisons when testing the equality of two instances of σ in two environments during the summing operation.

5.1.2 Executing an Example CH-IMP Program

To show how the `chimp` tool executes a CH-IMP program, we demonstrate the procedure from Section 5.1.1 being applied to the program in Listing 5.1 — a simple example in which an information leak may occur depending on the random value of a variable — and the initial environment function E_0:

Listing 5.1: a CH-IMP program containing a potential information flow from the secret value of *sec* to the observable value of *out*

```
1  new sec := { 0 → 0.5, 1 → 0.5 };
2  new out := 0;
3  new rand := { 0 → 0.25, 1 → 0.75 };
4  secret sec;
5  if (rand == 0) {
6      out := sec
7  } else {
8      out := 0
9  };
10 observe out
```

$$
\begin{aligned}
E_0 = \{ & \\
& (\langle\{\}\rangle, \langle\rangle, \langle\rangle) \to 1 \\
\}&
\end{aligned}
$$

The program is transformed into a sequence of commands C, and the first command in this sequence is applied to E_0; it declares a new variable *sec* with a value of 0 or 1 with equal probability. `chimp` generates two succeeding environments with the appropriate modifications made to σ in each environment, and adjusts the probabilities of each of them occurring as appropriate: both are given a probability of $P(\text{environment in } E_0 \text{ occurring}) \times P(\text{assignment occurring}) = 1 \times 1/2 = 1/2$. Note that the probability of the initial environment now being an accepting state in the DTMC is 0; it is therefore not shown in the modified environment function E_1.

$$
\begin{aligned}
E_1 = \{ & \\
& (\langle\{\mathit{sec} \to 0\}\rangle, \langle\rangle, \langle\rangle) \to 1/2, \\
& (\langle\{\mathit{sec} \to 1\}\rangle, \langle\rangle, \langle\rangle) \to 1/2 \\
\}&
\end{aligned}
$$

The next command in the list declares another variable *out*; `chimp` again makes the appropriate modifications to σ in each environment in E_1. The value assigned to *out* during declaration is always 0, so no changes to the probabilities in E_2 are necessary.

$$
\begin{aligned}
E_2 = \{ & \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0\}\rangle, \langle\rangle, \langle\rangle) \to 1/2, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0\}\rangle, \langle\rangle, \langle\rangle) \to 1/2 \\
\}&
\end{aligned}
$$

Another variable, *rand*, is declared; its value is 0 with probability 1/4, and 1 with probability 3/4. `chimp` generates four succeeding environments for E_2 with the variable binding for *rand* added to σ. As before, the adjusted probability given to a particular environment in the new environment function E_3 is $P(\text{environment in } E_2 \text{ occurring}) \times P(\text{assignment occurring})$; *e.g.*, for the environment in which *sec* equals 0 and the new variable *rand* equals 1, the probability is $1/2 \times 1/4 = 1/8$.

$$
\begin{aligned}
E_3 = \{ & \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 0\}\rangle, \langle\rangle, \langle\rangle) \to 1/8, \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 1\}\rangle, \langle\rangle, \langle\rangle) \to 3/8, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 0\}\rangle, \langle\rangle, \langle\rangle) \to 1/8, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 1\}\rangle, \langle\rangle, \langle\rangle) \to 3/8 \\
\}
\end{aligned}
$$

On line 4, the value of *sec* is marked as secret. In each environment, `chimp` evaluates *sec* in σ by searching the sequence of scope frames from left to right for a mapping from *sec* to some value n; `chimp` then stores a permanent copy of this mapping in $\mathcal{S}$.

$$
\begin{aligned}
E_4 = \{ & \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 0\}\rangle, \langle \mathit{sec} \to 0\rangle, \langle\rangle) \to 1/8, \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 1\}\rangle, \langle \mathit{sec} \to 0\rangle, \langle\rangle) \to 3/8, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 0\}\rangle, \langle \mathit{sec} \to 1\rangle, \langle\rangle) \to 1/8, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 1\}\rangle, \langle \mathit{sec} \to 1\rangle, \langle\rangle) \to 3/8 \\
\}
\end{aligned}
$$

On line 5, a branching operation occurs: the environments in E_5 are therefore partitioned according to whether the guard B (*rand* == 0) evaluates to true or false. Execution of the true branch of the `if` command continues using the environment function consisting of those environments E_{5_B} in which B evaluates to true, and execution of the false branch continues using the environment function consisting of those environments $E_{5_{\neg B}}$ in which B evaluates to false. Note that the probabilities in each environment function are *not* scaled up so that the sum of the probabilities equals 1: the functions will be merged when the commands in the true and false branches have been executed, so this is unnecessary.

$$
\begin{aligned}
E_{5_B} = \{ & \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 0\}\rangle, \langle \mathit{sec} \to 0\rangle, \langle\rangle) \to 1/8, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 0\}\rangle, \langle \mathit{sec} \to 1\rangle, \langle\rangle) \to 1/8 \\
\}
\end{aligned}
$$

$$
\begin{aligned}
E_{5_{\neg B}} = \{ & \\
& (\langle\{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 1\}\rangle, \langle \mathit{sec} \to 0\rangle, \langle\rangle) \to 3/8, \\
& (\langle\{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 1\}\rangle, \langle \mathit{sec} \to 1\rangle, \langle\rangle) \to 3/8 \\
\}
\end{aligned}
$$

`chimp` now applies the true branch to E_{5_B}. The list of commands in this branch begins with an implicit start command, per the *IfTrue* semantic rule in Definition 4.6 (p. 105); this inserts a new variable scope frame as the first element in each environment's σ:

$$\begin{aligned}
&E_{6_{\text{pre}}} = \{ \\
&\quad (\langle \{\}, \{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 0\} \rangle, \langle \mathit{sec} \to 0 \rangle, \langle \rangle) \to 1/8, \\
&\quad (\langle \{\}, \{\mathit{sec} \to 1, \mathit{out} \to 0, \mathit{rand} \to 0\} \rangle, \langle \mathit{sec} \to 1 \rangle, \langle \rangle) \to 1/8 \\
&\}
\end{aligned}$$

Next, the only command explicitly stated in the branch is executed: the value of *sec* is copied into *out* with probability 1. In each environment, `chimp` evaluates *sec* (producing the integer m), searches the sequence of scope frames from left to right for a mapping from *out* to some value n, and overwrites n with m. Note that this requires the modification of only one of the environments in E_6.

$$\begin{aligned}
&E_6 = \{ \\
&\quad (\langle \{\}, \{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 0\} \rangle, \langle \mathit{sec} \to 0 \rangle, \langle \rangle) \to 1/8, \\
&\quad (\langle \{\}, \{\mathit{sec} \to 1, \mathit{out} \to 1, \mathit{rand} \to 0\} \rangle, \langle \mathit{sec} \to 1 \rangle, \langle \rangle) \to 1/8 \\
&\}
\end{aligned}$$

The list of commands in the true branch ends with an implicit end, again inserted by the *IfTrue* semantic rule; this causes the narrowest variable scope frame — *i.e.* the first element in each environment's σ — to be destroyed (although, since no variables were declared in the true branch, no variables are actually unbound). This requires the probabilities of environments in E_6 that differ only by their first element in σ to be summed, but that is not necessary in this case because the environments also differ by other elements in σ and by their values of $\mathcal{S}$.

$$\begin{aligned}
&E_{6_{\text{post}}} = \{ \\
&\quad (\langle \{\mathit{sec} \to 0, \mathit{out} \to 0, \mathit{rand} \to 0\} \rangle, \langle \mathit{sec} \to 0 \rangle, \langle \rangle) \to 1/8, \\
&\quad (\langle \{\mathit{sec} \to 1, \mathit{out} \to 1, \mathit{rand} \to 0\} \rangle, \langle \mathit{sec} \to 1 \rangle, \langle \rangle) \to 1/8 \\
&\}
\end{aligned}$$

Having executed the true branch of the if command, `chimp` now applies the false branch to $E_{5_{\neg B}}$. Again, an implicit start is executed first:

$$
\begin{aligned}
&E_{8_{\text{pre}}} = \{ \\
&\quad (\langle \{\}, \{sec \to 0, out \to 0, rand \to 1\} \rangle, \langle sec \to 0 \rangle, \langle \rangle) \to 3/8, \\
&\quad (\langle \{\}, \{sec \to 1, out \to 0, rand \to 1\} \rangle, \langle sec \to 1 \rangle, \langle \rangle) \to 3/8 \\
&\}
\end{aligned}
$$

This is followed by the overwriting of the value of *out* with the integer 0 with probability 1; note that this requires no modifications to any of the environments in E_8, since *out* already evaluates to 0 in all of them.

$$
\begin{aligned}
&E_8 = \{ \\
&\quad (\langle \{\}, \{sec \to 0, out \to 0, rand \to 1\} \rangle, \langle sec \to 0 \rangle, \langle \rangle) \to 3/8, \\
&\quad (\langle \{\}, \{sec \to 1, out \to 0, rand \to 1\} \rangle, \langle sec \to 1 \rangle, \langle \rangle) \to 3/8 \\
&\}
\end{aligned}
$$

Finally in the false branch, the implicit end command is executed (again, causing the unbinding of no variables):

$$
\begin{aligned}
&E_{8_{\text{post}}} = \{ \\
&\quad (\langle \{sec \to 0, out \to 0, rand \to 1\} \rangle, \langle sec \to 0 \rangle, \langle \rangle) \to 3/8, \\
&\quad (\langle \{sec \to 1, out \to 0, rand \to 1\} \rangle, \langle sec \to 1 \rangle, \langle \rangle) \to 3/8 \\
&\}
\end{aligned}
$$

Now that both branches of the if command on line 5 have been executed, the final environment functions from each branch — $E_{6_{\text{post}}}$ from the true branch and $E_{8_{\text{post}}}$ from the false branch — must be merged to form the function that will be used in the execution of line 10. As with scope frame destruction, this involves summing the probabilities of the environments that occur in both $E_{6_{\text{post}}}$ *and* $E_{8_{\text{post}}}$ with a non-zero probability; in this example, however, this is not necessary because no environments occur in both functions with a non-zero probability.

$$
\begin{aligned}
&E_9 = \{ \\
&\quad (\langle \{sec \to 0, out \to 0, rand \to 0\} \rangle, \langle sec \to 0 \rangle, \langle \rangle) \to 1/8, \\
&\quad (\langle \{sec \to 0, out \to 0, rand \to 1\} \rangle, \langle sec \to 0 \rangle, \langle \rangle) \to 3/8, \\
&\quad (\langle \{sec \to 1, out \to 1, rand \to 0\} \rangle, \langle sec \to 1 \rangle, \langle \rangle) \to 1/8, \\
&\quad (\langle \{sec \to 1, out \to 0, rand \to 1\} \rangle, \langle sec \to 1 \rangle, \langle \rangle) \to 3/8 \\
&\}
\end{aligned}
$$

The final command of the program causes the value of *out* to be revealed to the attacker; `chimp` evaluates *out* in σ in each environment and stores its current value (but not the name of the variable itself) in $\mathcal{O}$.

$$
\begin{aligned}
E_{10} = \{ & \\
& (\langle \{sec \to 0, out \to 0, rand \to 0\} \rangle, \langle sec \to 0 \rangle, \langle 0 \rangle) \to 1/8, \\
& (\langle \{sec \to 0, out \to 0, rand \to 1\} \rangle, \langle sec \to 0 \rangle, \langle 0 \rangle) \to 3/8, \\
& (\langle \{sec \to 1, out \to 1, rand \to 0\} \rangle, \langle sec \to 1 \rangle, \langle 0 \rangle) \to 1/8, \\
& (\langle \{sec \to 0, out \to 0, rand \to 1\} \rangle, \langle sec \to 1 \rangle, \langle 1 \rangle) \to 3/8 \\
\} &
\end{aligned}
$$

The sequence of commands C is now empty; by following the algorithm described in Section 5.1.1, `chimp` has identified the information that would be contained in the set of accepting states $\underline{S}$ of the DTMC that would be induced if the program shown in Listing 5.1 were executed using the formal definition of CH-IMP with information flow semantics (Definition 4.6, p. 105), along with the probability of the DTMC entering each of these accepting states.

We now consider how the *final environment function* — in this example, E_{10} — may be used to compute the joint probability distribution $P_{\mathbf{SO}}$ of the secret and publicly-observable information that occurs in the program, and thus verify whether the program conforms to a particular quantitative security policy.

5.1.3 Verifying a CH-IMP Program's Security Policy

We have seen that, by executing the algorithm described in Section 5.1.1, `chimp` mutates an environment function that maps environments — the sequence of variable scope frames σ, secret information $\mathcal{S}$ and observable information $\mathcal{O}$ that occur in a state in the DTMC representation of a CH-IMP program's execution — to the probability of each of those environments occurring. When this algorithm terminates, CH-IMP is left with the *final environment function*: the environments in this function are the $(\sigma, \mathcal{S}, \mathcal{O})$ tuples that exist in each accepting state $\underline{s}$ in the DTMC, and the probability to which each environment is mapped is the probability of the DTMC entering that accepting

state (*i.e.* the probability of a particular path of execution of the program ending in that state). We now consider how `chimp` can compute the joint probability distribution $P_{\mathbf{SO}}$ from this final environment function, and use this distribution to verify whether a CH-IMP program's security policy is satisfied.

Recall from Section 4.5 (p. 106) that, by multiplying the probability of each transition that occurs between a path's initial and accepting states, we can compute the probability of the secret and observable information present in the accepting state occurring simultaneously; *i.e.* we can compute $P_{\mathbf{SO}}(\mathcal{S},\mathcal{O})$. Since this information is already stored in the final environment function, `chimp` can construct the joint probability distribution $P_{\mathbf{SO}}$ of the secret and observable information present when the program terminates by stripping the sequence of scope frames σ from each environment in the domain of the final environment function and mapping the remaining $(\mathcal{S},\mathcal{O})$ tuple to the probability of that tuple occurring. Superficially, this may appear to be simply equal to the probability of the environment occurring, but since two environments in the final environment function may differ only by their value of σ, `chimp` must sum over the probability of all environments in which both $\mathcal{S}$ and $\mathcal{O}$ occur to compute $P_{\mathbf{SO}}(\mathcal{S},\mathcal{O})$.

When this process has been performed on each environment in the final environment function, `chimp` is left with the joint probability distribution $P_{\mathbf{SO}}$. Internally, `chimp` represents this as a two-dimensional hash table: the "outer" hash table $H_{\mathbf{SO}}$ maps each sequence of secrets $\mathcal{S}$ that occurs with a non-zero probability to an "inner" hash table $H_{\mathcal{S}\to\mathbf{O}}$ that maps each sequence of observable values $\mathcal{O}$ (that occurs simultaneously with $\mathcal{S}$ with a non-zero probability) to the probability of both $\mathcal{S}$ and $\mathcal{O}$ occurring simultaneously in the final environment function. Note that it is possible for some secret information $\mathcal{S}$ to occur with some non-zero probability, but for some observable information $\mathcal{O}$ *not* to occur simultaneously with $\mathcal{S}$. In these cases, the "inner" hash table $H_{\mathcal{S}\to\mathbf{O}}$ will not contain a mapping for $\mathcal{O}$; since this is equivalent to $P_{\mathbf{SO}}(\mathcal{S},\mathcal{O}) = 0$ in the joint probability distribution, `chimp` uses the implicit probability 0 whenever a lookup

fails in an "inner" hash table.

From the two-dimensional hash table $H_{\mathbf{SO}}$ representing $P_{\mathbf{SO}}$, `chimp` can construct the (one-dimensional) hash tables $H_{\mathbf{S}}$ and $H_{\mathbf{O}}$ representing the probability distributions $P_{\mathbf{S}}$ and $P_{\mathbf{O}}$ respectively; this is analogous to the marginalisation of a discrete probability distribution (Definition 2.11, p. 28). Both $H_{\mathbf{S}}$ and $H_{\mathbf{O}}$ can be constructed in a single iteration over each of the hash tables in $H_{\mathbf{SO}}$, as demonstrated in Algorithm 5.1.

Now that `chimp` has representations of the probability distributions $P_{\mathbf{S}}$, $P_{\mathbf{O}}$ and $P_{\mathbf{SO}}$, a number of information leakage measures can be computed, and it is possible to verify whether the CH-IMP program conforms to a particular quantitative information policy. For instance, `chimp` can apply Algorithm 5.2, derived from the mutual information equation in Definition 2.21 (p. 35), to compute the mutual information of the

Algorithm 5.1: given the hash table $H_{\mathbf{SO}}$ representing $P_{\mathbf{SO}}$, generates the hash tables $H_{\mathbf{S}}$ and $H_{\mathbf{O}}$ representing $P_{\mathbf{S}}$ and $P_{\mathbf{O}}$ respectively

```
function ComputeMarginalDistributions(H_SO)
    H_S ← the empty hash table
    H_O ← the empty hash table
    for all s ∈ keys of H_SO do
        h ← H_SO(s)                          ▷ h is the "inner" hash table for s
        for all o ∈ keys of h do
            p ← h(o)                         ▷ p is the joint probability of s and o
            ▷ Increase probability of s occurring by p
            if s ∈ keys of H_S then
                H_S(s) ← H_S(s) + p
            else
                H_S(s) ← p
            end if
            ▷ Increase probability of o occurring by p
            if o ∈ keys of H_O then
                H_O(o) ← H_O(o) + p
            else
                H_O(o) ← p
            end if
        end for
    end for
    return H_S, H_O
end function
```

secret and observable information in the CH-IMP program, or Algorithm 5.3 (p. 126), derived from the min-entropy leakage equation in Definition 2.28 (p. 40), to compute the min-entropy leakage from the secret information to the observable information; this can be compared with the acceptable upper bound defined by the security policy to verify whether or not the program is secure.

5.1.4 Verifying an Example CH-IMP Program's Security Policy

We now show how the procedures and algorithms described in Section 5.1.3 can be applied to verify whether the program shown in Listing 5.1 (p. 117) conforms to a particular quantitative security policy. In this instance, we shall assume that the security policy states that *no* information may flow from the value of *sec* on line 4 to the publicly-observable values, otherwise the program is deemed to be insecure.

Recall that we computed the final environment function E_{10} by executing the program using the procedure described in Section 5.1.1:

Algorithm 5.2: computes the mutual information of the secret and observable information of a program, given the hash table $H_{\mathbf{SO}}$ representing $P_{\mathbf{SO}}$

function MutualInformation($H_{\mathbf{SO}}$)
  $H_{\mathbf{S}}, H_{\mathbf{O}} \leftarrow$ **ComputeMarginalDistributions**($H_{\mathbf{SO}}$)
  $l \leftarrow 0$
  for all $s \in$ keys of $H_{\mathbf{S}}$ **do**
    $h \leftarrow H_{\mathbf{SO}}(s)$ ▷ *h is the "inner" hash table for s*
    for all $o \in$ keys of $H_{\mathbf{O}}$ **do**
      ▷ *Denominator in the mutual information equation must be* > 0
      if $H_{\mathbf{S}}(s) > 0$ **and** $H_{\mathbf{O}}(o) > 0$ **then**
        $l \leftarrow l + h(o) \times \log_2(h(o) \div (H_{\mathbf{S}}(s) \times H_{\mathbf{O}}(o)))$
      end if
    end for
  end for
  return l ▷ *l is the mutual information of* **S** *and* **O**
end function

Algorithm 5.3: computes the min-entropy leakage from the secret information to the observable information of a program, given the hash table $H_{\mathbf{SO}}$ representing $P_{\mathbf{SO}}$

function MinEntropyLeakage($H_{\mathbf{SO}}$)
  $hts \leftarrow$ the empty hash table
  $hlo \leftarrow$ the empty hash table
  for all $s \in$ keys of $H_{\mathbf{SO}}$ **do**
    $h \leftarrow H_{\mathbf{SO}}(s)$ ▷ *h is the "inner" hash table for s*
    for all $o \in$ keys of $H_{\mathbf{O}}$ **do**
      if $s \in$ keys of hts **then**
        $hts(s) \leftarrow hts(s) + h(o)$
      else
        $hts(s) \leftarrow h(o)$
      end if
      if $o \notin$ keys of hlo **or** $h(o) > hlo(o)$ **then**
        $hlo \leftarrow h(o)$
      end if
    end for
  end for
  ▷ *hts is the sum of the probabilities in* $H_{\mathbf{SO}}$ *for a given* $\mathcal{S} \in \mathbf{S}$
  ▷ *hlo is the largest probability in* $H_{\mathbf{SO}}$ *for a given* $\mathcal{O} \in \mathbf{O}$
  $v \leftarrow 0$
  for all $ts \in$ keys of hts **do**
    if $hts(ts) > v$ **then**
      $v \leftarrow hts(ts)$
    end if
  end for
  ▷ *v is the vulnerability of* **S**
  $cv \leftarrow 0$
  for all $lo \in$ keys of hlo **do**
    $cv \leftarrow cv + hlo(lo)$
  end for
  ▷ *cv is the conditional vulnerability of* **S** *given* **O**
  $l \leftarrow \log_2(cv \div v)$
  return l ▷ *l is the min-entropy leakage from* **S** *to* **O**
end function

$$
\begin{aligned}
E_{10} = \{ & \\
& (\langle\{sec \to 0, out \to 0, rand \to 0\}\rangle, \langle sec \to 0\rangle, \langle 0\rangle) \to 1/8, \\
& (\langle\{sec \to 0, out \to 0, rand \to 1\}\rangle, \langle sec \to 0\rangle, \langle 0\rangle) \to 3/8, \\
& (\langle\{sec \to 1, out \to 1, rand \to 0\}\rangle, \langle sec \to 1\rangle, \langle 0\rangle) \to 1/8, \\
& (\langle\{sec \to 0, out \to 0, rand \to 1\}\rangle, \langle sec \to 1\rangle, \langle 1\rangle) \to 3/8 \\
\}
\end{aligned}
$$

To compute the joint probability distribution $P_{\mathbf{SO}}$, we strip the sequence of variable scope frames σ from each environment and sum the probabilities over the remaining $(\mathcal{S},\mathcal{O})$ tuples, giving the following joint probability distribution:

	Observable values	
Secret values	0	1
$sec = 0$	1/2	0
$sec = 1$	3/8	1/8

To compute the mutual information of the secret and observable information, we apply Algorithm 5.2. First, we compute the marginal probability distributions $P_{\mathbf{S}}$ and $P_{\mathbf{O}}$:

$\mathcal{S}$	$P_{\mathbf{S}}(\mathcal{S})$
$sec = 0$	1/2
$sec = 1$	1/2

$\mathcal{O}$	$P_{\mathbf{O}}(\mathcal{O})$
0	7/8
1	1/8

Then for each $(\mathcal{S},\mathcal{O})$ pair in $P_{\mathbf{SO}}$ we compute $P_{\mathbf{SO}}(\mathcal{S},\mathcal{O}) \cdot \log_2 \frac{P_{\mathbf{SO}}(\mathcal{S},\mathcal{O})}{P_{\mathbf{S}}(\mathcal{S}) \cdot P_{\mathbf{O}}(\mathcal{O})}$ and sum the results:

$$
\begin{aligned}
I(\mathbf{S};\mathbf{O}) = {} & 0.5 \cdot \log_2 \tfrac{0.5}{0.5 \cdot 0.875} \;+\; 0 \cdot \log_2 \tfrac{0}{0.5 \cdot 0.125} \\
& + \; 0.375 \cdot \log_2 \tfrac{0.375}{0.5 \cdot 0.875} \;+\; 0.125 \cdot \log_2 \tfrac{0.125}{0.5 \cdot 0.125} \\
\approx {} & 0.14 \text{ bits.}
\end{aligned}
$$

To compute the min-entropy leakage from the secret information to the observable information, we apply Algorithm 5.3. We first compute, in a single iteration over $P_{\mathbf{SO}}$, the marginal probability distribution $P_{\mathbf{S}}$ and the largest probability in $P_{\mathbf{SO}}$ for each $\mathcal{O}$:

$\mathcal{S}$	$P_{\mathbf{S}}(\mathcal{S})$
sec = 0	1/2
sec = 1	1/2

$\mathcal{O}$	$\max_{\mathcal{S} \in \mathbf{S}} P_{\mathbf{SO}}(\mathcal{S}, \mathcal{O})$
0	1/2
1	1/8

Next, we compute the vulnerability of **S**, which is the largest probability in the range of $P_{\mathbf{S}}$ (in this case, 1/2), and the conditional vulnerability of **S** given **O**, which is the sum of the largest probability in $P_{\mathbf{SO}}$ for each $\mathcal{O}$:

$$\begin{aligned} V(\mathbf{S} \mid \mathbf{O}) &= 0.5 + 0.125 \\ &= 0.625. \end{aligned}$$

Finally, we compute the min-entropy leakage from **S** to **O**:

$$\begin{aligned} \mathcal{L}_{\mathbf{SO}} &= \log_2 \tfrac{0.625}{0.5} \\ &\approx 0.32 \text{ bits}. \end{aligned}$$

The program therefore contains an information flow from the secret information to the publicly-observable information; regardless of which information leakage measure we compute, the program exceeds the 0 bits permitted by the security policy we defined and thus the program is deemed to be insecure.

Further manual analysis of the program reveals why it leaks information. The information leakage occurs as a result of the assignment to *out* on line 6: if the value of *rand* equals 0, the secret value of *sec* is copied into *out*, which is later observed by the attacker on line 10. However, the attacker is hindered from knowing the secret value of *sec* with certainty by two factors: (*a*) their uncertainty about the value of *rand*, which controls which branch of the if command is executed, and (*b*) the fact that the value of *out* is hardcoded to 0 in the false branch, which prevents the attacker from distinguishing between the less likely scenario where *sec* equals 0 and the secret value was leaked completely, and the more likely scenario where they are observing the hardcoded value. Therefore, from $I(\mathbf{S};\mathbf{O})$, 0.14 bits of the 1 bit of secret information in the program are shared with the information that is observable by the attacker; from

$\mathcal{L}_{SO}$, *sec* is on average $2^{0.32} \approx 1\frac{1}{4}$ times more vulnerable to having its value deduced correctly in one attempt by the attacker.

5.2 Performance Evaluation

`chimp` is not the only quantitative information flow analysis tool: some of the quantitative information flow models we reviewed in Section 3.2 (p. 51) have implementations of their own. We shall now compare `chimp` with QUAIL (Biondi et al., 2013b), the Java-based implementation of Biondi et al.'s (2013a) information flow model; we briefly reviewed both the model and tool as part of our literature review in Section 3.2.8 (p. 60). QUAIL was chosen over other tools described in Chapter 3 for this comparison for several reasons: (*a*) like `chimp`, its system model is based on Markov chains, (*b*) it is readily available,[1] and (*c*) the authors have demonstrated in the literature how the tool can be used to quantify leaks in small probabilistic algorithms and security protocols.

We begin with a short review of QUAIL's information flow model and an overview of the tool itself.

5.2.1 An Overview of QUAIL and its Information Flow Model

In QUAIL, programs consist of the declaration of one or more variables followed by any number of commands according to the grammar of a custom imperative language (Biondi et al., 2013b, Appendix B). The core commands in the language perform deterministic variable assignment, probabilistic variable assignment (which assigns the value 0 to a variable with some user-specified probability p and the value 1 with probability $1-p$), branching, and jumping to other user-specified commands in the program. Although the language does not formally feature a looping construct, the authors indicate that programs containing loops can be transformed into equivalent commands in the core language; this feature is provided in the tool, with the program being trans-

[1] In this section we evaluate version 1.0 of QUAIL, available at `https://project.inria.fr/quail/`.

formed into in the core grammar in a preprocessing stage. As in CH-IMP, all variables' values are integers. Unlike CH-IMP, QUAIL also supports high-level data types such as arrays; these are also transformed into constructs in the core grammar in the same preprocessing stage.

A key difference between QUAIL's and CH-IMP's information flow models is their treatment of high-security and low-security information: in QUAIL, variables are explicitly labelled as "secret", "public" or "observable" at declaration-time, and these labels persist throughout execution of the program; an attacker can inspect the value of an observable variable at any time, while the values stored inside public variables are shielded from the attacker. As we have shown in Chapter 4, CH-IMP has a more generalised information flow model: any number of variables can have their values marked as either "secret" or "observable" at arbitrary points in a program, and in any order; those markings persist for only as long as the command is executed, however, and CH-IMP does not prevent a variable whose value was once marked as secret from being marked as observable at some later point.

QUAIL is based on a Markov decision process (MDP) semantics similar to CH-IMP's DTMC semantics: states in the MDP are (pc, L, H) tuples, where pc is the program counter (*i.e.* a pointer to the command currently being executed), L is a function mapping public variable names to their current values, and H is the range of potential values that the secret variable could have; these ranges may be partitioned further either due to the attacker's prior knowledge of the program's behaviour or the knowledge they gain as the program executes. QUAIL's attacker model thus assumes an attacker with partial information of the program's state: the program's variables are partitioned according to their secrecy level, and the attacker is only able to discriminate between states in the MDP by knowing the value of H in those states. This symbolic approach is the largest difference between the information flow models of QUAIL and CH-IMP, and it allows QUAIL to quantify information flows in programs with a very large secret space, something that CH-IMP would be unable to achieve in polynomial time.

When QUAIL programs terminate, the MDP enters an *absorbing state* analogous to CH-IMP's notion of an accepting state. Non-determinism in the MDP is resolved (and the MDP is thus transformed into a Markov chain) by computing probability distributions over the possible values of the secret variable in each state using the attacker's prior knowledge of the program. QUAIL then "hides" states in the Markov chain that the attacker is unable to observe by removing them from the Markov chain's probability transition matrix and redistributing their probabilities amongst their succeeding states; this "hiding" procedure is usually applied to all but the initial and absorbing states, and Biondi et al. estimate that the resulting *observable reduction* usually consists of fewer than 10% of the states of the original Markov chain. Three discrimination relations — *observer*, *secret* and *joint* — are computed, as described earlier in Section 3.2.8 (p. 60); these are themselves Markov chains derived from the observable reduction, and the program's information flow is defined in terms of the entropies of these Markov chains.

Although QUAIL and `chimp` have different information flow models, we can benchmark each tool and compare their performance by selecting an appropriate program that (*a*) is proven *not* to leak information and (*b*) contains some parameter that alters the complexity of the program; by varying this parameter, we can compare how quickly and efficiently each tool verifies that the program is secure. A *DC-net* implementation is a good example of such a program, and we shall use one as part of our comparison (indeed, Biondi et al. themselves offer a DC-net implementation as a case study for QUAIL); we first describe the purpose of and security guarantees provided by DC-nets.

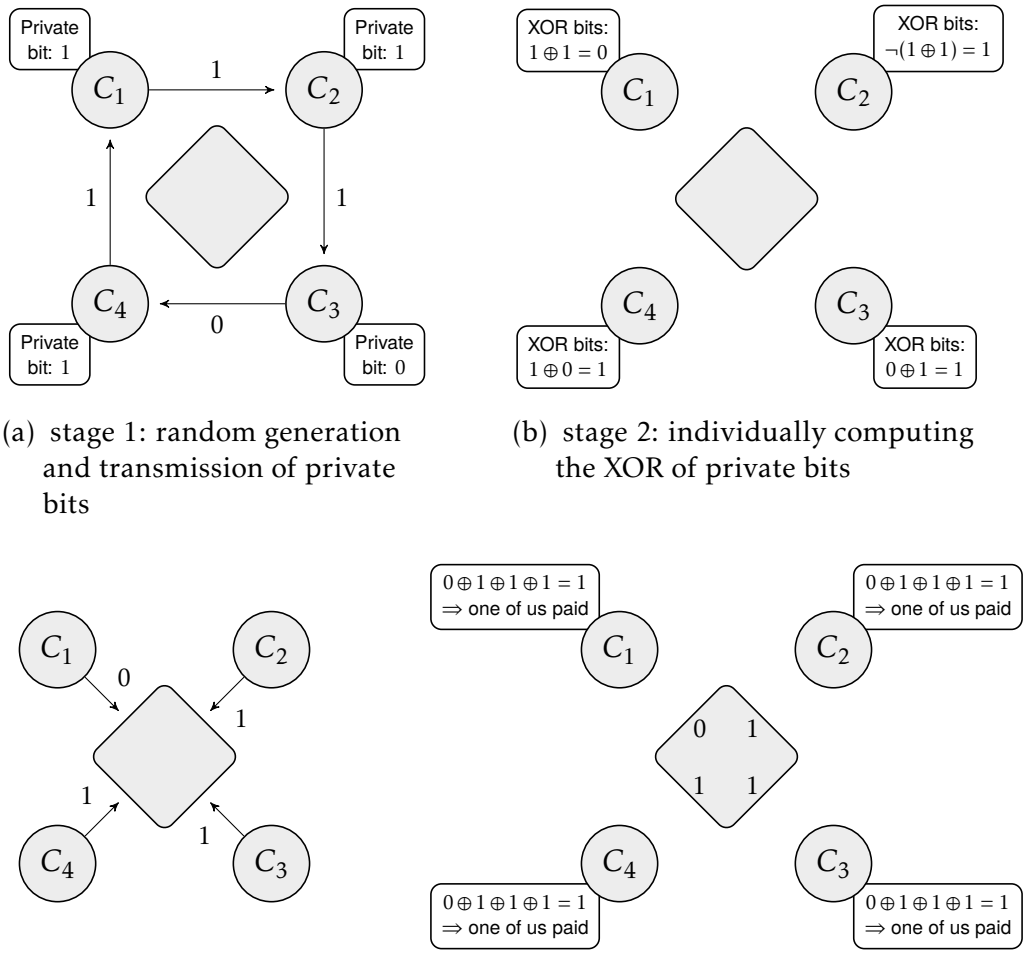
5.2.2 The Dining Cryptographers Problem and DC-Nets

The *dining cryptographers problem* was first posed by Chaum (1988), and investigates how anonymity can be guaranteed during secure multi-party computation. It is often informally described as follows: a group of cryptographers dine at a restaurant, and

at the end of the meal the waiter informs them that the bill has already been paid. The cryptographers notice a national security agent sitting at an adjacent table, and wonder whether the bill was surreptitiously paid by him. They do not wish to discuss the matter publicly with the agent or each other, but still want to know whether the bill was paid by one of their own; how do the cryptographers collectively discover whether one of them paid the bill while respecting their fellow cryptographers' right to anonymity?

The *DC-net* is a solution to the dining cryptographers problem; it provides uncon-

Figure 5.2: the four stages of the DC-net protocol; in this demonstration, the cryptographer C_2 is the bill-payer



(a) stage 1: random generation and transmission of private bits

(b) stage 2: individually computing the XOR of private bits

(c) stage 3: announcing output of XOR computation

(d) stage 4: verification of result

ditional sender and recipient untraceability. The cryptographers all participate in the DC-net, and together execute the following four-stage protocol, depicted in Figure 5.2:

1. Each cryptographer individually generates a random bit visible only to them and the cryptographer to their left, giving each cryptographer sight of two separate randomly-generated bits (Figure 5.2(a)).

2. Each cryptographer computes the XOR of their own bit and the bit shown to them by the cryptographer to their right (Figure 5.2(b)).

3. Each cryptographer in turn announces the result of this XOR computation publicly to the other cryptographers — except for the bill-payer (if any), who announces the *inverse* of their XOR computation (Figure 5.2(c)).

4. The XOR of all of the announcements allows each cryptographer to independently verify whether one of the cryptographers paid the bill: if the XOR result is 1, one of the cryptographers paid; if it is 0, nobody claimed to have paid — and therefore the cryptographers know that the national security agent paid instead (Figure 5.2(d)).

Given that the DC-net is an anonymity protocol whose purpose is to protect its bill-paying participant (if any), the secret information in a DC-net is the identity of the bill-paying cryptographer, and its security policy states that *no* information about the bill-paying cryptographer's identity should flow to any message that is broadcast to the rest of the DC-net; if this happens, the DC-net is insecure. (Since the national security agent does not participate in the four-stage protocol described above, the DC-net does *not* protect the national security agent's anonymity in the event that they paid the bill; however, given that the cryptographers' announcements themselves reveal whether or not the national security agent paid, this lack of protection is intentional and therefore *not* considered to be an information leak.) Provided that the generation of bits in stage 1 is truly random, the security policy is satisfied: it is not possible for

any of the participating cryptographers or any external entity (such as the national security agent) to discover the identity of a cryptographer who announced the inverse of their XOR computation in stage 3. This anonymity property holds regardless of the number of cryptographers who participate in the DC-net.

DC-net implementations are ideal for benchmarking QUAIL and `chimp`: by increasing the number of cryptographers participating in the DC-net, the complexity of the protocol increases, but the information flow from the bill-payer's identity to the collection of announcements by the cryptographers remains constant (*i.e.*, there is none). We shall therefore compare the performance of both QUAIL and `chimp` by providing them with implementations of DC-nets with increasing numbers of cryptographers.

5.2.3 Verifying the Security of DC-Nets in QUAIL and `chimp`

Listing 5.2 shows an implementation of a three-cryptographer DC-net in QUAIL; Listing 5.3 (p. 136) shows the equivalent implementation in `chimp`. Both implementations assume that the bill-payer is one of the cryptographers, and that each cryptographer is equally likely to pay the bill from the perspective of a passive observer. This could be construed as the scenario in which the national security agent, who has no prior knowledge of the likelihood of each cryptographer paying the bill, chose not pay the bill themselves — and therefore knows that one of the cryptographers must have — and is interested in learning the identity of the bill-paying cryptographer by observing the execution of the four-stage DC-net protocol.

The number of cryptographers participating in the DC-net is set on line 1. (By varying this integer, the complexity of the DC-net can be modified: larger DC-nets contain more cryptographers, and thus more private communication between and public announcements by cryptographers.) Next, the bill-payer's identity is selected from a uniform probability distribution; this is the only secret that occurs in either program. In the `chimp` program, the value of the variable storing the bill-payer's identity is explicitly marked as a secret at this point; in the QUAIL program, the secrecy of the

Listing 5.2: a QUAIL implementation of a three-cryptographer DC-net

```
1  const cryptographers := 3;
2  secret int32 billpayer := [1,cryptographers];
3  public int1 mybit := 0;
4  public int1 theirbit := 0;
5  public int1 firstbit := 0;
6  public int32 i := 1;
7  observable array [cryptographers] of int1 announcements;
8  random firstbit := randombit(0.5);
9  assign mybit := firstbit;
10 while (i < cryptographers) do
11   random theirbit := randombit(0.5);
12   assign announcements[i] := mybit ^ theirbit;
13   if (billpayer == i) then
14     if (announcements[i] == 0) then
15       assign announcements[i] := 1;
16     else
17       assign announcements[i] := 0;
18     fi
19   fi
20   assign i := i + 1;
21   assign mybit := theirbit;
22 od
23 assign theirbit := firstbit;
24 assign announcements[0] := mybit ^ theirbit;
25 if (billpayer == i) then
26   if (announcements[0] == 0) then
27     assign announcements[0] := 1;
28   else
29     assign announcements[0] := 0;
30   fi
31 fi
32 return;
```

Listing 5.3: a `chimp` implementation of a three-cryptographer DC-net

```
new cryptographers := 3;
new billpayer := { 1 -> 1/3, 2 -> 1/3, 3 -> 1/3 };
secret billpayer;
new firstbit := { 0 -> 0.5, 1 -> 0.5 };
new mybit := firstbit;
new theirbit := 0;
new i := 1;
while (i < cryptographers) {
  theirbit := { 0 -> 0.5, 1 -> 0.5 };
  new announcement := mybit xor theirbit;
  if (i == billpayer) {
    if (announcement == 0) { announcement := 1; }
    else { announcement := 0; }
  }
  observe announcement;
  i := i + 1;
  mybit := theirbit;
}
theirbit := firstbit;
new finalannouncement := mybit xor theirbit;
if (i == billpayer) {
  if (finalannouncement == 0) { finalannouncement := 1; }
  else { finalannouncement := 0; }
}
observe finalannouncement;
```

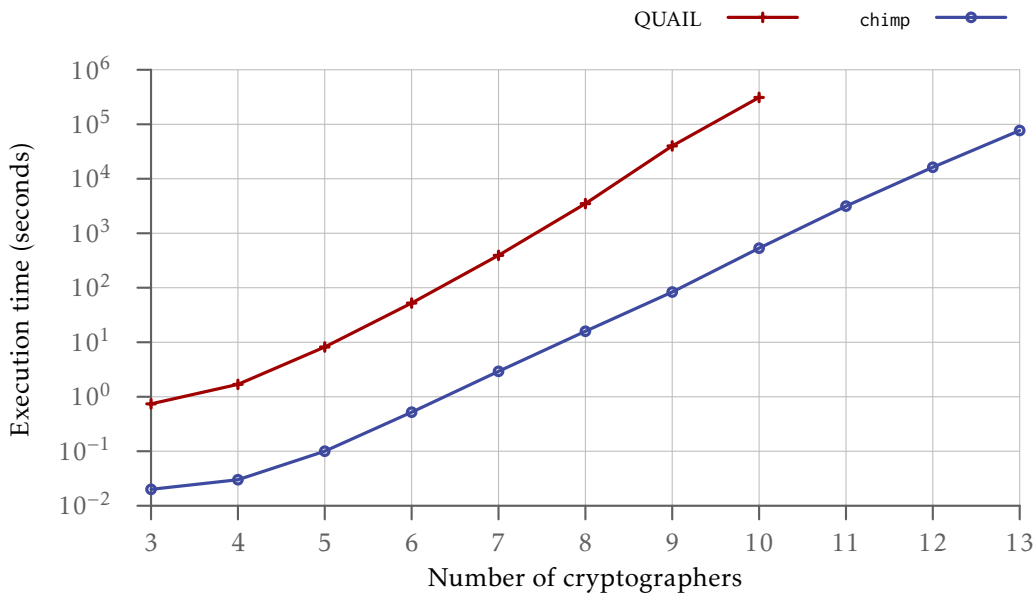
variable's value forms part of its type at declaration-time. The programs then iterate over the cryptographers, performing stages 1–3 of the DC-net protocol described in Section 5.2.2 for each cryptographer in a single iteration: the cryptographer's own randomly-generated bit is assigned to the variable `mybit`, and the bit ostensibly generated by the cryptographer to their immediate right is assigned to the variable `theirbit` (before the next iteration, the value of `theirbit` is copied into `mybit`). In the `chimp` program, the result of the XOR computation (or the inverse, if the cryptographer is the bill-payer) is announced using the `observe` command, revealing the value to the attacker; in the QUAIL program, the value is written to an index of the `announcements` array, whose contents are visible to the attacker at all times.

To evaluate the performance of each tool, the number of cryptographers was varied between 3 and 14 in each implementation; QUAIL and `chimp` were then tasked with verifying that no information flows from the identity of the bill-payer to the public announcements by the cryptographers in each DC-net. Each execution of each tool[2] was profiled for its total execution time (in wall-clock seconds) and its peak memory usage (specifically, the maximum resident set size of the process). Upon termination, both tools verified that the information flow from the secret values to the observable values in each DC-net was 0 bits; both tools therefore provided the expected results.

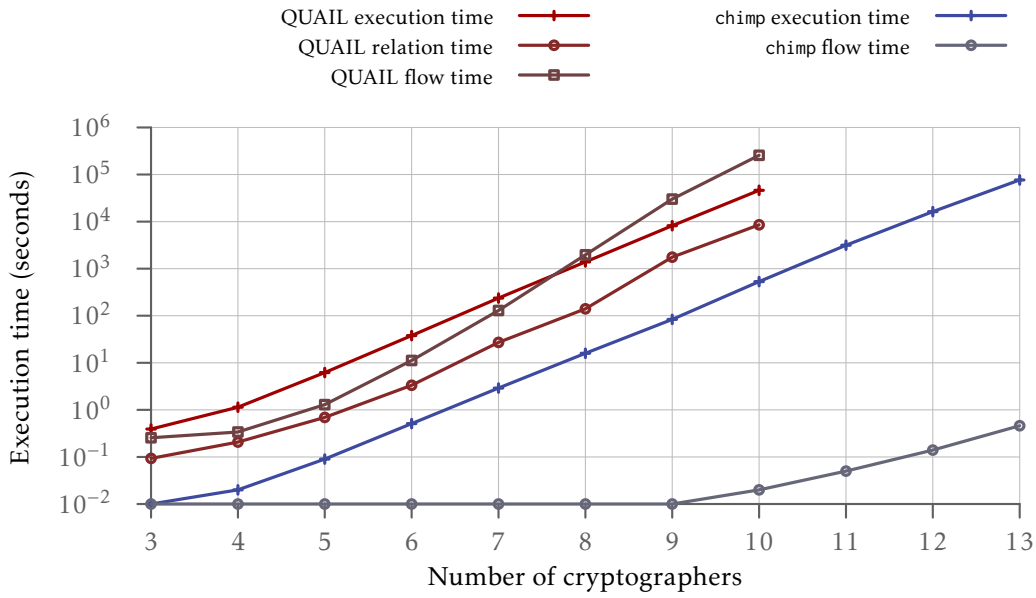
A comparison of the total time taken for both tools to verify each DC-net is shown in Figure 5.3(a) (p. 138). For small DC-nets (*i.e.* those containing up to six cryptographers), the performance of both tools is reasonable: they both verify that each DC-net is secure in under two minutes. For DC-nets containing more than six cryptographers, there is a larger difference in performance: QUAIL's execution time increases sharply as the exponential increase in the MDP's state space becomes more significant; this is consistent with the findings of Biondi et al. (2013b, Section 4), who state that QUAIL's analysis becomes computationally expensive for DC-nets containing more than 7 cryptographers. In our evaluation, it was unable to verify the security of an 11-

[2]Both tools were benchmarked on a 64-bit Linux system with a quad-core 3.2GHz processor and 4GB of memory; each was given full use of the system's resources.

Figure 5.3: the time taken by QUAIL and `chimp` to analyse DC-nets containing varying numbers of cryptographers



(a) total execution time of each tool



(b) execution time of each tool, broken down by activity

cryptographer DC-net in under 100 hours. chimp exhibits similar behaviour, although at a slower rate; it can efficiently analyse the anonymity of DC-nets of up to 13 cryptographers, failing to verify the anonymity of a 14-cryptographer DC-net within 100 hours.

Figure 5.3(b) offers some insight into why this might be the case; this graph shows the amount of time taken by each tool to (*a*) execute the DC-net program by exploring its state space until it terminates, (*b*) compute the three discrimination relations (for QUAIL only), and (*c*) quantify the information flow that occurs in the program. While both tools take an exponentially long time to execute the program, chimp is able to quantify the information flow that occurs in the program using the algorithms of Section 5.1.3 (p. 122) exceptionally quickly. In contrast, QUAIL takes almost as much time to quantify the information flow from the discrimination relations as it does to execute the program, and our evaluation shows that, for DC-nets containing more than 8 cryptographers, it is likely to take *more* time; the computation of the discrimination relations themselves is a similarly long process. Each of the three stages of QUAIL's analysis takes longer to complete than the entirety of chimp's analysis.

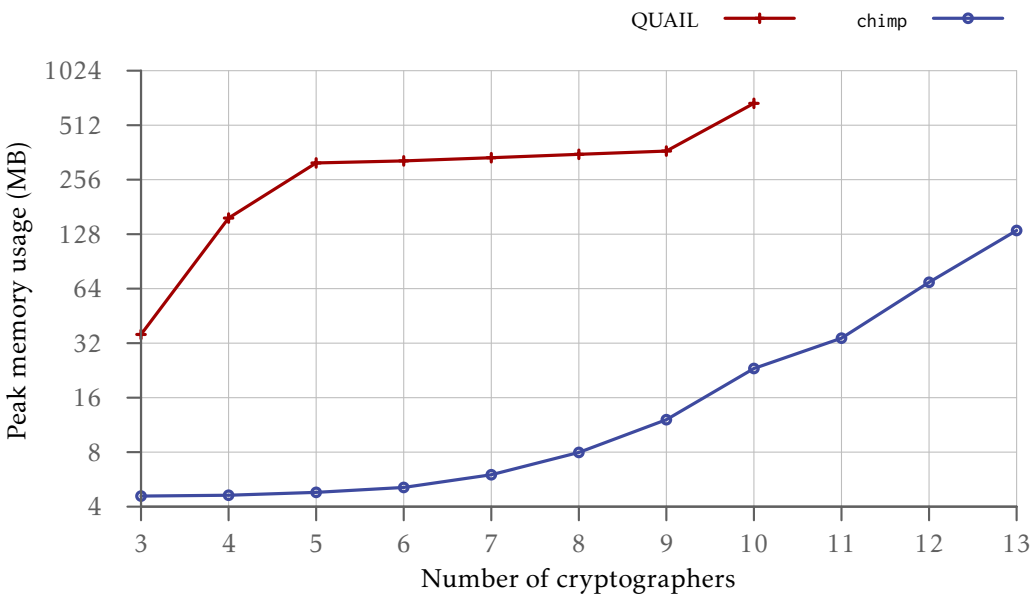
A comparison of the peak memory usage of each tool when verifying each DC-net is shown in Figure 5.4 (p. 140); in all cases, chimp consumes significantly less memory than QUAIL. This is likely to be due to QUAIL's need to store its observable reduction and three discrimination relations in memory simultaneously; CH-IMP's algorithms discard all but the necessary information to quantify the information flow of the program while it executes.

As described in Section 5.1.1 (p. 114), chimp implements the environment function as a list. Whenever two previously-partitioned environment functions are merged (*i.e.* following the execution of an if or while command), chimp must iterate over both lists and sum the probabilities of environments that appear in both lists when constructing the succeeding environment function, a process that is similar to the summing operation for environment-modifying commands described in Section 5.1.1. This is a costly

process when the partitions are large, as they are likely to be when the DTMC represented by the environment function contains many states. The semantic rules in the formal definition of CH-IMP (Definition 4.6, p. 105) make it clear that the number of unique states of the program (*i.e.* the number of secret values, observable values and variables in scope) is determined by the number of probabilistic declarations and assignments that occur, because only those commands cause the "branching" behaviour in the DTMC's state space; the limiting factor of the tool (and CH-IMP itself), therefore, is not the length of the program being analysed, but the number of probabilistic declarations and assignments that are made in it. This explains `chimp`'s poor performance when analysing larger DC-nets: an exponentially greater number of environments of an ever-increasing size must be processed when the partitioned environment functions are merged on lines 14, 18 and 24 of Listing 5.3 (p. 136).

We stated the differences between QUAIL's and CH-IMP's information flow models in Section 5.2.1 (p. 129), and in particular we emphasised that QUAIL's symbolic approach allows it to analyse programs with enormous secret spaces (something that

Figure 5.4: the peak memory usage of QUAIL and `chimp` while analysing DC-nets containing varying numbers of cryptographers



chimp would be incapable of doing within a reasonable time). The purpose of performing this evaluation is *not* to argue that chimp is a better information flow analysis tool — after all, it is trivial to find an example that QUAIL would successfully analyse and chimp would not — but that chimp's performance is comparable to that of a contemporary information leakage analysis tool from the literature when given an example that can be analysed by both tools.

5.2.4 Improving the Performance of chimp

Although chimp's performance is acceptable, it could be optimised to further improve its execution time or memory usage. We propose two such optimisations.

A CH-IMP security policy states an upper bound on the amount of information that it is acceptable for a program to leak. The information leakage measures we utilise in this thesis — mutual information and min-entropy leakage — are non-decreasing as execution proceeds. This fact could be exploited to improve the speed with which chimp can verify whether a program conforms to a given security policy: after executing a secret or observe command (thus triggering a modification to $\mathcal{S}$ or $\mathcal{O}$ respectively), chimp could compute the desired information leakage measure using the algorithms described in Section 5.1.3 (p. 122) and immediately terminate the program if the measure were to exceed the upper bound imposed by the security policy, because the amount of information leaked by the program could never fall below the upper bound at a later point during execution and once again satisfy the security policy. Clearly, this optimisation would be particularly valuable in situations where the program exceeds the upper bound early in its execution. However, given that the time complexity of the algorithms of Section 5.1.3 is a function of the number of unique secret and observable values that have occurred during execution, it may not be beneficial to perform this operation after the execution of *every* secret and observe command if the path of execution contains a large amount of secret or observable information.

The time complexity of the probability-summing operation on environment func-

tions described in Section 5.1.1 (p. 114) is a function of the number of bound variables at the point at which the operation occurs. Although this operation is made more efficient with the appropriate use of CH-IMP's variable scoping feature (which minimises the number of bound variables at any given point during execution), its efficiency could be improved further by automatically removing variable bindings from σ when it can be ascertained that they are no longer required; this could be achieved with the addition of a CH-IMP command that removes a particular variable binding from all environments in the current environment function, along with a pre-execution static analysis of the program in which instances of this command are inserted at appropriate points in order to destroy dead variables. This optimisation is reminiscent of the data-flow analyses performed by contemporary programming language compilers, and has the potential to significantly improve the execution time and memory usage of CH-IMP programs containing variables that are defined in long-lived scope frames and whose values are only used shortly after the scope frame is created. However, extreme care would need to be taken when implementing this optimisation: a naive implementation would most likely be detrimental to execution time, since the act of removing a variable binding would itself require the environments in the environment function to be enumerated; indiscriminately destroying variable bindings shortly before the scope frame containing them would have been destroyed anyway would counteract the beneficial effects of eliminating dead variables from the environment function, particularly if the domain of the environment function were very large.

5.3 Summary

In this chapter, we have presented `chimp`, an implementation of the quantitative point-to-point information flow model we introduced in Chapter 4. `chimp` shows that our information flow model is practical: probabilistic programs and protocols can be writ-

ten in the CH-IMP syntax and successfully analysed with the tool, as we demonstrated with the DC-net example in Section 5.2.3 (p. 134). We shall analyse the security of some more complex CH-IMP programs in the following chapter.

Furthermore, we have compared `chimp` with QUAIL, an implementation of Biondi et al.'s (2013a) information flow model, by encoding a DC-net in each tool's language and verifying how quickly and efficiently each tool confirms that there is no flow of information from the identity of the bill-paying cryptographer to the public announcements made by the cryptographers; by varying the number of cryptographers in the DC-net, we can determine the performance of each tool as the size and complexity of the system model scales up. We have shown that `chimp`'s performance is comparable to that of QUAIL.

There are, however, limitations to `chimp`'s ability to analyse probabilistic programs. Improvements can be made to the execution speed of CH-IMP programs in `chimp`, but, regardless of the efficiency of the implementation, the execution time of `chimp` is still ultimately exponential; we saw that it was unable to analyse a DC-net containing 14 cryptographers within 100 hours. This is due to the exponential number of states that exist in a probabilistic program as more probabilistic declarations and assignments are made. Thus, while `chimp` is ideal for analysing probabilistic programs and protocols of low to moderate complexity, it is not well-suited to analysing highly-complex systems in which enormous numbers of events occur with minute probabilities.

6

CH-IMP Case Studies

We have thus far presented CH-IMP, our quantitative information flow model for probabilistic systems, and `chimp`, an implementation of this model as a software tool for analysing CH-IMP programs. Although it is clear that the CH-IMP language is not suitable for general-purpose programming, provided that a system or protocol can be encoded using the CH-IMP syntax, `chimp` may nevertheless be used to analyse the information flows that occur in systems where the state space does not explode exponentially, as in the larger DC-nets we saw in Section 5.2.3 (p. 134). In this chapter, we present two such probabilistic systems that leak information, and analyse them using the `chimp` tool.

In the first case study, a DC-net in which the cryptographers' bit generation procedure is not truly random (*e.g.* as a result of accidental poor practice on behalf of all cryptographers in the DC-net) causes some leakage of a bill-paying cryptographer's identity to a passive attacker observing the public communication that occurs in the DC-net. We shall see that the precise amount of information leaked depends on the unfairness of the cryptographers' bit generation procedure. We shall also show, however, that a single rogue cryptographer attempting to force poorly-randomised bits upon other cryptographers to violate the DC-net's anonymity *ab intra* is unable to compromise the bill-paying cryptographer's anonymity, provided that the bit generation procedure used by the remaining cryptographers is completely fair.

In the second case study, we investigate the use of a pseudorandom number generation algorithm known as the *linear congruential generator* in a two-player playing card game, and see how the poor design of such an algorithm may lead to low-quality randomness and thus significant information leaks about the game's state to individual players. We demonstrate how the CH-IMP information flow model makes it convenient to model players with different levels of knowledge about the game's state, and how the `chimp` tool can be used to precisely quantify the information leaks that may bias the game in favour of or against particular players.

The presentation of both of these case studies follows a common structure: we introduce the scenario, present a CH-IMP program that models the protocol or system being analysed, and compute the information flow from its secrets to its observable information using `chimp`. We characterise this flow of information using the measures of mutual information and min-entropy leakage. We then vary different parameters in the CH-IMP program to investigate how they affect the flow of information.

It is important to note that, in both of these case studies, `chimp` quantifies the flow of information between arbitrary numbers of secret and observable values, but does not indicate *why* this information flow occurs. It is the role of `chimp`'s user to determine why a CH-IMP program appears to leak information — although `chimp` contains additional features that make this task easier, including providing a tabular representation of the joint probability distribution of the secret and observable values that occur during execution of the program, similar to the ones we constructed manually for our examples motivating CH-IMP's information flow model in Section 4.3 (p. 86).

As with Chapters 4 and 5, the case studies presented in this chapter are based primarily on one of our contributions to the literature (Chothia et al., 2013b), although we describe them in much greater depth in this chapter.

6.1 Violating the Anonymity of DC-Nets

In Section 5.2.2 (p. 131), we motivated and described the behaviour of the *DC-net*, a four-stage communication protocol facilitating the multi-party computation of one bit of information using the XOR function, providing unconditional sender and recipient untraceability. We briefly mentioned that the random bit generation by each cryptographer in the first stage must be truly random, otherwise a passive attacker may be able to compromise the anonymity of a bill-paying cryptographer. Making the same assumptions that we made in Section 5.2.2 — specifically, that the bill-payer is one of the cryptographers and that each cryptographer is equally likely to pay the bill from the perspective of a passive observer — we now use `chimp` to measure the reduction in the bill-payer's anonymity that occurs when the bits are not truly randomly generated for a variety of reasons. We use mutual information to quantify what a passive attacker learns about the bill-payer's identity when observing the announcements made by the cryptographers, and min-entropy leakage to quantify how likely it is that, having observed these announcements, their best strategy for guessing the identity of the bill-payer in a single attempt will succeed.

6.1.1 Faulty Random Bit Generation

A DC-net provides complete anonymity to the bill-payer because, from the perspective of each cryptographer, all of the other cryptographers are equally likely to announce 0 or 1 as the result of their XOR computation. Each cryptographer generates a random bit (containing 1 bit of Shannon entropy) and XORs it with the random bit of another cryptographer (containing another 1 bit of Shannon entropy); the value of each randomly-generated bit is known only by two cryptographers, and each cryptographer is the only one with first-hand knowledge of the two input bits to their own XOR function, so there is always 1 bit of Shannon entropy in the XOR output — when the bill-payer negates the value of this bit before announcing it publicly, there is there-

fore no externally observable indication that the negation has taken place. However, if others have *some* knowledge of the two input bits to another cryptographer's XOR function, the bill-payer's act of negating their XOR output becomes more noticeable, because there is no longer 1 bit of entropy in their XOR output; the severity of the compromise depends on the others' precise knowledge of the two XOR input bits.

The program shown in Listing 6.1 — based on the program shown in Listing 5.3 (p. 136) — models a DC-net consisting of four cryptographers where each cryptographer is equally likely to be the bill-payer (thus giving the program's secret 2 bits of Shannon entropy), but all of the bits generated by the cryptographers are biased toward 0 with some probability p. Implementing this bias requires minimal changes to be made to Listing 5.3: whenever values are assigned to the variables *mybit* and

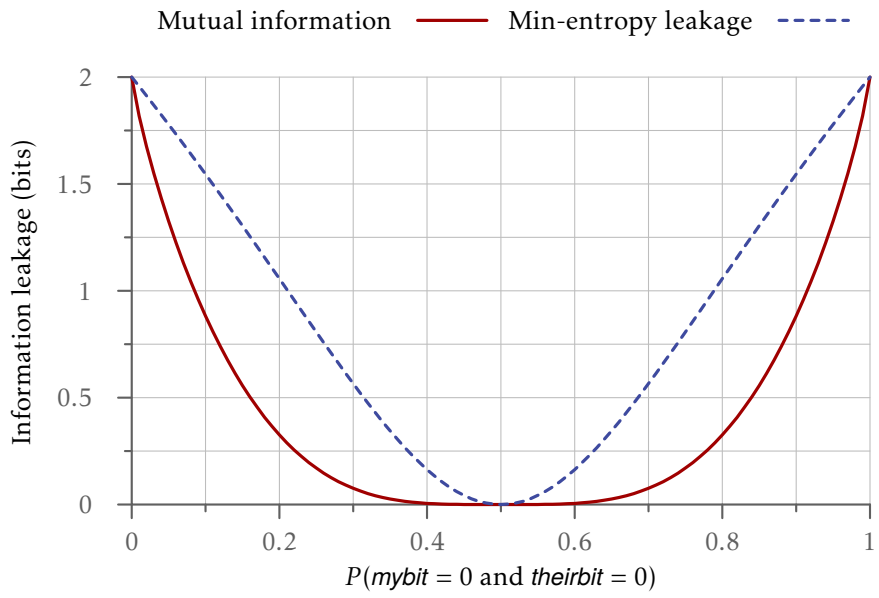
Listing 6.1: an incomplete CH-IMP DC-net implementation in which the cryptographers generate random bits that are biased toward 0 with a chosen probability p; the program can be completed by fixing a value of p, and setting ○ to p and ● to $(1-p)$

```
1  new cryptographers := 4;
2  new billpayer := { 1 → 0.25, 2 → 0.25, 3 → 0.25, 4 → 0.25 };
3  secret billpayer;
4  new firstbit := { 0 → ○, 1 → ● };
5  new mybit := firstbit;
6  new theirbit := 0;
7  new i := 1;
8  while (i < cryptographers) {
9     theirbit := { 0 → ○, 1 → ● };
10    new announcement := mybit xor theirbit;
11    if (i == billpayer) {
12       if (announcement == 0) { announcement := 1 }
13       else { announcement := 0 }
14    };
15    observe announcement;
16    i := i + 1;
17    mybit := theirbit
18 };
19 theirbit := firstbit;
20 new finalannouncement := mybit xor theirbit;
21 if (i == billpayer) {
22    if (finalannouncement == 0) { finalannouncement := 1 }
23    else { finalannouncement := 0 }
24 };
25 observe finalannouncement
```

theirbit, the probability distributions simply need to be weighted appropriately. The secret and observable information remain the same: the secret is the identity of the bill-payer on line 3, and the observable values are the XOR computation results announced by each cryptographer on lines 15 and 25. CH-IMP therefore measures the information flow from the bill-payer's identity to the public announcements made by the cryptographers.

Figure 6.1 graphs the information leakage from the bill-payer's identity to the cryptographers' announcements in this program, in terms of both mutual information and min-entropy leakage. When the generation of bits is fair (*i.e.* $p = 0.5$), there is no information shared between the bill-payer's identity and the cryptographers' announcements; however, as the variables' values become progressively more biased in favour of 0, it becomes much more likely that a non-paying cryptographer will announce $0 \oplus 0 = 0$ and the bill-payer will announce $0 \oplus 0 \oplus 1 = 1$. The same is true as the bias moves in favour of 1, making the graph symmetrical: non-paying cryptographers are more likely to announce $1 \oplus 1 = 0$, and bill-payers are more likely to announce

Figure 6.1: the information leakage from the program in Listing 6.1 for various probability distributions over the values of *mybit* and *theirbit*



$1 \oplus 1 \oplus 0 = 1$. Both leakage measures converge on 2 bits as generation of the random bits becomes increasingly deterministic: if the DC-net contains the bill-payer, it is guaranteed that their announcement will be the inverse of the others.

The attacker is only able to observe the announcements made by each cryptographer, but given that CH-IMP's attacker model also assumes that the attacker knows the behaviour of the system, they also know the bias present in the bit generation; the attacker's best strategy, therefore, is to identify the cryptographer whose announcement disagrees with those made by the others and assume that this cryptographer is the bill-payer. This strategy quickly begins to pay off: when $p = 0.65$ the amount of mutual information is only 0.03 bits, but the min-entropy leakage measure reveals that the vulnerability of the bill-payer's identity to this guessing attack increases by a factor of around 1¼.

6.1.2 Insider Attacks on DC-Nets

So far we have only considered external passive attacks against the DC-net: the attacker has only been able to observe the announcements by the cryptographers. If the attacker is one of the non-paying cryptographers, and thus has the ability to directly influence one of the random bits used by another cryptographer, are they able to compromise the anonymity of the DC-net?

We can model this scenario in CH-IMP by making minor changes to Listing 6.1. Assuming that the bill-payer is the first cryptographer to announce the result of their XOR computation, we (*a*) assign the value 1 a probability of 0 in the declaration of *billpayer* on line 2 and uniformly redistribute the other probabilities, indicating that the first cryptographer is never the bill-payer (reducing the entropy of the secret information in the program from 2 to around 1.59 bits); (*b*) assign the value 0 to *firstbit* in its declaration on line 4, to make the malicious cryptographer's bit deterministic (this bit will later be used by the second cryptographer); and (*c*) mark the values of *firstbit* on line 4 and *theirbit* on line 6 as observable using the observe command, since the cryptog-

rapher knows both the bit they generated and the bit generated by the cryptographer to their right. Listing 6.2 contains the updated CH-IMP code.

Initially, we assume that the bits generated by the other cryptographers are truly random. Figure 6.2 (p. 151) depicts the DC-net from the perspective of the malicious cryptographer. It is clear that they do not have enough information to violate the anonymity of the bill-payer: that would require *some* knowledge of the private bits generated by the cryptographers other than the one to their immediate right, who exposes their own private bit to the malicious cryptographer anyway. The mutual information of the bill-payer's identity and the values observed by the malicious cryptog-

Listing 6.2: an incomplete CH-IMP DC-net implementation in which the first cryptographer is an insider attempting to violate the anonymity of the DC-net by forcing a chosen bit onto the second cryptographer, and in which the other cryptographers generate random bits that are biased toward 0 with a chosen probability p; as with Listing 6.1 (p. 147), the program can be completed by fixing a value of p, and setting ○ to p and ● to $(1-p)$

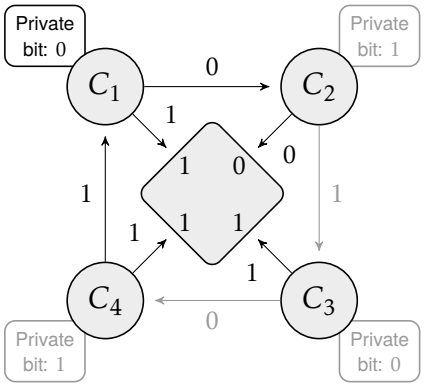
```
1  new cryptographers := 4;
2  new billpayer := { 2 → 1/3, 3 → 1/3, 4 → 1/3 };
3  secret billpayer;
4  new firstbit := 0;
5  new mybit := firstbit;
6  new theirbit := 0;
7  observe firstbit;
8  observe theirbit;
9  new i := 1;
10 while (i < cryptographers) {
11    theirbit := { 0 → ○, 1 → ● };
12    new announcement := mybit xor theirbit;
13    if (i == billpayer) {
14       if (announcement == 0) { announcement := 1 }
15       else { announcement := 0 }
16    };
17    observe announcement;
18    i := i + 1;
19    mybit := theirbit
20 };
21 theirbit := firstbit;
22 new finalannouncement := mybit xor theirbit;
23 if (i == billpayer) {
24    if (finalannouncement == 0) { finalannouncement := 1 }
25    else { finalannouncement := 0 }
26 };
27 observe finalannouncement
```

rapher is 0 bits; therefore, when the other private bits are truly randomly-generated, DC-nets are fault-tolerant in the presence of a single malicious cryptographer.

We now consider the scenario in which the bits generated by the other cryptographers are biased toward 0 with some probability p. The malicious cryptographer's best strategy for identifying the bill-payer remains the same as the passive attacker's strategy in Section 6.1.1: assume that the bill-payer is the cryptographer whose announcement disagrees with those made by the others. However, since one of the private bits — the one under the control of the malicious cryptographer — is *already* deterministic, the DC-net leaks information more quickly as p diverges from 0.5.

Figure 6.3 graphs the information leakage that occurs from the program as p is varied between 0 and 1. As in Figure 6.1 (p. 148), the mutual information varies between 0 bits when generation of the bits is fair (*i.e.* $p = 0.5$) and its maximum possible value when generation of the bits is deterministic (*i.e.* their values are guaranteed to be either 0 or 1). However, as the random bit generation becomes even slightly biased, the malicious cryptographer's knowledge of the bill-payer's identity increases much faster than that of the passive attacker in Section 6.1.1: for instance, when $p = 0.6$, the first derivative of the continuous function representing the mutual information with respect to p is 0.3 in Figure 6.1, but 1.1 in Figure 6.3. The strategy is therefore much

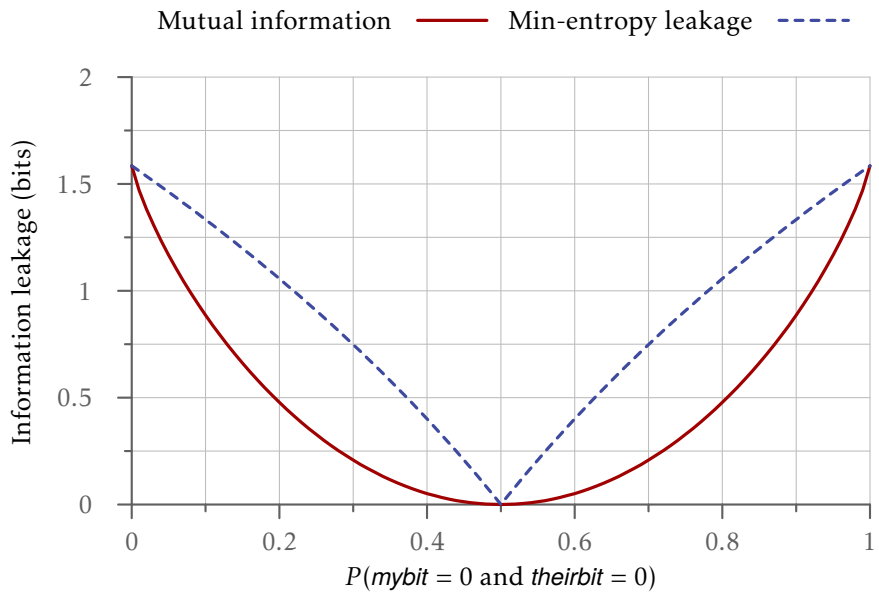
Figure 6.2: a DC-net containing a malicious cryptographer C_1, with C_2 as the bill-payer; the information exposed to the malicious cryptographer is shown in black, and the information concealed from them is shown in grey



more effective for the malicious cryptographer than it is for the passive observer, as indicated by the min-entropy leakage plot in Figure 6.3: when $p = 0.6$, the identity of the bill-payer is already 1.3 times more vulnerable to a correct first-attempt guess, compared with 1.1 times for the bill-payer's identity in Section 6.1.1.

This case study demonstrates that DC-nets provide perfect anonymity for all participants — even if one of them is malicious — in the presence of passive attackers with knowledge of the system's behaviour and the ability to observe information exchanged over public channels, provided that the (honest) participants can draw upon true randomness when generating their private bits. As this randomness decreases, and the participants' private bits become more deterministic, the anonymity provided by the DC-net also decreases; the rate at which the anonymity decreases is more severe when the attacker becomes a participant in the DC-net.

Figure 6.3: the information leakage from Listing 6.2 for various probability distributions on the values of *mybit* and *theirbit*



6.2 Pseudorandom Number Generation

A *pseudorandom number generator* (PRNG) is an algorithm for deterministically generating sequences of numbers that are not, but nevertheless appear to be, random. They are of importance in fields such as cryptography, where they are commonly used in the generation of nonces and keys; in these security-critical situations, a PRNG must additionally pass the *next-bit test* (Yao, 1982): it should not be possible for an attacker armed only with knowledge of the first n bits of output from the PRNG to predict the $n+1$th bit of output in polynomial time. A PRNG that satisfies this additional property is known as a *cryptographically secure PRNG*.

6.2.1 Linear Congruential Generators

The *linear congruential generator* (LCG) is an example of a PRNG algorithm, implemented using modulo arithmetic. A particular instance of an LCG is defined by the recurrence relation

$$L_n \equiv (L_{n-1} \cdot a + c) \bmod m$$

where m, a and c are preselected, fixed parameters: m is the *modulus*, a positive integer; a is the *accumulator*, an integer in the interval $[2, m)$ that is designed to make the output of the multiplication operation much larger than m, thus triggering the modulo operation; and c is the *constant*, an integer in the interval $[0, a)$. Each output L_n of the LCG is defined in terms of the last using this recurrence relation; the special value L_0 is the *seed*, an integer in the interval $[0, m)$ that is used to initialise the LCG. Given a particular combination of m, a, c and L_0, it is possible to reconstruct the entire sequence of integers outputted by the LCG; m, a and c are static and it is assumed that they are known to an attacker if the design of the system using the LCG is known, so the seed is the only variable determining the sequence produced by the LCG and it is therefore crucial that it is selected uniformly from the interval $[0, m)$.

LCGs are widely used to implement pseudorandomness because they are easily understood (and therefore simple to implement), can have their state stored efficiently in memory, and because their use of modulo arithmetic allows them to generate sequences quickly on many computer architectures. Indeed, the `rand()` function available in the standard library of many programming languages is implemented using an LCG, and there is often an `srand()` function for setting the LCG's seed. For instance, the `java.util.Random` class in the Java API implements an LCG with the parameters $m = 2^{48}$, $a = 25214903917$ and $c = 11$; the seed L_0 is an optional argument to the class's constructor. When a `java.util.Random` object is called upon to produce a pseudorandom number, the succeeding state L_{n+1} is generated and the highest 32 bits of the succeeding state are returned to the caller.

LCGs must be designed and used with care, however: if the LCG parameters are not chosen carefully for the task at hand, or if the LCG's output is misused, the detrimental effect on the system's security can be enormous. To demonstrate this, we present a simple application of a linear congruential generator as a means of providing pseudorandomness: a playing card game, whose players will naturally be prepared to exploit any information they obtain in order to beat the other players.

6.2.2 A Playing Card Game

In this case study, the game is played with two players — Alice first, and Bob second — and proceeds as follows. A standard, shuffled 52-card deck of playing cards is revealed to both players; for the purposes of this game we assume that the order of cards in the deck is public information. The dealer privately communicates the position of a random card in the deck to each player in turn; each player should be assigned a different card in the deck, so the dealer must ensure that a different position in the deck is communicated to each player. The players may then choose to bet on the likelihood of the face value of their card being higher than that of the other player's card. If both players choose to bet, the dealer reveals the position of the card in the deck assigned

to each player; the one with the card having the highest face value wins the game (and the other player's wager).

Clearly, it is advantageous for one player to place a bet only when they can be reasonably confident of the face value of the other player's card; since the ordering of the deck is assumed to be public, it is therefore crucial that the two positions selected in the deck by the dealer are unrelated. It should also be noted that, even in the best-case scenario, each player will of course always learn *some* information about the other's card: specifically, that it is not the card at the position assigned to them by the dealer. The dealer must therefore ensure both that the method by which the deck positions are chosen passes the next-bit test, and that the generated numbers are used correctly.

For any given position of Alice's card in the deck, the ideal implementation of this game would ensure that the position of Bob's card is uniformly distributed over all 51 remaining positions, producing the joint probability distribution sketched in Table 6.1. The mutual information of the two deck positions would be $52 \cdot 51 \cdot (\frac{1}{2652} \log_2 \frac{1/2652}{1/52 \cdot 1/52}) \approx 0.0280$ bits — each player's knowledge that the position of their opponent's card in the deck is not the same as their own. The min-entropy leakage $\mathcal{L}_{XY}$ from the position of each player's card to that of their opponent's card would be $\log_2 \frac{1/51}{1/52} \approx 0.0280$ bits;

Table 6.1: the joint probability distribution of the secret and observable values that occur in the ideal implementation of the playing card game

	Observable value						
Secret value	0	1	2	⋯	49	50	51
0	0	1/2652	1/2652	⋯	1/2652	1/2652	1/2652
1	1/2652	0	1/2652	⋯	1/2652	1/2652	1/2652
2	1/2652	1/2652	0	⋯	1/2652	1/2652	1/2652
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮
49	1/2652	1/2652	1/2652	⋯	0	1/2652	1/2652
50	1/2652	1/2652	1/2652	⋯	1/2652	0	1/2652
51	1/2652	1/2652	1/2652	⋯	1/2652	1/2652	0

since the program would leak the minimum amount of information, each player's best strategy for correctly guessing the position of their opponent's card in one attempt would be to select one of the remaining positions at random, a strategy that would succeed on average once every $\frac{1}{2^{\mathcal{L}_{XY}}/52} = 51$ attempts.

We now examine a CH-IMP implementation of this game in which an LCG determines the face value of each player's card. We shall see how a lack of care in selecting appropriate LCG parameters and correctly using the output of the LCG causes a catastrophic effect on the game's fairness.

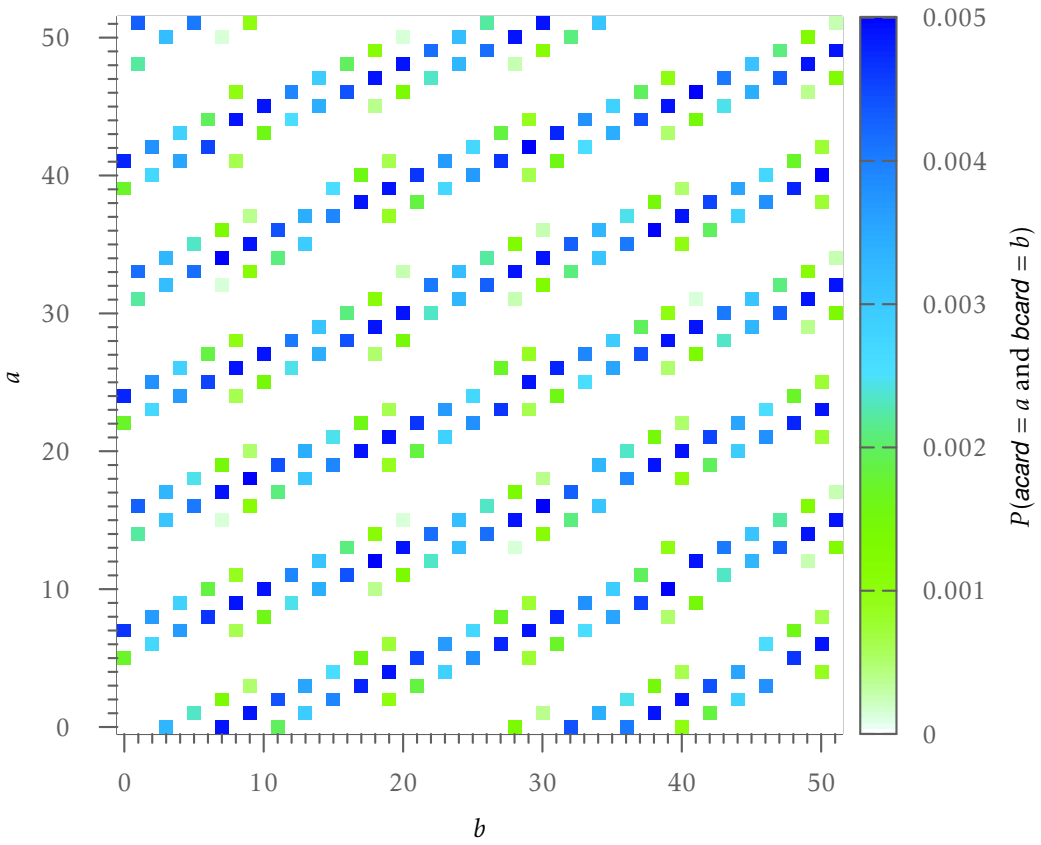
Listing 6.3 is a CH-IMP program modelling the card selection phase of the playing card game described above, as seen from the perspective of Alice; from her point of view, the observable information is the position of her own card in the deck (*acard*) and the secret information is the position of Bob's card (*bcard*). This implementation generates positions in the deck pseudorandomly using an LCG with parameters $m = 2^{13}$, $a = 789$, and $c = 366$, and chooses a seed *l0* uniformly from the set of all possible seeds (the expression [n_l .. n_h] on line 4 is a convenient syntax for specifying the uniform probability distribution over the integers in the interval $[n_l, n_h]$). The first pseudorandom number *l1* is generated by the LCG, and a simple modulo operation maps this random number to a position in the deck; this position (*acard*) is communicated to

Listing 6.3: CH-IMP implementation of the card selection phase of the playing card game, played from Alice's point of view: the observable information is Alice's card (*acard*), and the secret information is Bob's card (*bcard*)

```
1  new m := 8192;
2  new a := 789;
3  new c := 366;
4  new l0 := [ 0 .. 8191 ];
5
6  new l1 := (a * l0 + c) mod m;
7  new acard := l1 mod 52;
8  observe acard;
9
10 new l2 := (a * l1 + c) mod m;
11 new bcard := (acard + 1 + (l2 mod 51)) mod 52;
12 secret bcard
```

Alice on line 8. The second pseudorandom number *l2* is then generated, and a more complex modulo operation maps it to a different position in the deck by mapping it to an integer in the interval [1,51] and adding the result to the position of Alice's card, modulo 52. This second position is that of Bob's card (*bcard*), and is identified as a secret of interest to Alice on line 12. By measuring the flow of information from the position of Bob's card in the deck to that of Alice's card and computing the amount of information shared between the two positions, we can evaluate the suitability of this particular implementation for selecting random deck positions.

Figure 6.4: a heat map representing the (very unevenly distributed) joint probability distribution of the values of *acard* and *bcard* in the program in Listing 6.3



`chimp` reveals that there are around 3.1706 bits of information shared between the positions of Alice's and Bob's cards in the deck in the program shown in Listing 6.3; this is significantly worse than the 0.0280 bits of mutual information in the ideal implementation, and therefore this implementation clearly has disastrous effects on the game's fairness. The joint distribution of the values of *acard* and *bcard* in this scenario, depicted in Figure 6.4, is very unevenly distributed: for a given value of *acard*, only between six and nine corresponding values of *bcard* occur with a non-zero probability; *e.g.*, when the value of *acard* is 48, the value of *bcard* is drawn from one of $\{10, 12, 27, 29, 44, 46\}$, as opposed to being drawn from one of the remaining 51 values as in the ideal implementation. Even worse, the values of *bcard* that *do* occur with a non-zero probability are themselves unevenly distributed; *e.g.*, when the value of *acard* is 48, the value of *bcard* is 44 with probability around 0.08, but is 22 with probability around 0.25. This uneven distribution makes it much easier for Alice to guess the position of Bob's card in the deck based on the position of her own: the min-entropy leakage of around 3.4075 bits from *acard* to *bcard* reveals that Alice is on average approximately 10.6 times more likely to correctly guess the position of Bob's card in the deck in a single attempt in this implementation when compared to the ideal implementation.

We can see the same game being played from Bob's perspective by exchanging the locations of the secret and observe commands in Listing 6.3: line 8 becomes secret *acard*, and line 12 becomes observe *bcard*. The mutual information is the same as it is from Alice's perspective — unsurprising, given that mutual information is symmetric — but the min-entropy leakage of around 3.7165 bits from the position of Bob's card in the deck to the position of Alice's reveals that Bob is around 13.1 times more likely to guess Alice's card correctly in a single attempt. This implementation of the game's card selection phase is therefore deeply flawed, and gives Bob an unfair advantage over Alice when the players decide whether to place a bet.

6.2.3 Maximising the Period of a Linear Congruential Generator

It is worth considering precisely what makes the program in Listing 6.3 leak so much information. We saw in the ideal implementation of the playing card game that neither player learns more information about the deck position of the other's card than they do by observing their own — what property of this program causes it to leak more information than the minimum amount?

One flaw in the program's implementation of the game's card selection phase is that an LCG is used to select cards from random positions in the deck, but LCGs are *not* cryptographically secure PRNGs. The arithmetic in the recurrence relation is performed modulo m, meaning that any integer i outputted by an LCG must fall in the interval $[0, m)$, and an LCG with fixed parameters m, a and c will always produce the same succeeding state L_{n+1} for a given current state L_n; an LCG is therefore a *periodic function* (*i.e.* it repeats its output sequence after a certain interval), meaning that the sequence generated by an LCG must be finite. The period of an LCG is an important factor in determining the quality of the randomness it generates: long periods make the output sequence less predictable and thus provide better-quality (although *not* true) randomness, whereas short periods make the output sequence more predictable and thus provide poor-quality randomness.

We must therefore maximise the period of the LCG in the program to make its output as unpredictable as possible. Since $0 \leq L_n < m$, the period of an LCG is at most m, but this maximum period is not guaranteed for all LCGs; Hull and Dobell (1962) prove the existence of a relationship between the period of an LCG and its parameters m, a and c. They show that, to maximise the period of an LCG for all possible seeds $0 \leq L_0 < m$, the values of m, a and c must be chosen such that:

(a) for every prime number p dividing m, $(a-1)$ is a multiple of p;

(b) if m is a multiple of 4, $(a-1)$ is a multiple of 4; and

(c) c is relatively prime to m; *i.e.*, there must be no positive integer greater than 1

that divides both c and m.

For any combination of m, a and c that does not fulfil these criteria, the period will not be maximised for all possible seeds.

Are the values of m, a and c used in Listing 6.3 consistent with Hull and Dobell's theorem? We can check: $m = 8192$ has only one distinct prime factor (2), and $(a-1) = 788$ is a multiple of 2, so the first condition is satisfied, and both 8192 mod 4 = 0 and 788 mod 4 = 0, so the second condition is also satisfied. However, 8192 and $c = 366$ are both even and thus they clearly have a common divisor of 2; the third condition is not satisfied, so the period of an LCG with these parameters is shorter than $m = 2^{13}$; it therefore provides poor-quality randomness.

Now that we know these LCG parameters are poor, we can tweak them to find values that satisfy Hull and Dobell's theorem. Keeping the modulus $m = 8192$ the same, we find a value $(a-1)$ that is a multiple of both 4 and all of m's distinct prime factors; since m has only the distinct prime factor 2, any multiple of 4 will suffice. We shall choose 2048, meaning that $a = 2049$. Next, we must find a coprime integer of 8192; again, there are many to choose from, and we shall select $c = 8141$. These parameters guarantee that the period of the LCG is 8192 for all possible seeds; we can now test the effect they have on the fairness of card selection.

Listing 6.4: a fairer CH-IMP implementation of the card selection phase of the playing card game described in Section 6.2.2; greater fairness is achieved by selecting better values for the LCG parameters *m*, *a* and *c*

```
1  new m := 8192;
2  new a := 2049;
3  new c := 8141;
4  new l0 := [ 0 .. 8191 ];
5
6  new l1 := (a * l0 + c) mod m;
7  new acard := l1 mod 52;
8  observe acard;
9
10 new l2 := (a * l1 + c) mod m;
11 new bcard := (acard + 1 + (l2 mod 51)) mod 52;
12 secret bcard
```

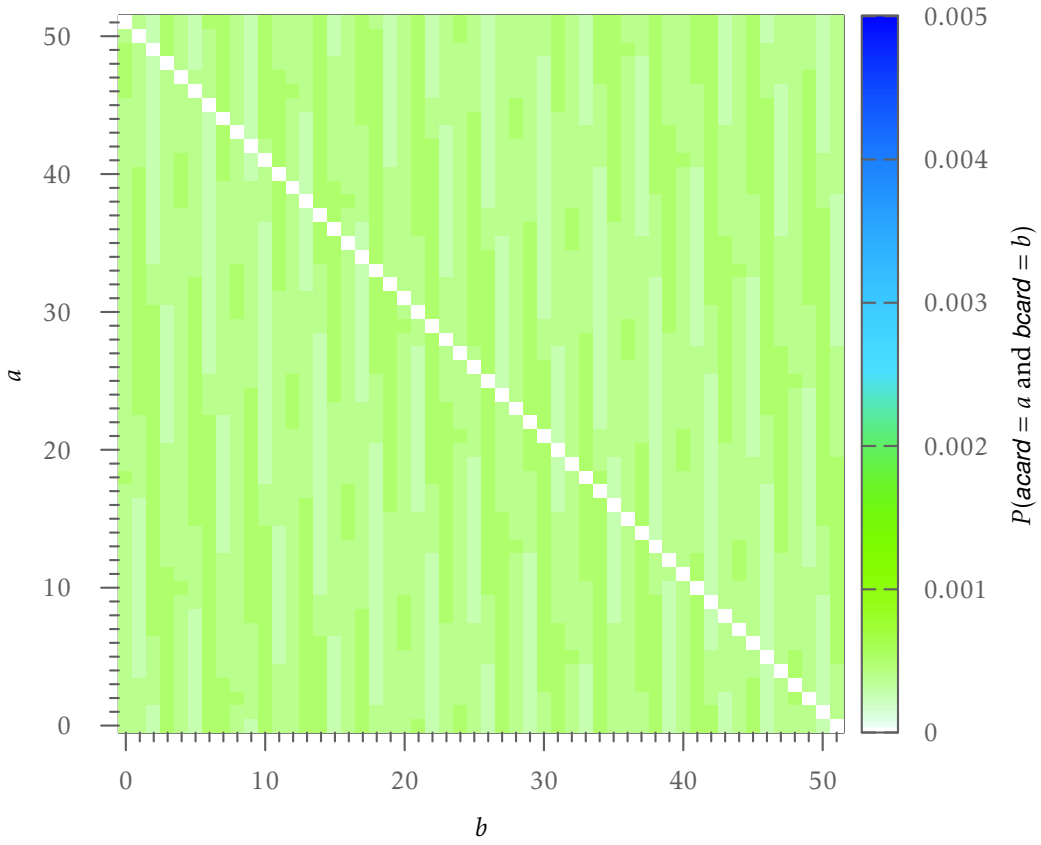
Listing 6.4 (p. 160) is a modified version of Listing 6.3 that implements an LCG with these optimal parameters. Again, the program models the game being played from Alice's perspective: the value of *acard* on line 8 is observable, and the value of *bcard* on line 12 is secret. Using `chimp`, we find that the mutual information of the positions of Alice's and Bob's cards in the deck is now around 0.067 bits. The joint probability distribution of the values of *acard* and *bcard*, depicted in Figure 6.5, reveals a much more even distribution of card positions: for any given value of *acard*, *all* 51 possible values of *bcard* occur with a non-zero probability. However, the 51 values of *bcard* are still unevenly distributed in favour of the values close to *acard*; *e.g.*, when *acard* is 30, the values of *bcard* between 25 and 48 occur with a higher probability (around 0.025) than the values of *bcard* between 3 and 21 (around 0.013). The min-entropy leakage of around 0.3785 bits from *bcard* to *acard* shows that Alice is on average 1.3 times more likely to correctly guess the position of Bob's card in the deck.

However, by exchanging the positions of the secret and observe commands to see the game being played from Bob's perspective, we see that he still has a small advantage over Alice because of the slight asymmetry of the two joint probability distributions: the min-entropy leakage of around 0.3967 bits from *acard* to *bcard* shows that Bob is on average 1.3165 times more likely to correctly guess the position of Alice's card in the deck. This implementation is therefore an enormous improvement on the one shown in Listing 6.3, but is still not quite as fair as the ideal implementation. How can it be perfected?

6.2.4 Choosing Appropriate LCG Parameters for a Given Program

There is one remaining flaw in Listing 6.4, and it relates to the way in which the LCG output is used on lines 7 and 11. The LCG parameters are optimal, so the LCG will output an integer evenly distributed over the interval $[0, 8191]$; however, these integers are reduced modulo 52 on line 7 and modulo 51 on line 11. Because neither 52 nor 51 divide 8192 evenly, the residue of neither modulo operation is uniformly distributed

Figure 6.5: a heat map representing the (slightly unevenly distributed) joint probability distribution of the values of *acard* and *bcard* in the program in Listing 6.4



over either the interval $[0, 51]$ or $[0, 50]$: it is biased toward the integers closer to 0. This affects both the selection of *acard* (whose value is more likely to be closer to 0) and the offset from *acard* used in the computation of *bcard* (whose value is more likely to be nearer to *acard*'s value).

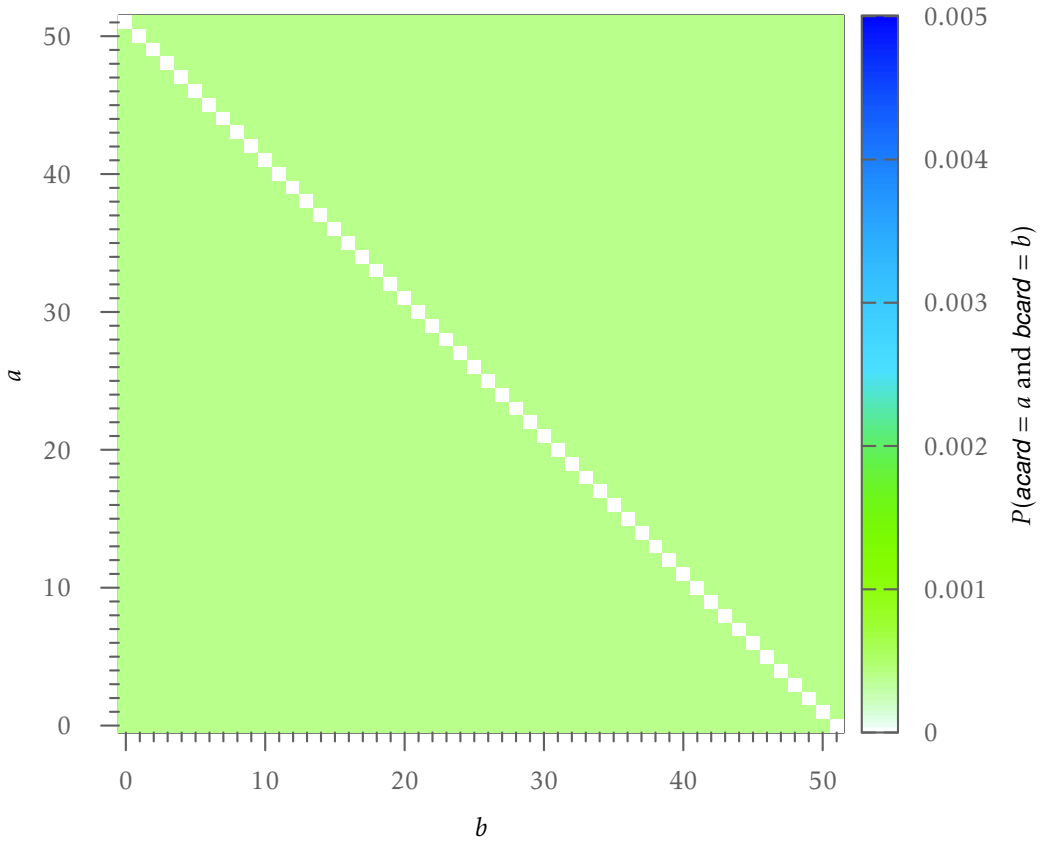
To resolve this flaw, we must choose a modulus m such that this bias does not occur; *i.e.*, we must choose m to be a multiple of $52 \cdot 51$. We shall choose $m = 52 \cdot 51 \cdot 3 = 7956$. We must also ensure that Hull and Dobell's theorem is satisfied, so we select values of a and c using the same process as in Section 6.2.3. The distinct prime factors of 7956 are 2, 3, 13 and 17, so we must choose a value of $(a - 1)$ that is a multiple of all of them; $m \bmod 4 = 0$, so we must also choose a value of $(a - 1)$ that is a multiple of 4. The simplest solution is to multiply all of these integers, giving $2 \cdot 3 \cdot 4 \cdot 13 \cdot 17 = 5304$; therefore, $a = 5305$. Finally, we must choose a coprime integer of 7956 to be c; again, there are many to choose from, and we shall select 7819.

These LCG parameters are used in Listing 6.5; note the updated line 4, which selects a seed uniformly from the interval $[0, 7955]$. `chimp` verifies that this is equivalent to the ideal implementation of the card selection phase of the playing card game, even though it does not use a cryptographically secure PRNG: the joint distribution of the values of *acard* and *bcard* is perfectly distributed apart from the cases where *acard* $\neq$

Listing 6.5: a completely fair CH-IMP implementation of the card selection phase of the playing card game described in Section 6.2.2: Alice learns only that the value of *bcard* is not the value of *acard*; the same is true for Bob's knowledge of Alice's card

```
1  new m := 7956;
2  new a := 5305;
3  new c := 7819;
4  new l0 := [ 0 .. 7955 ];
5
6  new l1 := (a * l0 + c) mod m;
7  new acard := l1 mod 52;
8  observe acard;
9
10 new l2 := (a * l1 + c) mod m;
11 new bcard := (acard + 1 + (l2 mod 51)) mod 52;
12 secret bcard
```

Figure 6.6: a heat map representing the (ideally distributed) joint probability distribution of the values of *acard* and *bcard* in the program in Listing 6.5



bcard, as shown in Figure 6.6 (p. 164). The mutual information of the values of *acard* and *bcard* is around 0.0280 bits — as it is in the ideal implementation described at the beginning of this section — and the min-entropy leakage in both directions is also around 0.0280 bits, indicating that neither Alice nor Bob gain any particular advantage when attempting to guess the position of the other player's card in the deck.

This case study demonstrates that extreme care must be taken when choosing parameters for a linear congruential generator, and underlines why they should not be used to provide randomness for arbitrary cryptographic applications.

6.3 Summary

In this chapter, we have presented two examples of CH-IMP's point-to-point information flow model being used to quantify the information leakage from various probabilistic protocols and systems. We have shown that both the CH-IMP model and `chimp` tool offer practical benefits to those seeking to analyse the flow of information in probabilistic systems of low to moderate complexity.

The first collection of systems modelled both secure and insecure DC-nets, and we used `chimp` to analyse how modifications to the DC-net protocol affected the anonymity of a bill-paying cryptographer. In doing so, we discovered what makes a DC-net secure, and how this security can be violated by contaminating the sources of randomness drawn upon by the participants; we found that the protocol is anonymity-preserving in the presence of a single malicious participant intent on disclosing the sender's identity, although the protocol leaks information at a faster rate to this malicious participant than to a passive observer as the source of randomness becomes more contaminated. A future designer of anonymity protocols could use `chimp` in a similar manner, to ensure that their protocol does not inadvertently leak information, and to discover the limits of the anonymity offered by the protocol.

The second case study involved the modelling of a two-player playing card game

in CH-IMP, and discovering what made the implementation unfair. This was a long and complex case study that involved non-trivial mathematical concepts (*cf.* Hull and Dobell's (1962) proof of the maximum period theorem introduced in Section 6.2.3). CH-IMP is able to abstract away all of this complexity, instead focusing on answering simple questions of interest to an analyst of the system: "what does Alice learn about the position of Bob's card in the deck based on that of her own?"; "how successful is Bob's strategy for correctly guessing the face value of Alice's card in a single attempt?"; "which of these linear congruential generators provides the best results for this particular program?". We were able to analyse the fairness of the game from different players' perspectives simply by moving around secret and observe commands in the program's source code; the information flow models from the literature that we reviewed in Chapter 3 are incapable of analysing systems in this simple way, and it is this simplicity that makes CH-IMP's information flow model so appealing.

III

Estimating Information Leakage from Programs

7

An Information Leakage Estimation Tool for Java

In Chapter 4 we motivated the creation of a new point-to-point information flow model, and in Chapter 5 we presented an implementation of this model in a software tool that enables the precise computation of information leakage from programs written in CH-IMP, a simple probabilistic imperative language. By providing a semantics for our information flow model we have demonstrated its soundness, and by providing two detailed case studies (and several smaller examples) of its usage we have shown its usefulness in precisely quantifying the information leakage in systems and protocols of low to medium complexity.

The execution of CH-IMP programs is defined in terms of discrete-time Markov chains; this allows for easily-understood definitions of both program execution and information flow. However, this simplicity comes with a cost: as with other DTMC-based system models, and as we alluded to in Section 5.2.3 (p. 134), it is computationally infeasible to analyse complex programs using CH-IMP due to the explosion of the state space that occurs in such programs. It is also infeasible to represent programs written in real-world programming languages in our system model (*i.e.* by transforming them into CH-IMP programs): to do so would require the modelling of (*i.e.* defining a formal semantics for) complex features found in modern programming languages in CH-IMP,

such as socket-based communication and inter-process communication; since every programming language implements these features differently, the transformation process would be prohibitively difficult, requiring a different translation semantics for each source language. Additionally, even if this translation were practical, many information leakage vulnerabilities in programs are in fact caused by programmers erroneously implementing theoretically secure algorithms and protocols (Chatzikokolakis et al., 2010); transforming a given program into a CH-IMP program would therefore likely leave information leaks in the original implementation undetected.

Clearly, it is desirable to quantify the flow of information in programs that cannot be represented in our system model; we therefore now consider a slightly different approach.

Recall the three components of our information flow model: the system model (*i.e.* a CH-IMP program), the attacker model (*i.e.* a passive attacker with knowledge of the system's behaviour), and the security policy (*i.e.* a statement of the maximum amount of information that may flow from the system's secret values to its observable values, defined in terms of an information leakage measure such as mutual information or min-entropy leakage). To determine whether the security policy is satisfied, we compute the joint probability distribution of the secret and observable values from the system model and compute the leakage measure from this distribution. One solution to the state space explosion problem that would conveniently also allow us to analyse programs in languages other than CH-IMP would be to relax the system model: rather than deriving the exact joint probability distribution from the execution of a CH-IMP program modelled as a DTMC, we could instead derive an *approximation* of it from the repeated execution of a program written in another language, collecting the secret and observable values that occur during each execution; provided that enough executions were performed with respect to the number of possible secret/observable value pairs that could occur, this approximate joint distribution would be an accurate estimate of the true joint distribution. A quantitative security policy would then

state *approximately* the amount of information that could flow from a program's secret values to its observable values before being deemed insecure, defined using the same information-theoretic leakage measures.

Being free of the state space explosion constraints of DTMCs, this approach would indeed allow us to analyse the security of more complex programs, but it causes other problems. One is that, by constructing the joint probability distribution by repeatedly executing a program and collecting the secret and observable values that occur, we have no guarantee of the similarity of the estimated distribution and the true distribution: how could we be sure that any purported flow of information were not due to noise in the data we collected? Another is that the leakage measures of mutual information and min-entropy leakage that we have relied upon thus far require that the random variables on which they operate are known (*i.e.*, the probabilities in the probability distributions describing the random variables are exact): how do we account for variance in the sample when we compute the leakage measure to verify whether the security policy is satisfied?

In this chapter, we solve these problems using the statistics results from the literature that we reviewed in Section 3.3 (p. 63). We show how this new approximation of our information flow model can be applied to Java programs (although similar techniques can be applied to programs written in *any* language). We investigate some of the engineering challenges involved in doing so, and present novel techniques for overcoming them. This culminates in the development of LeakiEst, a Java library for estimating information leakage from samples of probabilistic systems, and a second software tool, LeakWatch, for estimating the information leakage that occurs in Java programs.

This chapter incorporates content from two of our publications in the literature (Chothia et al., 2013a, 2014).

7.1 Sample-Based Estimation of Information-Theoretic Measures

We begin by demonstrating how the two information-theoretic measures we employed as information leakage measures in Chapter 4 can be estimated from a sample. Recall from Section 2.3.1 (p. 33) that mutual information is defined as the amount of information gained about one random variable, X, by observing another, Y, and from Section 2.3.2 (p. 38) that min-entropy leakage is defined as the average amount of additional information about X that can be guessed correctly in one attempt by observing Y.

As in Chapter 4, we assume that the sample space of X is the possible secret information that occurs in a program, and the sample space of Y is the possible publicly-observable information that occurs; thus, our measures of information leakage quantify the information shared between the secret and observable values, and the vulnerability of the secrets to single-attempt guessing attacks by attackers capable of viewing the observable information. Each data point in the sample represents a single execution of the program, and consists of the secret and observable information that occurred during that execution; thus, the sample defines the estimated probability distribution $\hat{P}_{XY}$ described in Section 3.3 (p. 63). This distribution can be marginalised to give $\hat{P}_X$ and $\hat{P}_Y$, which can in turn be used to estimate X and Y respectively. We are therefore assuming that neither X nor Y is known (*i.e.*, the probability of a particular piece of secret or observable information occurring is not exact); it is reasonable to make this assumption for complex programs, where it may not necessarily be easy to identify how likely it is that each of the possible collections of secret or observable values could occur during execution.

7.1.1 Estimating Mutual Information

We have already seen in Section 3.3.1 (p. 66) how to estimate the mutual information of two random variables if neither are known; we use the findings of Moddemeijer (1989) and Brillinger (2004) to identify whether the estimated mutual information $\hat{I}(X;Y)$ is a good approximation of the true mutual information $I(X;Y)$, and thus determine whether there is a statistically significant amount of mutual information between the secret and observable information that occurs in the generated sample.

As we are estimating both X and Y, the relevant tests are derived from Brillinger's χ^2 distribution for zero mutual information (Theorem 3.3, p. 67) and Moddemeijer's normal distribution for non-zero mutual information (Theorem 3.1, p. 66). To test for the presence of a statistically significant information leak in the sample, we check whether our mutual information estimate $\hat{I}(X;Y)$ is consistent with the 95% confidence interval of the distribution defined in Theorem 3.3. If it is, we conclude that the sample *does not* contain statistically significant evidence of mutual information; if it is not, we conclude that the sample *does* contain statistically significant evidence of mutual information. If we conclude from this first test that there is evidence of information leakage in the sample, we compute both the expected true mutual information by accounting for the bias in the estimate from the definition of the mean using Theorem 3.1, and the 95% confidence interval of the normal distribution to give the estimated range of the true mutual information.

In Section 3.3.1 we discussed some of the disadvantages of estimating mutual information in this manner. One of these disadvantages was the large number of data points n required to produce meaningful statistics, relative to the number of unique pieces of secret ($\#X$) and observable ($\#Y$) information; in particular, a large value of n is required to minimise the variance in the normal distribution described in Theorem 3.1, from which the 95% confidence interval is derived, and an insufficient value of n results in a confidence interval that is too wide to produce a useful estimated range

for the true mutual information. There is, however, a penalty associated with sampling the program more than is necessary to derive a sufficiently narrow confidence interval: one data point in the sample represents the information recorded from a single execution of the program being analysed, and since our goal is to estimate the information leakage that occurs from complex programs that may take some time to terminate, we must minimise the number of executions performed. Chatzikokolakis et al. provide an algorithm for determining whether the sample is large enough to conclude whether there is evidence of information leakage in the sample — it involves testing whether the estimate is consistent with the 95% confidence intervals of *both* distributions, or *neither* of them, and a larger sample must be generated if either is the case — but the authors do not provide an equivalent algorithm for determining when the sample size is sufficient both to perform a correction to the mutual information estimate and to provide accurate bounds on the true mutual information. We therefore present the following algorithm for doing so; it determines an approximate *checkpoint* at which the sample size should be sufficient.

We note that the mean and variance of the normal distribution described in Theorem 3.1 are both defined in terms of Taylor series (Moddemeijer, 1989). Computing the sum of the Taylor series in the variance definition is impractical: the sums over $\hat{P}_X$, $\hat{P}_Y$ and $\hat{P}_{XY}$ in each term make it computationally expensive to evaluate more than the first term for large sample spaces of X or Y, which is often the case for complex programs. We therefore evaluate only the first term in each series and consider it an approximation of the series' sum, and empirically determine how much error the absence of the remaining uncomputed terms introduces into the approximation by indirectly observing its effect on the mean, which is much more efficient to compute.

We begin by collecting 400 data points (followed by the initial checkpoint), and then a further 100 data points (followed by a second checkpoint). We then test whether all of the following conditions are met:

(a) $\#X$ and $\#Y$ are both positive (otherwise the estimated joint probability distri-

bution $\hat{P}_{XY}$ is undefined, and a mutual information estimate $\hat{I}(X;Y)$ cannot be computed for the final condition);

(b) $\#X$ and $\#Y$ remained constant between the previous two checkpoints (otherwise the values in the final condition cannot be compared meaningfully);

(c) at least $4 \cdot \#X \cdot \#Y$ data points have been collected; and

(d) the value $\hat{I}(X;Y) - \frac{(\#X-1)\cdot(\#Y-1)}{2n}$ did not change beyond a given tolerance ε between the previous two checkpoints (otherwise the higher-order terms in the Taylor series are non-negligible, and are still affecting the estimation of the distribution's mean).

If these conditions are met, we stop collecting data points; if not, we collect another 100 or $\#X \cdot \#Y$ data points (whichever is greater), followed by another checkpoint at which we again test whether the conditions are satisfied. By the time the algorithm stops collecting data points, the value of n should be large enough such that the $\mathcal{O}(1/n^2)$ and higher-order terms in both series are orders of magnitude smaller than the first term, and the first term is a good enough approximation of the series' sum to produce an accurate approximation of the true mutual information with a reasonable 95% confidence interval.

We have experimentally verified the quality of the samples generated by this algorithm; it proves to be a useful heuristic for generating samples that produce accurate mutual information estimates and confidence intervals for programs containing both small and large amounts of unique secret and observable information. Requiring a minimum of 100 data points to be collected before each subsequent checkpoint prevents the algorithm from stopping prematurely when generating samples for programs that contain very small amounts of unique secret and observable information (*e.g.* those where $\#X + \#Y < 10$) due to the low value of $\#X \cdot \#Y$ in these situations, and requiring a maximum of $\#X \cdot \#Y$ data points to be collected before each subsequent checkpoint allows the algorithm to scale with the amount of secret and observable

information that occurs. The tolerance ε is a parameter of the algorithm, and can be fine-tuned depending on the program being sampled; a reasonable initial value is 0.01, although programs containing very subtle information leaks sometimes require lower values of ε (*e.g.* 0.001) in order to generate samples that can detect those leaks. We have implemented this algorithm in both leakiEst and LeakWatch using a default tolerance of $\varepsilon = 0.01$ (although this can easily be modified by the user).

7.1.2 Estimating Min-Entropy Leakage

In Section 3.3.2 (p. 70) we summarise Chothia et al.'s (2014) method of computing a greater than 95% confidence interval for min-entropy leakage estimates. Unlike Chatzikokolakis et al.'s (2010) work on mutual information estimation, this work does not attempt to correct any bias that may be present in a sample that would affect the min-entropy leakage estimate; we therefore assume that the sample contains statistically significant evidence of information leakage if the lower bound of the confidence interval is greater than 0 bits, and contains no evidence of information leakage otherwise.

Chothia et al. also present iterative algorithms for experimentally finding $s_{\max}$ and $s_{\min}$, the joint frequency distributions that maximise and minimise the conditional vulnerability respectively, and $u_{\max}$ and $u_{\min}$, the input frequency distributions that maximise and minimise the vulnerability respectively. The maximising distributions are found from the equivalent empirical distributions by increasing the maximum frequencies, decreasing the other frequencies at the same rate, and testing the effect on the vulnerability or conditional vulnerability; the inverse is performed to find the minimising distributions. We use the algorithms presented by Chothia et al. as-is to find these frequency distributions.

The authors use Pearson's (1900) χ^2 tests to determine whether various joint frequency distributions derived from the sample follow the expected joint frequency distributions. Pearson's χ^2 test makes several assumptions about the sample to which it is

applied: (*a*) data points are sampled randomly, and are independent of each other (we make the same assumptions when estimating mutual information), (*b*) at least 80% of the frequencies in the joint frequency distribution are greater than 5, and none are equal to zero, and (*c*) the sample is sufficiently large to avoid false negatives in the test results. If these assumptions are not met, the test results are unreliable; as with mutual information estimation, we must therefore ensure that the sample is sufficiently large for the tests to produce reliable results. We present the following algorithm for doing so; it is similar to the equivalent algorithm for mutual information described in Section 7.1.1, in that it determines an approximate *checkpoint* at which the sample size should be sufficient.

We begin by collecting 1,000 data points (followed by the initial checkpoint), and then a further 1,000 data points (followed by a second checkpoint). We then test whether all of the following conditions are met:

(a) $\#X$ and $\#Y$ are both positive (otherwise the joint frequency distribution is undefined);

(b) $\#X$ and $\#Y$ remained constant between the previous two checkpoints (otherwise the program has evidently not been sampled a sufficient number of times to capture the range of possible pieces of secret and publicly-observable information that truly occur);

(c) at least $10 \cdot \#X \cdot \#Y$ data points have been collected;

(d) at least 80% of the frequencies in the joint frequency distribution are greater than 5 (an assumption of the χ^2 tests); and

(e) none of the frequencies in the joint frequency distribution are equal to zero (another assumption of the χ^2 tests).

If these conditions are met, we stop collecting data points; if not, we collect another 1,000 or $\#X \cdot \#Y$ data points (whichever is greater), followed by another checkpoint at

which we again test whether the conditions are satisfied. By the time the algorithm stops collecting data points, the value of n should be large enough for the χ^2 tests to succeed, and therefore the confidence interval for the min-entropy leakage estimate should indicate whether or not there is any statistically significant evidence of min-entropy leakage from the secret information to the observable information.

Chothia et al. note that the number of data points required for a reliable min-entropy leakage estimate is significantly larger than the number required for a reliable mutual information estimate, and this is factored into the algorithm: note that subsequent checkpoints occur after collecting a minimum of 1,000 data points, compared with 100 in the mutual information variant of the algorithm. In practice, we find that this algorithm is an effective heuristic for collecting a sample that produces a "stable" min-entropy leakage estimate (*i.e.*, by collecting more data points than this algorithm states is necessary, the min-entropy leakage estimate varies only slightly from the estimate provided by the algorithm). The confidence intervals derived from samples generated by this algorithm are much wider than their mutual information counterparts, but this is understandable given that the precise distribution from which mutual information estimates are drawn is known and thus a precise confidence interval can be computed, whereas Chothia et al.'s algorithm determines the min-entropy leakage confidence interval experimentally; in any case, the confidence interval is sufficient to determine whether the sample contains any statistically significant evidence of min-entropy leakage from the secret information to the observable information.

7.2 leakiEst: A Sample-Based Information Leakage Estimation Tool

Based on the information leakage measure estimation results from both the literature and Section 7.1, we have developed leakiEst, a freely-available[1] quantitative information-theoretic measure estimation tool and library. leakiEst can be used both as a command-line tool and as a Java API that exposes information leakage measure estimation functionality, as well as general information-theoretic functionality, to third-party Java software.

leakiEst is system-agnostic: rather than estimating the information leakage from a program directly, it operates on a *dataset* — essentially a sample of a system's secret and observable information generated via repeated, independent and identically distributed (i.i.d.) executions. This affords leakiEst a great deal of flexibility: it can be used to estimate the information leakage of *any* system that accepts its secret data and outputs its observable data in a predictable manner; for example, it is possible to automatically generate leakiEst-compatible datasets for cryptographic hardware such as that embedded in electronic passports (Chothia and Smirnov, 2010), and programs whose source code is unavailable, such as closed-source firmware distributed in binary format. The requirement for i.i.d. executions is a prerequisite of the statistical tests of Chatzikokolakis et al. (2010), and was discussed in Section 3.3.1 (p. 66); leakiEst does not perform any statistical tests to check whether the data points are i.i.d., leaving the responsibility of doing so to the user.

Several dataset formats are supported; the two most commonly-used are a simple "secret/observation" file format, where each line represents a single data point from a sample (*i.e.* the secret and corresponding observation that occurred during a single execution of the system), and the more structured attribute-relation file format (ARFF)

[1] leakiEst is available at https://www.cs.bham.ac.uk/research/projects/infotools/leakiest/, and is licensed under version 3 of the GNU General Public License.

initially developed for use with the machine learning tool Weka (Hall et al., 2009). Given a dataset in a supported format, LeakiEst computes the estimated joint probability distribution of the system's secrets and observations, and performs the statistical tests described in Section 7.1 to compute an accurate estimate of an information-theoretic measure of the user's choice. By modifying a probabilistic program so that it writes its secrets and observations to a file in a format supported by LeakiEst, we can use LeakiEst to estimate the information leakage of the program; we shall investigate how this can best be automated for Java programs in Section 7.3 (p. 185).

The focal point of LeakiEst's API is the `Observations` class, which represents a parsed dataset (*i.e.* its unique secrets and observations, and the probability of each secret and observation occurring simultaneously); other classes perform the statistical analysis required for estimating the mutual information and min-entropy leakage of the secrets and observations represented in `Observations` objects using the algorithms described in Section 7.1. Notably, LeakiEst implements the "sufficient sample size" algorithms described in Sections 7.1.1 and 7.1.2, and will therefore stop parsing a dataset when it detects that the number of data points it has read is sufficient to compute an accurate leakage estimate.

7.2.1 Analysing Random Number Generation Programs with LeakiEst

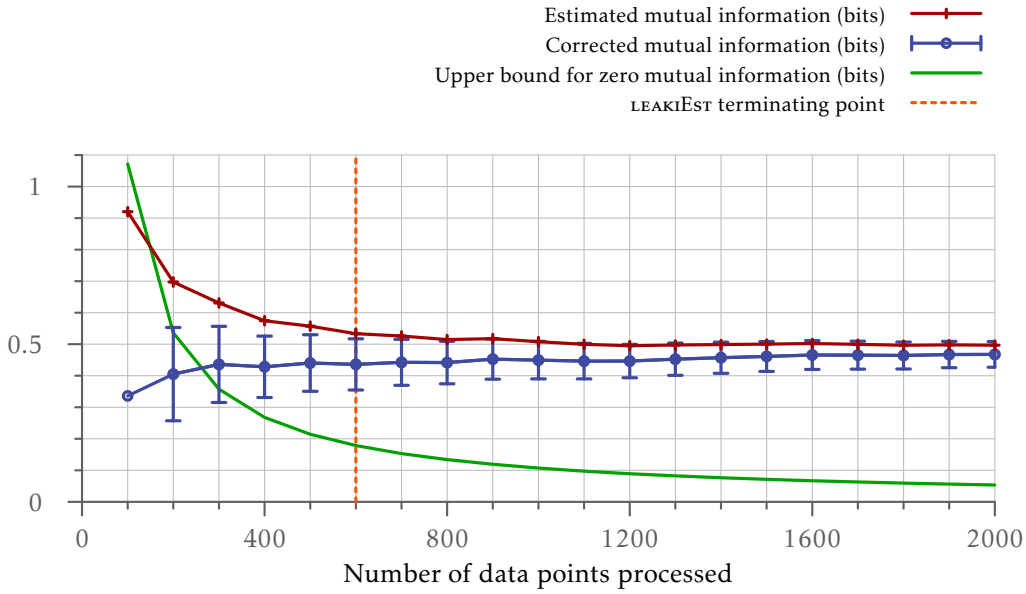
Figures 7.1 and 7.2 (pp. 181 and 182) demonstrate how LeakiEst processes datasets that are generated by programs that repeatedly execute a given block of code and write the secrets and observations that occur in LeakiEst's "secret/observation" dataset format. The two Java programs that generated these datasets are shown in Listings 7.1 and 7.2 (p. 183). They are similar to the CH-IMP programs in the linear congruential generator (LCG) case study from Chapter 6: two random integers in the interval $[0, 9]$ are generated consecutively and treated as observable and secret values respectively. The difference between the programs lies in how they randomly generate their integers: the first program uses the LCG-based `Random` class from the Java API, and the

second uses the cryptographically-secure `SecureRandom` class from the Java API; both are seeded with a random integer in the interval $[0,199]$ that is generated using the same class. A single data point in each dataset therefore consists of a pair of integers, *e.g.* `("2","9")`.

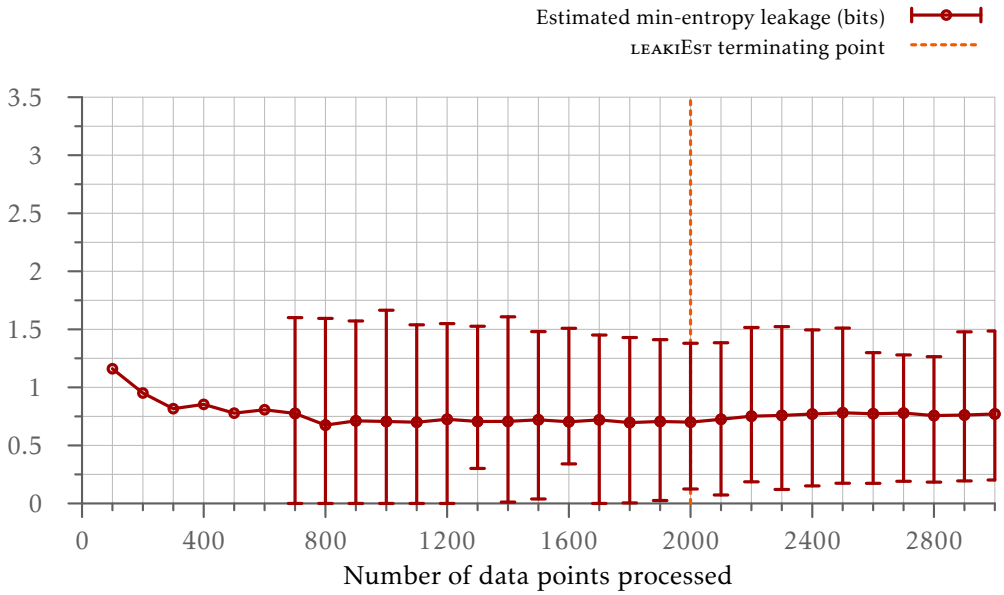
As we saw in Section 6.2 (p. 153), LCGs provide poor-quality randomness unless properly designed for specific applications: their output is otherwise easily predictable, especially when the numbers they generate are mapped onto a smaller range, and we would therefore expect there to be a correlation between the two integers in each data point in the `Random` dataset. Figure 7.1(a) verifies that this is the case: after processing 600 data points, leakiEst confirms that the random variable defined by the estimated joint probability distribution $\hat{P}_{XY}$ is no longer consistent with the χ^2 distribution for zero mutual information (Theorem 3.3, p. 67), and that there is evidence of an information leak from the secrets to the observations in the dataset. Figure 7.1(b) confirms the presence of a statistically significant min-entropy leakage from the second integer to the first after processing 2,000 data points.

Cryptographically secure PRNGs, however, provide much higher-quality randomness; we would therefore expect there to be *no* correlation between the two integers in each data point in the `SecureRandom` dataset. Figure 7.2(a) confirms that this is the case: at no point during processing of the second dataset does leakiEst find that the random variable defined by the estimated joint probability distribution is inconsistent with the χ^2 distribution for zero mutual information, and leakiEst states that there is *no* evidence of information being shared between the secrets and observations in the dataset after reading 700 data points. Again, leakiEst correctly identifies that there is no statistically significant min-entropy leakage from the second integer to the first after reading 2,000 data points.

Figure 7.1: LEAKIEST processing datasets derived from repeatedly executing two Java programs that generate two consecutive random integers between 0 and 9 inclusive using the Java API's `Random` class; the first integer is the program's publicly-observable information, and the second is its secret information

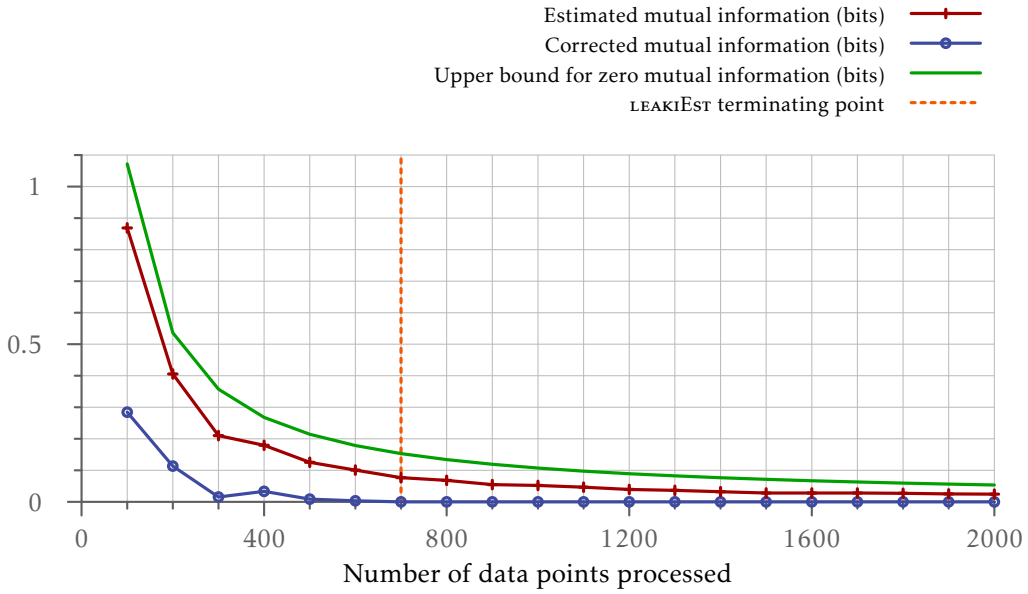


(a) the mutual information of the two integers

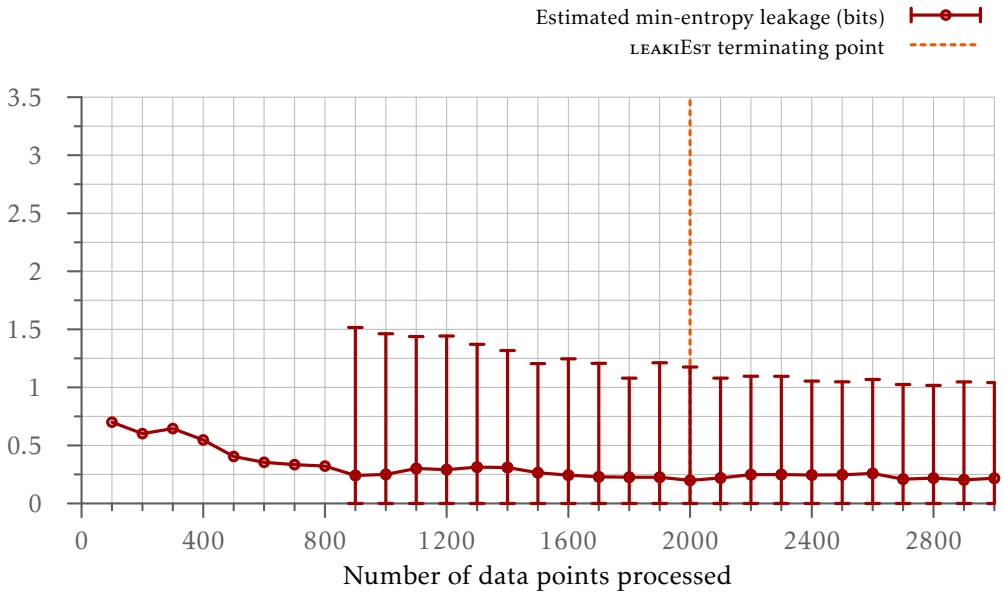


(b) the min-entropy leakage from the second integer to the first

Figure 7.2: LEAKIEST processing datasets derived from repeatedly executing two Java programs that generate two consecutive random integers between 0 and 9 inclusive using the Java API's `SecureRandom` class; the first integer is the program's publicly-observable information, and the second is its secret information



(a) the mutual information of the two integers



(b) the min-entropy leakage from the second integer to the first

Listing 7.1: the Java program that generated the dataset for Figure 7.1

```
1  import java.util.Random;
2
3  public class LowEntropyRandom {
4
5    public static void main(String[] args) {
6      for (int i = 0; i < 3000; i++) {
7        Random genSeed = new Random();
8        Random gen = new Random(genSeed.nextInt(200));
9
10       int obs = gen.nextInt(10);
11       int sec = gen.nextInt(10);
12
13       System.out.println("(" + sec + "," + obs + ")");
14     }
15   }
16
17 }
```

Listing 7.2: the Java program that generated the dataset for Figure 7.2

```
1  import java.nio.ByteBuffer;
2  import java.security.SecureRandom;
3
4  public class LowEntropySecureRandom {
5
6    public static void main(String[] args) {
7      for (int i = 0; i < 3000; i++) {
8        SecureRandom genSeed = new SecureRandom();
9        long seed = genSeed.nextInt(200);
10       byte[] bytes = ByteBuffer.allocate(8).putLong(seed).array();
11       SecureRandom gen = new SecureRandom(bytes);
12
13       int obs = gen.nextInt(10);
14       int sec = gen.nextInt(10);
15
16       System.out.println("(" + sec + "," + obs + ")");
17     }
18   }
19
20 }
```

7.2.2 Performance Evaluation

leakiEst reaches each of the conclusions of Section 7.2.1 after around a wall-clock second of dataset processing.[2] The dashed vertical line in each graph illustrates the point at which leakiEst stops reading data points from the dataset because of its confidence that the data points it has read are sufficient to produce an accurate estimate. As can be seen from the trend beyond these lines, the algorithm from Section 7.1.1 correctly identifies when the corrected mutual information estimate has stabilised (*i.e.* when the higher-order terms in the mean and variance equations in Theorem 3.4 (p. 67) are infinitesimal), and therefore does not process an unnecessarily large number of data points from the dataset in order to make a conclusive final estimate. The algorithm from Section 7.1.2 also correctly identifies when the min-entropy leakage estimate has stabilised, although the estimate is less precise than the mutual information estimate because the confidence interval is much wider than its mutual information equivalent (for all the reasons we discussed in Section 7.1.2); however, it still correctly identifies the existence of an information leak in the dataset.

leakiEst must iterate over each unique combination of secret and observation to compute the joint probability distribution for a given dataset, which is computationally expensive for large numbers of unique secrets and observations. Rather than explicitly storing the joint probability distribution of the unique secrets and observations that occur in a dataset and recomputing it every time a new data point is read, an `Observations` object instead stores a two-dimensional frequency table (the first dimension for secrets and the second for observations), along with the total number of data points read so far from the dataset; it is much more efficient to update these data structures when new data points are processed than it is to recompute the entire joint probability distribution. The joint probability distribution can then be constructed from this information only when it is needed rather than each time a new data point

[2] This benchmark — and the other benchmarks described in the rest of this chapter — was performed on the same computer as the `chimp` and QUAIL benchmarks in Section 5.2.3 (p. 134).

is processed, meaning that leakiEst scales excellently for datasets containing small to moderate numbers of unique secrets and observations and very large numbers of data points: leakiEst is capable of processing a dataset containing 10 unique secrets, 10 unique observations and 500 million data points[3] in under 3 minutes. leakiEst scales less impressively for datasets containing large numbers of unique secrets and observations, because of the need to derive a joint probability distribution from the two-dimensional frequency table; nevertheless, leakiEst is able to process a dataset containing 2^{14} unique secrets and observations in under 10 seconds and a dataset containing 2^{20} unique secrets and observations in around 24 hours.

7.3 LeakWatch: Automated Information Flow Analysis for Java Programs

leakiEst estimates the information leakage that occurs from secrets to observations in arbitrary probabilistic systems, and we have shown that it can be used to accurately estimate both the mutual information and min-entropy leakage of programs when given a dataset consisting of the secrets and observations that occur during executions of the program.

leakiEst allows us to estimate the flow of information in more complex systems, but it would not be easy to analyse programs with leakiEst: datasets would have to be generated by repeatedly executing the program (which would undoubtedly require extensive modification of source code to ensure that it only produces output that conforms to a leakiEst dataset format) and ensuring that all executions of the program are i.i.d., and a framework would have to be created for automatically providing input to the program if and when it expected it. These additional steps are all non-trivial, and

[3] We acknowledge that it is highly unlikely that a dataset of this size would be required to accurately estimate either mutual information or min-entropy leakage for a system containing 20 unique secrets and observations — especially given the good performance of our "sufficient sample size" algorithms — unless evidence of a particularly subtle information leak were being sought. This dataset was generated for the purpose of benchmarking leakiEst's `Observations` class.

would discourage programmers from using leakiEst to analyse the security of their programs.

We have therefore developed a second software tool, LeakWatch, which operates directly on Java programs rather than on datasets generated from programs, and provides the following additional functionality not offered by leakiEst:

(a) the automated collection of secret and observable information from Java programs via simple API calls, similar to the secret and observe commands in the CH-IMP language, requiring the user to insert only a single line of Java source code to record the occurrence of a secret or observable value;

(b) the sandboxing of programs running inside the same Java virtual machine (JVM) — a feature prohibited by default due to the JVM's design — allowing for the rapid yet statistically independent execution of Java programs, thus enabling multiple copies of the program to be executed simultaneously using multithreading; and

(c) the automatic provision of input to Java programs that read from the standard input stream in a way that does not violate either the semantics of the point-to-point information flow model on which LeakWatch is based, or the prerequisites for a meaningful statistical analysis of the program's behaviour.

LeakWatch thus not only makes it possible to perform a point-to-point quantitative information flow analysis of some types of real-world programs, but also significantly simplifies the process from the perspective of the user.

Like chimp and leakiEst, LeakWatch is freely available.[4]

[4] LeakWatch is available at https://www.cs.bham.ac.uk/research/projects/infotools/leakwatch/, and is licensed under the Simplified BSD License.

7.3.1 Automated Collection of Secret and Observable Information

One of the largest barriers to leakiEst's adoption as an information flow analysis tool is its processing of datasets, rather than programs. Programs typically provide output to users via one of the *standard streams* (*i.e.* standard out or standard error), and suppressing all of that output temporarily in order to output secret and publicly-observable information in a specific format for the purposes of information flow analysis is disruptive for the programmer, is likely to lead to an unmaintainable code base, and may in fact *introduce* information leakage bugs into the code in cases where the programmer forgets to remove the statements that output the secret information to the standard streams.

LeakWatch removes the need for programmers to perform this time-consuming and error-prone task by providing its own API — the `LeakWatchAPI` class — consisting of specific methods for recording the occurrence of secret and publicly-observable information at points in the program chosen by the programmer. The `LeakWatchAPI` class provides two methods, analogous to commands in the CH-IMP language (Section 4.1, p. 79): `secret()`, which allows the programmer to record the occurrence of a secret value and the name of the variable with which the value was associated at that point during execution, and `observe()`, which allows the programmer to record the occurrence of a publicly-observable value at that point during execution. These methods serve the same purpose in LeakWatch as they do in CH-IMP's information flow model: when the program terminates, all data that was passed to `LeakWatchAPI` via the `secret()` and `observe()` methods is converted into the unique string representations s and o respectively; s and o are then stored as a single data point in a leakiEst `Observations` object. leakiEst then computes the estimated joint probability distribution of the secrets and observations and performs its statistical analysis.

An example of LeakWatch being applied to a familiar piece of code is shown in Listing 7.3. This is the code from Listing 7.1 that estimates the mutual information

of two integers generated via Java's `Random` class, with two notable changes: (*a*) the line outputting the values of `sec` and `obs` to the standard output stream for the generation of a LEAKIEST dataset has been removed — calls to the LEAKWATCH API methods have instead been inserted on lines 11 and 13, indicating the occurrence of secret and publicly-observable information at those points; and (*b*) the `for` loop surrounding the contents of the `main()` method has been removed — LEAKWATCH will automatically execute the program's `main()` method the number of times necessary to compute an accurate information leakage estimate, with no further interaction required on behalf of the programmer.

The programmer compiles the code in Listing 7.3 with a standard Java compiler and runs LEAKWATCH, providing it with the name of a *target class* (in this example,

Listing 7.3: the Java code from Listing 7.1 (p. 183), with statements for generating LEAKIEST datasets replaced with simpler calls to the LEAKWATCH API

```
1  import bham.leakwatch.LeakWatchAPI;
2  import java.util.Random;
3
4  public class LowEntropyRandom {
5
6    public static void main(String[] args) {
7      Random genSeed = new Random();
8      Random gen = new Random(genSeed.nextInt(200));
9
10     int obs = gen.nextInt(10);
11     LeakWatchAPI.observe(obs);
12     int sec = gen.nextInt(10);
13     LeakWatchAPI.secret("sec", sec);
14   }
15
16 }
```

Listing 7.4: the output from LEAKWATCH after it estimates the mutual information of the secret and observable information in the program shown in Listing 7.3

```
1  Stopped after 600 executions: corrected leakage: 0.44 bits
2  There IS evidence of an information leak (estimated range: 0.35 - 0.52 bits).
```

`LowEntropyRandom`). LeakWatch loads the target class (and any other classes it references), locates and executes its `main()` method repeatedly to gather execution data via the `LeakWatchAPI` class, and creates a leakiEst `Observations` object consisting of the secret and observable values that were gathered; finally, it produces the output shown in Listing 7.4 (p. 188). The result is consistent with the output from leakiEst itself (Figure 7.1(a), p. 181), but no action other than identifying the occurrences of the secret and observable information was required of the programmer in order to estimate the information leakage that occurs from the target class's `main()` method.

7.3.2 Ensuring the Statistical Independence of Program Executions

We saw in Section 7.2 that leakiEst facilitates the information flow analysis of many different types of system, including those for which source code is unavailable; provided that a dataset can be generated from the system's behaviour, and that enough data points can be collected, it can be analysed successfully. This usually requires the repeated execution of the system, providing a range of secrets and recording the corresponding observations that occur. While it is also possible for programmers to generate datasets from their Java programs in this way, it is inefficient: a new JVM must be started for every execution of the program, and the JVM startup speed is relatively slow; this problem is compounded by the large number of data points (relative to the number of unique secrets and observations that occur) that are required for the successful analysis of a system using leakiEst.

A naive solution would be to modify the Java program's `main()` method so that the code contained within it is executed a fixed number of times. This eliminates the need to start up a new JVM to collect each data point. In many cases, this technique will work: in Listing 7.1 (p. 183), the main logic of the program and the statement that produces the output for the dataset are enclosed within a `for` loop whose body is executed 2,000 times to produce the dataset. This is still not an optimal solution: Figure 7.1(a) shows that only around 600 data points are required for a successful

statistical analysis of this program; the remaining 1,400 executions of the program were therefore performed unnecessarily.

There are additional cases where this technique will subtly fail to produce reliable datasets due to dependencies between executions. Consider the code in Listing 7.5; this program chooses an index of an 8-element integer array at random (which is treated as secret information), overwrites the integer at this position with the integer 0, and reveals the modified array to an attacker. There is clearly an information leak in this program: our information flow model assumes that the attacker knows the program's source code, and because the attacker knows that none of the elements in the array are initially 0, they can locate the position of the 0 in the array revealed to them to find which of the eight array indices is the program's secret. By executing this program repeatedly to generate a dataset, LEAKIEST estimates after processing around 400 data points that there are approximately 2.92 bits of mutual information between the secret array index and the array observed by the attacker, 0.08 bits short of the

Listing 7.5: a program that leaks information about the secret array index that is chosen at random due to an operation performed on the integer at that index

```
1  import java.util.Arrays;
2  import java.util.Random;
3
4  public class ChooseInt {
5
6    private static int[] ints = { 1, 2, 3, 4, 5, 6, 7, 8 };
7
8    public static void main(String[] args) {
9      // Choose a random index of the "ints" array
10     int randIndex = new Random().nextInt(ints.length);
11
12     // Overwrite the integer at this index with 0
13     ints[randIndex] = 0;
14
15     // The secret is the index we chose; the observable is the array after the
16     // overwrite operation
17     System.out.println("(" + ints[randIndex] + "," + Arrays.toString(ints) + ")");
18   }
19
20 }
```

$\log_2(8) = 3$ bits of mutual information that genuinely exist. However, if a dataset were to be generated by surrounding the contents of the `main()` method with a `for` loop and executing the program once, as in Listing 7.1, a different result would be obtained: leakiEst would state that the dataset contains no evidence of an information leak occurring. This is because the `ints` array is a static member variable: all iterations of the `for` loop operate on the same copy of the array, and over time it would increasingly be the case that *all* of the elements in the array were 0. Under these circumstances the attacker would indeed not learn any information about the secret array index, but the dataset being generated would no longer be an accurate reflection of the program's original behaviour. This naive solution may therefore accidentally introduce dependencies between executions: they may no longer be i.i.d., and leakiEst's statistical analysis may draw incorrect conclusions about the behaviour of the program.

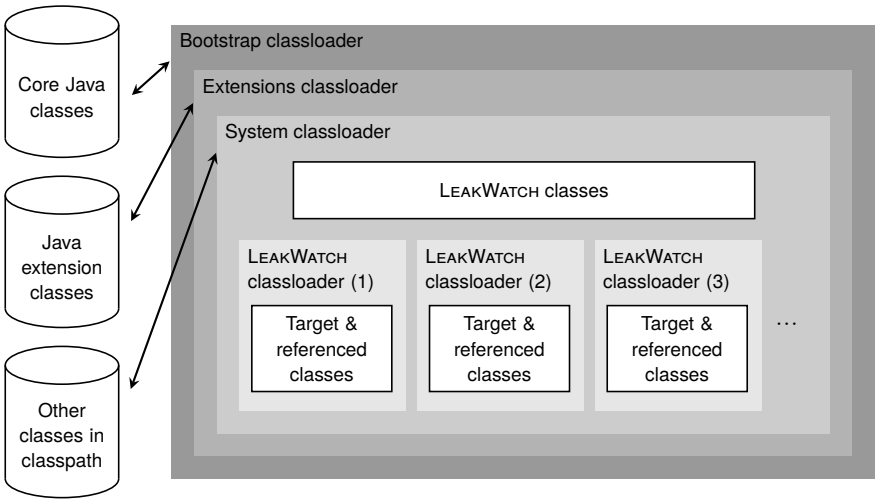
This is analogous to a situation that arises in LeakWatch when it executes a target class's `main()` method. The JVM loads classes into memory on-demand; the task of locating the bytecode that defines a class, parsing the bytecode, and returning a reference to the new class to the caller is performed by a *classloader*, itself a Java class. A key restriction is that a class with a given name may be loaded only once by a given classloader. The JVM contains three classloaders by default, arranged in a hierarchy: the *bootstrap classloader* loads the core Java classes (*i.e.* those whose names begin with `java.`), the *extensions classloader* loads Java extension classes (*e.g.* classes that ship with many JVMs that perform cryptographic operations), and the *system classloader* loads other classes. This hierarchy is strictly enforced; for example, the system classloader delegates the loading of `java.lang.String` to the extensions classloader, which in turn delegates it to the bootstrap classloader. By default, this means that LeakWatch's classes *and* the target class (and any class it references) are loaded by the system classloader.

LeakWatch runs the target class by invoking its `main()` method and waiting for it to return; this may involve the loading of other classes referenced by the target class.

If one of these classes contains static member variables — as is the case in Listing 7.5 — having the system classloader load it would cause the same problem as the one described above: LeakWatch would not be able to "reset" the values of static member variables before invoking the target class's `main()` method again, so some state may be preserved between executions. As we have shown, this potentially violates the independence of executions of the target class.

LeakWatch solves this problem by loading the target class with its own classloader, positioned in the hierarchy between the system classloader and any classloaders used by the target class (Figure 7.3). Before each invocation of the target class's `main()` method, a new LeakWatch classloader is initialised, defining only the target class. As the `main()` method executes, the LeakWatch classloader creates a new instance of any class referenced by the target class; subsequent requests for the class with that name will return this instance of the class, rather than the instance that would usually be returned by the system classloader. When the `main()` method returns, LeakWatch destroys this classloader (and therefore any class loaded by it), ensuring that earlier invocations of the `main()` method cannot interfere with future invocations. This guarantee even holds when multiple instances of the LeakWatch classloader exist con-

Figure 7.3: LeakWatch's classloader hierarchy, allowing the (mostly) sandboxed execution of multiple copies of the target class's `main()` method inside the same JVM



currently, allowing LeakWatch to perform multiple isolated invocations of the target class's `main()` method at the same time using multithreading; this can significantly reduce the time taken to collect the number of data points necessary for leakiEst to perform its statistical analysis.

The JVM's classloader hierarchy prevents the full sandboxing of the target class if it references classes that are usually loaded by either the bootstrap or extensions classloaders: in the JVM's security framework the LeakWatch classloader *must* delegate the class-loading request to them. This means, for instance, that all executions of the target class's `main()` method reference the same copy of the Java API classes, since the JVM requires that those classes are loaded by the bootstrap classloader. Although this could reintroduce statistical dependencies between executions, it is only possible with methods in a handful of classes (*e.g.* `java.lang.System`'s `setProperty()` method), and verifying whether those methods are called is achievable via static analysis.

7.3.3 Sound, Automated Provision of Input to Java Programs

Programs often accept and process user input; however, the process of prompting and waiting for the user to supply input interrupts the program's execution. This is another barrier to analysing real-world programs: the secret and publicly-observable information that occurs in a program cannot be collected dynamically if the program's execution is blocked. We now consider how LeakWatch can automatically provide user input to Java programs that expect it, in a way that is both consistent with CH-IMP's information flow model and compliant with leakiEst's prerequisites for statistical analysis.

It is possible to extend the CH-IMP model to include a formalism of user input (Chothia et al., 2013b),[5] whereby a CH-IMP program can accept input that is chosen from a predefined, fixed range according to the observable information that has

[5]The author of this thesis is a co-author of this publication, but did not contribute to the user input extension of CH-IMP; hence, it is introduced here rather than in the description of CH-IMP's formal model in Chapter 4.

occurred so far during execution; the process of deciding which input should be supplied to the program is an *input strategy*, of which there may be one and only one for a given execution of the CH-IMP program. A CH-IMP program that accepts user input can be reduced to the DTMC defined in Definition 4.6 (p. 105). For the purposes of automatically providing input to Java programs with LeakWatch, the key finding of this work is the following theorem:

Theorem 7.1 (information leakage from input-consuming CH-IMP programs)
For any CH-IMP program, there exists an input strategy that makes the program leak information if, and only if, the program leaks information for an input strategy that chooses inputs with a uniform distribution.

Given that LeakWatch adopts CH-IMP's information flow model, this theorem has the following implication for estimating the information leakage from Java programs that accept user input: provided that a single input strategy is followed for every execution (making the executions identically distributed), and that the inputs chosen during one execution do not depend on those chosen during another execution (making the executions statistically independent), we can satisfy leakiEst's i.i.d. prerequisite for performing a statistical analysis on the secret and observable information collected from the program *and* detect flows of information in that data simply by supplying inputs uniformly from a predefined, fixed range whenever the program prompts for input.

Input can be provided to Java programs in many ways (*e.g.* via keyboard or mouse interaction with GUI elements, over network sockets, or from files). We focus our attention on input provided via the standard input stream, a universal method of supplying data to software; in Java, this stream is accessed via the static member variable `System.in` of type `java.io.InputStream`, whose `read()` method returns the next byte from the input buffer. Operating systems provide a single standard input stream to a process; this means that all classes loaded by a particular JVM read from the same `System.in` stream. This is problematic because LeakWatch's classes and the target class

all execute within the same JVM; even though LeakWatch sandboxes each execution of the target program using its own classloader, all instances of the target class will share (and therefore read from) the same input stream. This is most noticeable when using multithreading to perform multiple concurrent executions of the target class's `main()` method: two instances of the `main()` method reading 20 bytes of input from `System.in` will conflict, each using the `read()` method to read approximately 10 bytes from the same stream. This leaves both instances of the program with meaningless input, and violates the i.i.d. prerequisite for executions of the `main()` method.

We solve both problems by transforming every class loaded by LeakWatch's classloader that reads from the standard input stream to instead read from an *input driver*, a mock object that mimics `System.in`. When using LeakWatch to analyse a program that reads from the standard input stream, the programmer must also write an appropriate input driver to supply input when it is required. The purpose of the input driver is to implement an input strategy, as defined above: like `System.in`, it is a subclass of `java.io.InputStream`, but its `read()` method may consult the observable information that has been encountered so far during execution — although this is optional, given the implication of Theorem 7.1 — and returns a stream of bytes comprising the selected input to the caller. When classes are loaded by LeakWatch's classloader, their bytecode is dynamically transformed using the `ASM` library (Bruneton et al., 2002) so that all references to `System.in` are replaced with references to the programmer's input driver. The loading of the input driver is also performed by LeakWatch's classloader, so each execution of the target class believes it alone is reading from the standard input stream; this means that multiple concurrent executions of a Java program that reads from the standard input stream can be analysed successfully using LeakWatch.

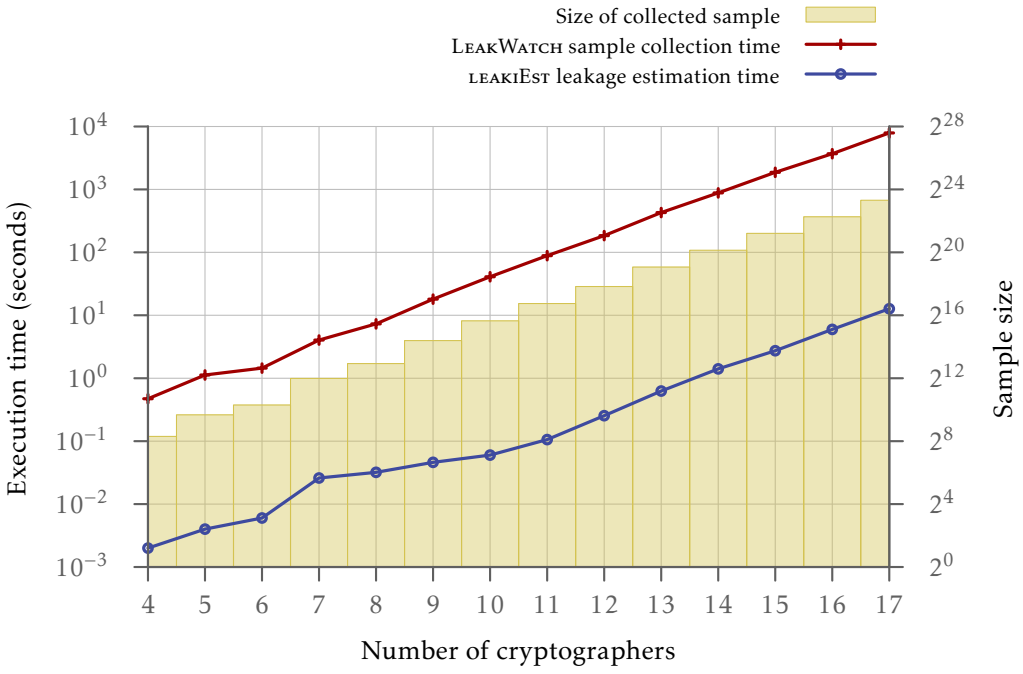
7.3.4 Performance Evaluation

The graph in Figure 7.4 shows the amount of time LeakWatch takes to estimate the information leakage that occurs from increasingly large DC-nets implemented in Java,

broken down into the time taken by LeakWatch to collect the sample and the time taken by leakiEst to inspect the sample for evidence of information leakage. The implementation contains a leak caused by biased random bit generation (a 0 is generated with probability ¾, rather than ½), making it similar to the CH-IMP DC-net example we investigated in Section 6.1 (p. 146).

Figure 7.4 shows that, as more cryptographers are added to the DC-net, the amount of time LeakWatch takes to collect a sample that is sufficient to compute an accurate leakage estimate increases exponentially; this is consistent with the performance evaluation of `chimp`, where an exponential amount of time was required to formally verify the security of DC-nets containing increasing numbers of cryptographers. The statistical estimation method employed by LeakWatch therefore ultimately suffers from the same complexity problems as the formal verification approach of CH-IMP. However, the graph shows that the major advantage of statistical estimation over the formal verification approach is its improved scalability: LeakWatch is able to collect a sam-

Figure 7.4: the performance of LeakWatch when analysing a biased DC-net with a varying number of cryptographers



ple large enough to successfully analyse a DC-net containing 12 cryptographers in 3 minutes (compared to the 4½ hours taken by `chimp` to analyse the same system formally), and LeakWatch successfully analyses a 14-cryptographer DC-net in around 15 minutes, something that `chimp` was unable to do within 100 hours. The amount of time required for leakiEst to quantify the flow of information in the sample also increases exponentially, but remains comparatively very small (less than a second for DC-nets containing fewer than 14 cryptographers), indicating that the vast majority of LeakWatch's time is spent collecting a sufficient sample, and not computing leakage estimates.

7.4 Summary

In this chapter, we have shown how the theoretical statistical results for estimating information-theoretic leakage measures that we presented in Section 3.3 (p. 63) were used to create LeakWatch, a fully-automated quantitative information flow analysis tool for Java. LeakWatch approximates CH-IMP's system model by estimating the joint probability distribution of the secret and publicly-observable information that occurs in the Java program by executing it an appropriate number of times and collecting the secret and observable information that occurs during each execution; it can therefore be used to verify whether an *approximate* security policy (*i.e.* "does the program leak less than *approximately* this amount of information?") is satisfied. We have shown how LeakWatch's design allows it to analyse multiple copies of the same Java program concurrently and independently, even if it consumes user input from the standard input stream; this can greatly reduce the amount of time taken to collect a sufficiently large sample to analyse the program.

The primary limitation of this approach is the requirement to execute the program a large number of times relative to the unique number of pieces of secret and publicly-observable information that occur. We acknowledge that this prevents LeakWatch

from being used to analyse Java programs that contain very large amounts of secret and publicly-observable information. However, if a program meets the requirements for a successful statistical analysis with LeakWatch — the ability to repeatedly execute a program with a small to moderate number of unique secrets and observations, while being able to guarantee the independent and identical distribution of the executions — it is certainly possible to sacrifice a small amount of accuracy in the information leakage measure for the ability to successfully analyse the program.

8

LeakWatch Case Studies

In Chapter 7 we presented leakiEst, an information theory library and sample-based quantitative information leakage analysis tool, and LeakWatch, an automated quantitative information flow analysis tool for Java.

In this chapter, we present three examples of LeakWatch being used to analyse insecure Java programs. We begin by revisiting a familiar topic — the security of DC-nets — and show how poor implementations of provably secure protocols can nevertheless lead to the breakdown of the security of those implementations. We then use LeakWatch to analyse a proprietary cryptographic stream cipher, and show how its behaviour reveals a surprising amount of information about its design, even though its creators intended for the design to be secret. Finally, we investigate how a misunderstanding of the security guarantees offered by OpenPGP, a data encryption standard, can lead to the compromise of a message recipient's identity.

In all three case studies, we show how a programmer can use LeakWatch to discover inadvertent information leakage vulnerabilities in the code, fix (or mitigate) the vulnerabilities, and verify that the modifications result in the eradication (or reduction in the size of) the information leak that occurs. In each case study, we provide the approximate time[1] taken by LeakWatch to perform its analysis as a means of benchmarking LeakWatch's real-world performance. As with the CH-IMP case studies presented

[1] All benchmarks described in this chapter were performed on the same computer as the `chimp`, QUAIL, leakiEst and LeakWatch benchmarks from Chapters 5 and 7.

in Chapter 6, it is important to note that LeakWatch will not explain *why* information leaks arise, *e.g.* by automatically locating the blocks of code that cause them; it will only answer questions of the form "approximately how much information flows from the secrets at these points in the code to the values at these other points in the code?". The narrative we provide for each of these case studies describes the procedure that we would expect a security-minded programmer to follow in order to verify the security of their own code, *e.g.* as part of quality assurance.

All of these case studies use an external source of randomness (namely, the `java.security.SecureRandom` class in the Java API) to provide probabilistic behaviour; this class's output does not have a statistically significant effect on the probability distribution describing the observable information that occurs in the programs, and thus it is unnecessary to identify its output as observable information in its own right, as we explained in Section 3.3.1 (p. 66).

Given the realistic nature of these case studies, they contain significantly more source code than the CH-IMP case studies presented in Chapter 6, and it cannot all be reproduced as part of this chapter. The full source code listings are therefore available on the LeakWatch web site,[2] and we highlight the relevant parts of the code in this chapter.

All of these case studies were first published in the literature as part of the publication presenting LeakWatch (Chothia et al., 2014).

8.1 A Poorly-Implemented DC-Net

We demonstrated in Section 6.1 (p. 146) that the anonymity guarantees provided to participants in DC-nets hold — even in the presence of a single malicious participant in the DC-net itself — as long as the remaining participants generate truly random bits. As the bits they generate become biased in favour of either 0 or 1, the DC-net

[2] The source code listings can be found in the "Examples" section of the web site at https://www.cs.bham.ac.uk/research/projects/infotools/leakwatch/examples/.

begins to leak information about a bill-paying cryptographer: because a bill-paying cryptographer announces the negation of the XOR of the two random bits they know, and because both of those bits are likely to have the same value, a passive attacker observing the announcements can become increasingly sure that the participant whose announcement differs from the others' announcement is the bill-payer. High-quality randomness, therefore, is crucial to the secure operation of a DC-net.

However, it is not enough merely for the DC-net (or any other security protocol, for that matter) to be theoretically secure: any implementation must also execute the protocol correctly, and, crucially, must not introduce any vulnerabilities of its own that undermine the security of the protocol. In this case study, we see how a poor implementation of the DC-net protocol in a more complex program erodes the theoretical security provided by the protocol. (As in Section 6.1, we shall assume that the bill-payer is one of the cryptographers participating in the DC-net, and that each cryptographer is equally likely to pay the bill from the perspective of a passive observer.)

The program in question implements a multithreaded, object-oriented DC-net in Java. It consists of four classes:

(a) `Diner`, a participant in the DC-net with the ability to communicate privately with individual `Diners`;

(b) `Table`, the provider of a communication channel shared by one or more `Diners`;

(c) `Waiter`, an entity assigned to a `Table` that is responsible for collecting the payment details of the bill-paying `Diner`;

(d) `DiningCryptographers`, the class that creates instances of all of the others and selects a random `Diner` to be the bill-payer (this class also contains the program's `main()` method, and thus is the class whose name is given to LeakWatch when the program is analysed).

All instances of the classes run in separate threads, and communicate with each other

via sockets using the `Socket` and `ServerSocket` classes in the `java.net` package of the Java API.

The program begins with an initialisation process, a fragment of which is shown in Listing 8.1. The `DiningCryptographers` class creates a number of `Diner` objects chosen by the user (for consistency with previous DC-net examples, we choose four), and one each of `Table` and `Waiter`. The `DiningCryptographers` class seats the `Diners` at the `Table` by disclosing to them the port number of a `ServerSocket` on which the `Table` is listening; each `Diner` connects via their own `Socket`, opens a `ServerSocket` of their own for private communication with other `Diners`, and reports this private port number to the `Table`. The `Table` then sends to each `Diner` the private port number of the `Diner` to their "left", so each `Diner` knows which other `Diner` should receive their randomly-generated bit. As part of this initialisation process, the `DiningCryptographers` class also randomly selects one of them to be the payer; since the identity of the payer is the DC-net's secret, it is marked as such with the LeakWatch API's `secret()` method on line 16. The bill-paying `Diner` is additionally informed of the port number of the `Waiter`'s `ServerSocket`.

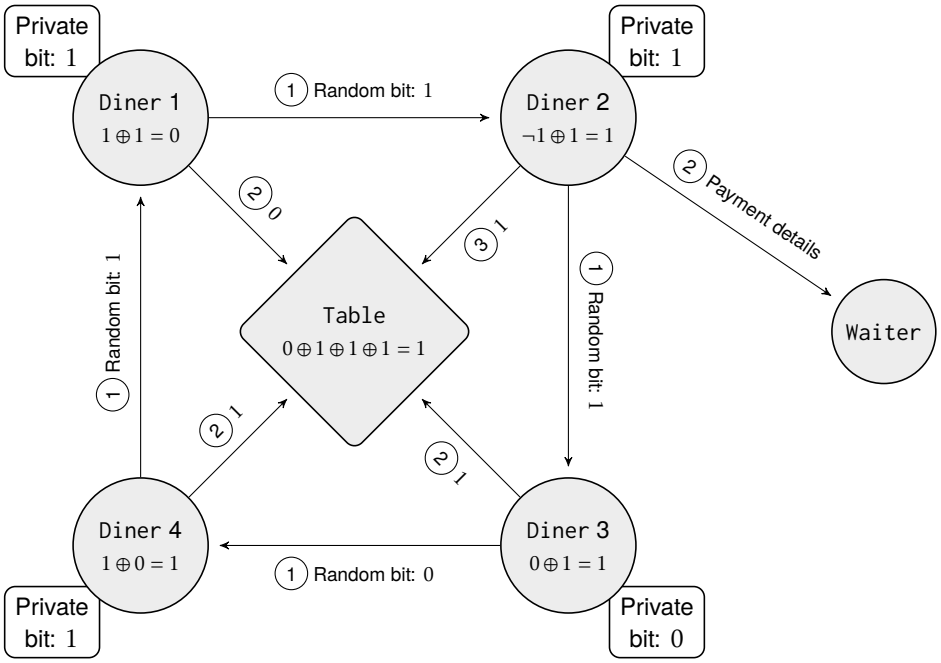
Listing 8.1: a fragment of the DC-net initialisation procedure, taken from the `DiningCryptographers` class; the program's secret is identified with a call to the LeakWatch API's `secret()` method on line 16

```
1  // Create the Waiter and Table
2  Waiter waiter = new Waiter();
3  Table table = new Table(totalDiners, waiter);
4
5  // Create the Diners, and seat them at the Table
6  Diner[] diners = new Diner[totalDiners];
7  for (int i = 0; i < totalDiners; i++) {
8    Diner diner = new Diner(i + 1);
9    diner.sitAt(table);
10   diners[i] = diner;
11 }
12
13 // Choose a Diner at random to be the bill-payer, and mark their identity as a secret
14 int dinerToPay = new SecureRandom().nextInt(totalDiners);
15 diners[dinerToPay].orderToPay();
16 LeakWatchAPI.secret("dinerToPay", dinerToPay);
```

The Diners then concurrently execute the DC-net protocol (Figure 8.1): they privately exchange their randomly-generated bits with each other over their private communication channels, and the payer sends their payment details to the Waiter over another private channel. The Diners then announce the results of their XOR computation to the Table; since the messages sent over the Table's shared channel are considered to be publicly visible, they are marked as observable with a call to the LeakWatch API's observe() method (on line 14 of Listing 8.2 (p. 204)). LeakWatch thus answers the question "what does a passive attacker learn about the identity of the bill-paying Diner by watching the messages sent over the Table's shared communication channel?".

LeakWatch takes around 2 minutes to perform 4,700 executions of the main() method in the DiningCryptographers class, and leakiEst takes around 300ms to compute a leakage estimate. LeakWatch estimates that there are approximately 1.15 of a possible 2 bits of information shared between the identity of the bill-payer and the messages the Diners send over the Table's communication channel. The min-entropy

Figure 8.1: an overview of the Java DC-net implementation, in which Diner 2 pays the bill; the final result is 1, indicating that one of the Diners paid



Listing 8.2: the `Table` class's asynchronous `run()` method, which handles incoming socket connections from `Diners` and marks their XOR computation announcements as publicly-observable information with a call to the LeakWatch API's `observe()` method on line 14

```
1  @Override
2  public void run() {
3    synchronized (dinerAnnouncements) {
4      try {
5        BufferedReader in = new BufferedReader(
6          new InputStreamReader(client.getInputStream())
7        );
8
9        // Connecting Diners are expected to send one message to the Table, in the
10       // following format:
11       // [id]: [0|1]
12       // After receiving this message, the Table will close the connection
13       String dinerResult = in.readLine();
14       LeakWatchAPI.observe(dinerResult);
15       dinerAnnouncements.add(
16         dinerResult.replaceFirst("\\d+: ", "").equals("1")
17       );
18
19       in.close();
20       client.close();
21     } catch (IOException e) {}
22   }
23 }
```

leakage is estimated to be 1 bit after 12,000 executions; 6 minutes were spent performing these executions, and 220ms were spent computing the leakage estimate. Although the messages themselves reveal no information about the identity of the bill-payer (being a faithful implementation of the provably-anonymous DC-net protocol), the order in which they are sent *does*: the additional time taken by the bill-payer to send their payment details to the `Waiter` while executing the protocol means that, more often than not, they are one of the last `Diners` to announce the result of their XOR computation to the `Table`. An attacker can therefore assume that the first and second (and probably third) `Diners` to announce are not the bill-payer, a strategy that leads to a more successful single-guess attempt at identifying the bill-payer.

This leak can be eliminated in several ways; the easiest is to modify the `Diner` class so that each `Diner` waits a short period of time (*e.g.* 100ms) between performing their XOR operation and sending their announcement to the `Table`. Assuming that it takes the bill-payer under 100ms to send their payment details, this padding makes it more likely that the `Diners`' messages will arrive at the `Table` in an arbitrary order. After performing 7,200 executions of a DC-net implementation containing this patch (taking around 13 minutes), LeakWatch confirms that there is no longer any information shared between the bill-payer's identity and the `Diners`' announcements.

8.2 Analysing the Design of Stream Ciphers

Crypto-1 is a proprietary stream cipher used to encrypt transmissions in several commercial RFID tags. The design of Crypto-1 is confidential, and its inventors intended for it to remain a secret, but cryptanalysis by Garcia et al. (2008) revealed that the cipher consists of a 48-bit linear feedback shift register (LFSR) and three Boolean functions. Each keystream bit is generated by tapping 20 bits from the LFSR and passing groups of four of these bits as input into one of two Boolean functions (f_a, f_b), which each produce one bit of output; these five output bits are in turn passed into the third

Boolean function (f_c), whose output is used to compute the next bit of the cipher's keystream. In this case study, we show that a considerable amount of detail about this structure is revealed when performing a simple black-box analysis of the cipher using LeakWatch.

We assume that we are given the JVM bytecode — but *not* the Java source code — of an implementation of the Crypto-1 cipher in a class named `Crypto1`, and that we know nothing about the class's structure other than that its constructor accepts a single parameter, an array encoding a 48-bit key that is used to initialise the LFSR's state, and that it exposes a method, `keyStreamBit()`, that returns successive bits of the keystream. This scenario is realistic, given that Crypto-1 is a proprietary (and therefore closed-source) cipher, and demonstrates that LeakWatch can be used to estimate the information leakage from programs whose source code is not necessarily available,

Listing 8.3: part of the `Crypto1BitFlip` class, which drives the `Crypto1` bytecode that implements the Crypto-1 stream cipher

```
1  // Generate a random 48-bit key
2  boolean[] key = new boolean[48];
3  SecureRandom rnd = new SecureRandom();
4  for (int i = 0; i < key.length; i++) key[i] = rnd.nextBoolean();
5
6  // Use the key as the initial state for a Crypto-1 cipher
7  Crypto1 cipher = new Crypto1(key);
8
9  // Observe first bit of the keystream
10 LeakWatchAPI.observe(cipher.keyStreamBit() ? "1" : "0");
11
12 // Flip the nth bit of the key with probability 0.5: the secret is whether the bit was
13 // flipped between initialisations of the cipher
14 if (rnd.nextBoolean()) {
15   key[n] = key[n] ? false : true;
16   LeakWatchAPI.secret("flipped" + n, true);
17 } else {
18   LeakWatchAPI.secret("flipped" + n, false);
19 }
20
21 // Use the key as the initial state for another Crypto-1 cipher
22 cipher = new Crypto1(key);
23
24 // Observe first bit of the keystream of this second cipher
25 LeakWatchAPI.observe(cipher.keyStreamBit() ? "1" : "0");
```

usually because it is legally unobtainable and challenging to reverse-engineer from bytecode.

Information about the structure of the cipher is revealed by loading different initial states into the LFSR and observing the output from the final Boolean function f_c. We begin by creating a class, `Crypto1BitFlip`, that drives the `Crypto1` class and tests the effect that the bit at index i in the LFSR has on the cipher's output; the salient part of this class's source code is shown in Listing 8.3. It begins by creating an instance of the `Crypto1` class with a randomly-generated initial state, and obtains the cipher's first output bit from f_c. The value of this bit is marked as observable using the LeakWatch API's `observe()` method on line 10. Another `Crypto1` object is created with the same initial state as before, but with the value of the bit at index i flipped with probability $1/2$ — the decision about whether or not this bit is flipped is marked as secret information using the LeakWatch API's `secret()` method on lines 16 and 18. It then obtains the output from f_c for this second `Crypto1` object, and its value is also marked as observable using the LeakWatch API's `observe()` method on line 25. Thus, the question being asked of LeakWatch is "what is the correlation between the bit at index i in the LFSR being flipped and the output of f_c changing?"; if the output bit changes when the bit at index i is flipped, clearly the bit at that index has some effect on the cipher's output, and therefore must be one of the tapped bits.

By analysing the `Crypto1BitFlip` class with LeakWatch 48 times, each time using a different value of i between 0 and 47, we can estimate the influence of every index in the LFSR on the behaviour of the cipher. As one would expect, LeakWatch reveals which indices of the LFSR are tapped and passed as input to the Boolean functions in the cipher: the indices on the x axis that fall above the upper bound for zero leakage in Figure 8.2 (p. 208) correlate with the tapped bits of the LFSR, as identified by Garcia et al.'s reverse-engineering of the cipher (Figure 8.3, p. 208). LeakWatch takes a total of 56 seconds to collect the 48 samples, and leakiEst takes a total of 530ms to analyse these samples and produce the 48 leakage estimates; each execution of LeakWatch

collects a sample of up to 700 data points from `Crypto1BitFlip`'s `main()` method to perform a successful analysis.

By collecting more than the minimum number of data points required for a successful analysis, it is possible to use LeakWatch to learn more about the structure of Crypto-1. Recall the question that LeakWatch answers of the `Crypto1BitFlip` class: "what is the correlation between the bit at index i in the LFSR being flipped and the output of f_c changing?". This is a quantitative measure, rather than a qualitative one; informally, it can be seen as the influence of the bit at index i on the cipher's keystream. If it can be estimated precisely enough, the *amount* of influence each bit has over the keystream may therefore reveal more information about the cipher's structure.

Figure 8.2: the influence over the first bit of the keystream of each bit in the 48-bit secret initial state for Crypto-1, with LeakWatch collecting the minimum sample sizes necessary for a successful analysis of each initial state bit

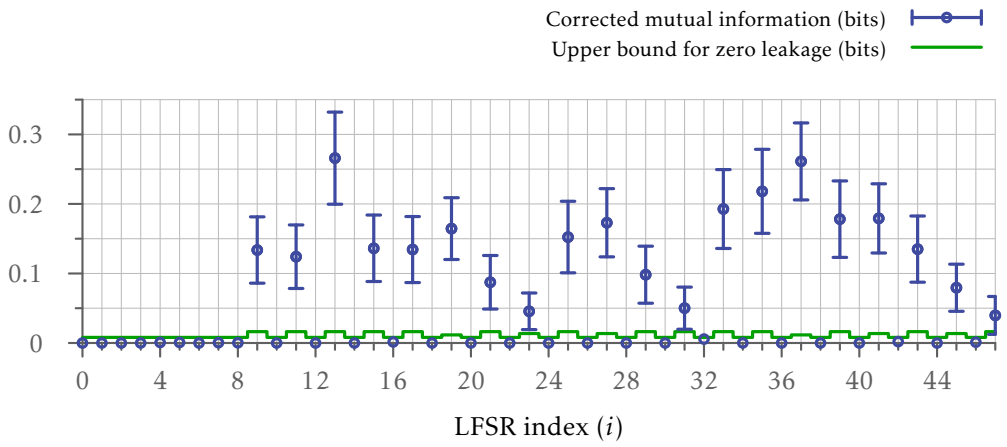
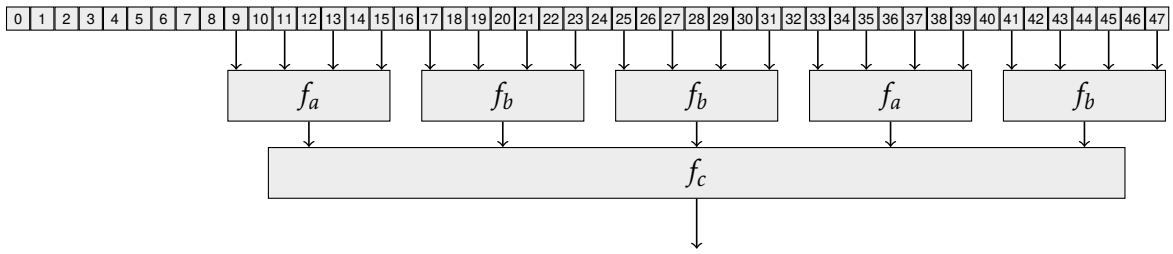
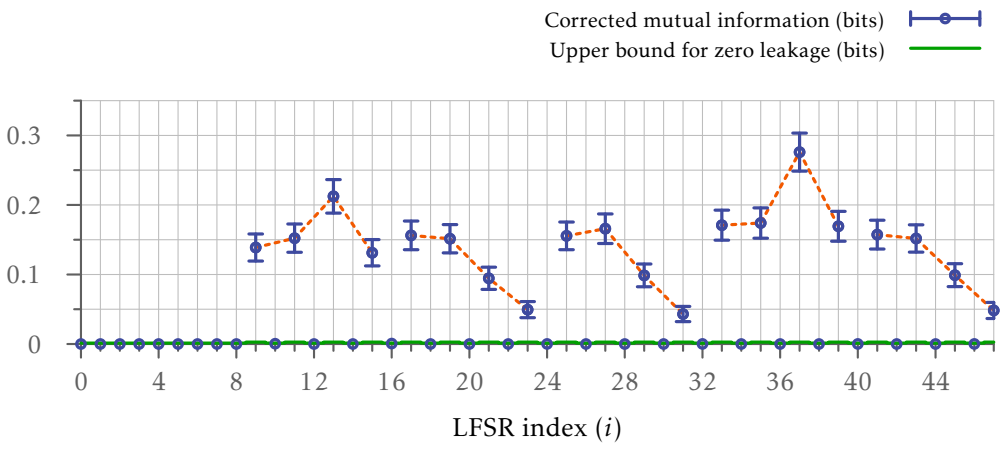


Figure 8.3: the structure of the Crypto-1 stream cipher (with the feedback function omitted), as reverse-engineered by Garcia et al. (2008)



The graph in Figure 8.4 shows the results of the same experiment performed again, but with the sample size for each execution of LeakWatch increased to 3,000 data points. Again, by reading off the indices on the x axis that fall above the upper bound for zero leakage, we see which bits are tapped to produce the keystream; the most noticeable difference is the narrower confidence interval for each point compared with Figure 8.2, owing to the increased sample size. However, notice also how the relative vertical distances between the points for each group of four indices now reveal two distinct patterns — marked with dashed lines — that were not obvious before. These patterns characterise the Boolean functions f_a and f_b: the pattern for groups $\{9, 11, 13, 15\}$ and $\{33, 35, 37, 39\}$ represents f_a, and the pattern for $\{17, 19, 21, 23\}$, $\{25, 27, 29, 31\}$ and $\{41, 43, 45, 47\}$ represents f_b (*cf.* the inputs to these functions in Figure 8.3). It is therefore possible to "see" which indices are tapped by each of these functions. Moreover, the slight variation in the groups' distances from the x axis reveals information about the third Boolean function f_c, into which the output from the other Boolean functions is passed; it is clear, for instance, that the fourth bit passed as input to f_c (*i.e.* the output from the second f_a function) has a greater influence over the keystream bit than the

Figure 8.4: the influence over the first bit of the keystream of each bit in the 48-bit secret initial state for Crypto-1, with LeakWatch collecting a sample of size 3,000 for each initial state bit; the relative vertical distances between the points characterise the Boolean functions f_a and f_b, as marked by the dashed lines



others, given that the corrected mutual information for all bits in that group is higher.

This case study shows that just the first bit of Crypto-1's keystream leaks a significant amount of information about the cipher's structure: not only does the analysis identify which bits of the cipher's LFSR are tapped to produce the keystream, it also quantifies the influence of each of these bits over the keystream. This, in turn, discloses the arrangement of other components in the cipher's supposedly secret design. LeakWatch can be used in a similar manner to analyse the structure of other proprietary LFSR-based stream ciphers, such as Hitag-2 (Verdult et al., 2012).

8.3 Recipient Disclosure in OpenPGP Encrypted Messages

OpenPGP, as defined in RFC 4880 by Callas et al. (2007), is a widely-used data encryption and authentication standard based on public-key cryptography. RFC-compliant encrypted OpenPGP messages commonly contain three types of *packet*:

(a) *Public-Key Encrypted Session Key (PKESK)* packets, containing a symmetric session key randomly generated by the sender of the message and encrypted with the public key of one of the recipients (one of these packets is present in the encrypted message for each recipient, although they all share the same symmetric session key);

(b) *Symmetrically Encrypted Integrity Protected Data (SEIPD)* packets, containing the sender's message encrypted under the symmetric session key that was generated for each of the PKESK packets; and

(c) *Modification Detection Code (MDC)* packets, allowing recipients to detect any evidence of tampering with the SEIPD packet.

Although OpenPGP provides message confidentiality (via the PKESK and SEIPD packets) and integrity (via the SEIPD and MDC packets), it does *not* guarantee recipient confidentiality, as this case study demonstrates.

In this program, two principals (represented as `Person` objects) attempt to communicate securely using OpenPGP while concealing their identities from a passive attacker with the ability to read messages sent over the communication medium. There are six principals in total, each with a different name and their own OpenPGP public key. Their details are listed in Table 8.1.

The first principal — the sender, represented as a `Person` object — is chosen randomly from the pool of six principals, and a second — the recipient, represented as another `Person` object — is chosen from the remaining five (Listing 8.4). These objects store both the public and private OpenPGP key for each principal. The identi-

Table 8.1: the names and PGP public key fingerprints of the principals

Name	Public key fingerprint
Alice	`1CE2 5A83 C3B6 97EA 180B 279F 82BA 4CE5 B015 A803`
Bob	`4199 49B7 BBA5 19BF FBEB 1C17 821F 3204 A7BD A788`
Charlie	`1540 A4C9 FD86 30CC D4FD 909E 28B5 4831 1115 A81C`
Dan	`551A 1C3F 95D6 5AE7 7850 1C39 4467 76C2 8E24 D2A1`
Elvis	`FF93 8431 AAD4 6FDE E008 5524 B84D A82C A2D6 ABCA`
Fred	`6EEC 72E0 7CCD 8F26 C455 AF69 05FF 36DE 5074 321D`

Listing 8.4: the random selection of two principals from the pool of six shown in Table 8.1; the first will attempt to securely send a message to the second

```
1  // Choose a person at random to be the sender, and mark their identity as a secret
2  Person sender = new Person(
3    Name.values()[rand.nextInt(Name.values().length)]
4  );
5  LeakWatchAPI.secret("sender", sender.getName());
6
7  // Choose a different person at random to be the recipient, and mark their identity as a
8  // secret too
9  Person recipient;
10 do {
11   recipient = new Person(
12     Name.values()[rand.nextInt(Name.values().length)]
13   );
14 } while (recipient.getName().equals(sender.getName()));
15 LeakWatchAPI.secret("recipient", recipient.getName());
```

ties of both sender and recipient are marked as secret information using calls to the LeakWatch API's `secret()` method on lines 5 and 15. Next, the BCPG Java library[3] is used to encrypt a message from sender to recipient greeting the recipient by name (`"Hello, [recipient]"`) using OpenPGP (Listing 8.5). The encrypted message therefore consists of a PKESK packet encrypted under the recipient's OpenPGP public key, which, when decrypted by the recipient using their OpenPGP private key to reveal the enclosed symmetric key, can be used to decrypt the contents of the following SEIPD packet encrypted under the symmetric key to retrieve the sender's message.

Two features of the encrypted message — its first 96 bits, and its total length in bytes — that would be known to a passive attacker observing the communication medium are marked as observable information using calls to the LeakWatch API's `observe()` method on lines 13 and 16 of Listing 8.5. Recall that the CH-IMP information flow model assumes that an attacker knows the source code of the program; thus, in this example, LeakWatch answers the question "how much information does an

[3]BCPG is a third-party OpenPGP library for Java; it is available from https://www.bouncycastle.org/java.html and is licensed under the MIT License.

Listing 8.5: a greeting from sender to recipient is encrypted using OpenPGP and sent over an insecure communication medium, allowing a passive attacker to observe both the start of the encrypted message and its entire length in bytes

```
1  // Use the OpenPGP API to encrypt "Hello, <recipient's name>" with the recipient's public
2  // key
3  byte[] encryptedMessage = sender.encryptMessageFor(
4    recipient, "Hello, " + recipient.toString()
5  );
6  StringBuilder encryptedString = new StringBuilder();
7  for (byte b : encryptedMessage) {
8    encryptedString.append(String.format("%02X", b));
9  }
10
11 // Mark part of the Public-Key Encrypted Session Key Packet as observable by the passive
12 // attacker
13 LeakWatchAPI.observe(encryptedString.toString().substring(0, 24));
14
15 // Mark the length of the encrypted message, in bytes, as observable by the attacker
16 LeakWatchAPI.observe(encryptedString.toString().length());
```

attacker with knowledge of the principals' OpenPGP public keys and the structure of the plaintext message learn about the chosen principals' identities by observing these two features of the encrypted message?".

Assuming the two principals are selected randomly, there are approximately 4.9 bits of secret information present: around 2.6 bits from the sender's identity, and around 2.3 bits from the recipient's identity. After executing this program around 1,200 times (taking 12 seconds to collect the sample and 300ms to produce the leakage estimate), LeakWatch reveals that there are approximately 2.52 bits of mutual information between the principals' identities and the features of the encrypted message observable by the attacker. A closer reading of RFC 4880 (Callas et al., 2007, Section 5.1) explains why: a PKESK packet contains the *key ID* (*i.e.* the lowest 64 bits of the SHA-1 hash) of the OpenPGP public key that was used to encrypt the packet's contents, as shown by Table 8.2. According to the RFC, this is so that a recipient's OpenPGP implementation can quickly locate the correct PKESK packet in messages that were encrypted for many recipients, rather than being forced to speculatively decrypt each PKESK packet in turn (a process which would fail for all but one of the PKESK packets). Because our attacker model assumes that the attacker knows the

Table 8.2: the first 128 bits of OpenPGP encrypted messages sent between various principals; the recipient's key ID inside each encrypted message is highlighted

Sender	Recipient	OpenPGP encrypted message
Charlie	Fred	`85010C03` `05FF36DE5074321D` `010800BA...`
Charlie	Elvis	`85010C03` `B84DA82CA2D6ABCA` `0107FE39...`
Fred	Charlie	`85010C03` `28B548311115A81C` `0107FE29...`
Dan	Elvis	`85010C03` `B84DA82CA2D6ABCA` `010800C9...`
Fred	Alice	`85010C03` `82BA4CE5B015A803` `010800AE...`
Fred	Charlie	`85010C03` `28B548311115A81C` `0108008B...`
Alice	Fred	`85010C03` `05FF36DE5074321D` `0107FE37...`
Dan	Charlie	`85010C03` `28B548311115A81C` `0107FF64...`
Bob	Fred	`85010C03` `05FF36DE5074321D` `0107FF5B...`
Dan	Bob	`85010C03` `821F3204A7BDA788` `010800C5...`
Elvis	Bob	`85010C03` `821F3204A7BDA788` `0107FE3F...`

OpenPGP public keys of all of the principals — a realistic assumption, given that OpenPGP public keys are widely published on online key servers — the attacker is able to corroborate the key ID in the PKESK packet with the identity of the owner of that key. There is therefore a complete leakage of the recipient's identity, and a further leakage of around 0.2 bits of the sender's identity, because the attacker also knows that the sender is not the recipient.

Some OpenPGP implementations mitigate this leakage of the recipient's identity; for example, when encrypting a message using GnuPG's[4] command line interface, the `-R` option replaces the key ID in PKESK packets with a string of null bytes; this is permitted by RFC 4880, which states that such "wild card" key IDs may be used to thwart analysis of encrypted messages (Callas et al., 2007, Section 5.1). BCPG's output can be patched in a similar way by overwriting the 8 bytes beginning at byte offset 4 in the encrypted message, as in Listing 8.6.

After applying this patch, the mutual information decreases to approximately 1.79 bits, which LeakWatch verifies after around 900 executions of the patched program. The recipient's identity is no longer leaked completely via the key ID in the PKESK packet, but the length of the SEIPD packet still reveals some information about the recipient's identity: longer recipient names produce longer ciphertexts when encrypted, and the attacker knows the format of the plaintext message being sent to the recipient; the attacker can therefore use the total length of the encrypted message to deduce the length of the plaintext message, and therefore the length of the recipient's name. In some cases, the attacker can guess the recipient's identity in a single attempt (*e.g.* an

[4]GnuPG, the GNU Privacy Guard, is a free implementation of the OpenPGP standard; it is available at https://www.gnupg.org.

Listing 8.6: a patch that removes the key ID from the first PKESK packet in messages encrypted with BCPG, partially fixing the information leak

```
1  // Replace the key ID in the Public-Key Encrypted Session Key packet with null bytes
2  encryptedString.replace(8, 24, "0000000000000000");
```

encrypted message with a length of 340 bytes must be intended for a recipient with a seven-character name, and only Charlie fits this description); in others, the attacker may require multiple attempts to guess the recipient's identity (*e.g.* if the length is 338 bytes, the recipient's name must be five characters long — a description matched by both Alice and Elvis). LeakWatch determines the min-entropy leakage of the patched program to be around 2 bits after 8,000 executions (taking approximately 20 seconds to collect the sample and 200ms to compute the leakage estimate); the attacker is therefore approximately four times more likely to correctly guess the identity of both the sender and recipient in a single attempt after observing the length of the patched encrypted message.

Our analysis assumes that the first 96 bits of the encrypted message are observable by the attacker. Realistically, the attacker can observe *all* of the bits in the encrypted message, but LeakWatch is unable to analyse entire encrypted messages of around 340 bytes: some of these bytes are defined (*i.e.* static) or range-limited by the RFC, but most of them comprise the encrypted session key in the PKESK packet and the plaintext message encrypted under this session key in the SEIPD packet. A successful analysis of the entire encrypted message space using LeakWatch would therefore require the enumeration of every possible PKESK and SEIPD packet for every possible combination of sender and recipient — this would require at least $4 \cdot 6 \cdot 5 \cdot 2^{256}$ executions of the program for a successful analysis (or more, if the symmetric cipher chosen to encrypt the session key were to use a key length greater than 256 bits), which is clearly infeasible.

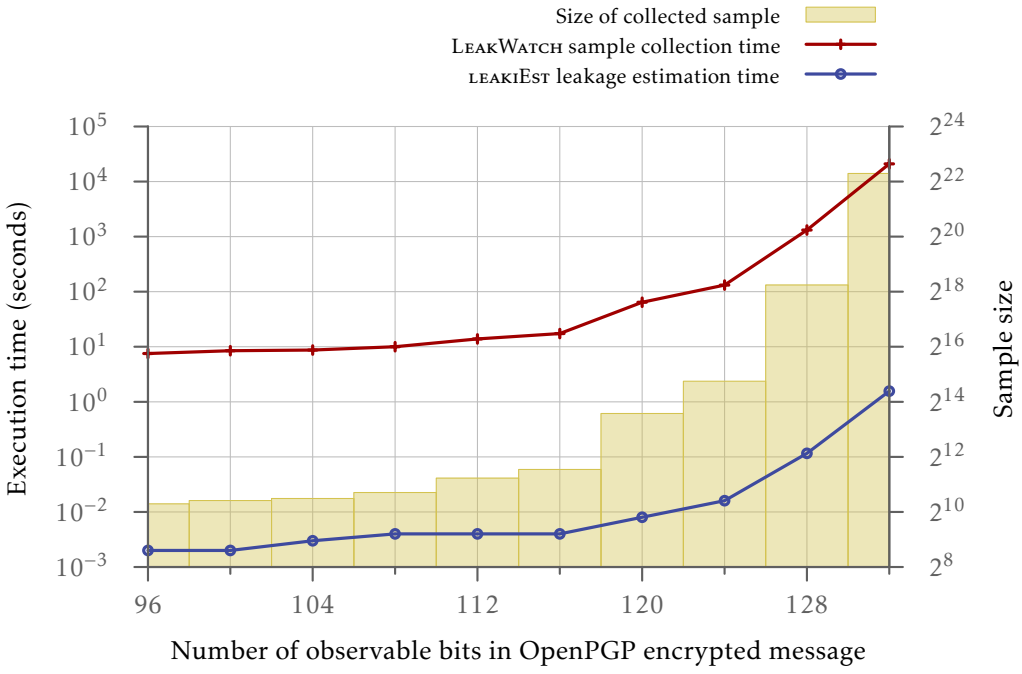
Figure 8.5 (p. 216) shows how increasing the number of bits in the message that are observable by the attacker affects LeakWatch's execution time. Predictably, it reveals a result consistent with the LeakWatch performance evaluation in Section 7.3.4 (p. 195): as the length of the observable part of the encrypted message increases, the amount of time required for LeakWatch to perform the number of executions required to estimate the program's information leakage increases exponentially (around 6 hours

when the length of the observable part of the encrypted message reaches 132 bits). The amount of time required to estimate the size of the information leak from the dataset, however, remains comparatively very small (around 1.5 seconds when the length of the observable part of the encrypted message reaches 132 bits), as it does in Section 7.3.4.

8.4 Summary

In this chapter, we have presented three detailed case studies demonstrating that LeakWatch can be used to estimate the amount of information leaked by some complex Java programs. In each one, we demonstrated that LeakWatch quickly and correctly generates a sample that can be successfully analysed for information flows. It would be infeasible to perform this analysis in a system model such as CH-IMP's, which requires the precise joint probability distribution of the secret and publicly-observable information to be known: even though some of the code we provide in this

Figure 8.5: the performance of LeakWatch when analysing the program in the OpenPGP case study, with a varying number of bits in the encrypted message marked as observable



chapter is reasonably short, there is still an enormous amount of complexity behind the calls to various Java APIs and third-party libraries used by the code; encoding all of this in a formal model would be prohibitively difficult. By relaxing CH-IMP's system model and instead *approximating* the joint probability distribution of the secret and publicly-observable information that occurs in a program, we are able to meaningfully analyse the flow of information in many complex programs that we otherwise could not.

The main advantage of LeakWatch's information flow model is its "point-to-point" property inherited from CH-IMP's information flow model: because it answers questions of the form "approximately how much information does a passive attacker learn about secret values at these points by observing the values at these other points?", existing code can be analysed simply by inserting one-line calls to methods in LeakWatch's API for identifying secret and observable information, similarly to how CH-IMP programs can be analysed simply by inserting calls to the language's secret and observe commands: no further annotation or refactoring of code is necessary.

We have investigated the practical extent to which LeakWatch is able to analyse programs, and found that it is poor at analysing programs that contain a large amount of entropy in their secret and publicly-observable information (*e.g.* when a program's output is the ciphertext of a symmetric cipher). This prevents LeakWatch from being able to analyse *all* types of complex programs automatically — as we have seen, to successfully analyse these programs it is necessary to divide the secret or observable information into smaller, lower-entropy segments. However, provided that the program's secret and observable information can be divided in this way, LeakWatch is a convenient tool for analysing the information security of complex Java programs.

IV

Closing Statements

9

Future Work & Conclusions

9.1 Future Work

We conclude the thesis by suggesting possibilities for future directions that this research could take.

9.1.1 Narrower Bounds for Min-Entropy Leakage Estimates

In Section 3.3.2 (p. 70), during our review of the literature for estimating information-theoretic measures, we described work by Chothia et al. (2014) on computing min-entropy leakage estimates from samples of systems. The authors describe a method of deriving greater than 95% confidence intervals for these estimates, thus placing bounds on their accuracy. However, we saw in Section 7.1.2 (p. 175) that these confidence intervals are much wider than the ones for mutual information estimates for samples of an equivalent size (Section 7.1.1, p. 172); consequently, a much larger sample is required to place accurate bounds on the min-entropy leakage of a system. This is a side-effect of Chothia et al.'s method using Pearson's χ^2 test for goodness of fit, which requires larger samples for reliable results. Finding this distribution would remove the need to experimentally determine an empirical frequency distribution's goodness of fit with the expected frequency distribution using Pearson's χ^2 tests, and

would lead to the ability to derive much narrower confidence intervals for min-entropy leakage.

A forthcoming publication by Boreale and Paolini (2014) presents a technique for deriving narrower confidence intervals for min-entropy leakage in deterministic systems. They show that, when the probability distribution defining the system's inputs is relaxed slightly (*i.e.* allowing for the probabilities to be inexact), an estimator that provides excellent lower bounds on the confidence interval can be used; it also provides very good upper bounds if the input distribution induces an approximately uniform output distribution. To choose the input distribution, the authors rely upon random sampling using the Metropolis Monte Carlo method and another method based on rejection sampling. Our relaxed information flow model operates on probabilistic systems and does not sample a subset of the inputs, and it is unclear whether a technique similar to Boreale and Paolini's could be used to derive accurate bounds on the min-entropy leakage in such systems, but it is worthy of investigation.

9.1.2 Automated Identification of Publicly-Observable Information

Both the original and relaxed information flow models presented in this thesis require that the occurrence of a program's publicly-observable information is clearly marked (*i.e.* with CH-IMP's observe command, or the LeakWatch API's `observe()` method). It is often possible to infer where this information occurs in real-world programs; *e.g.*, in Java programs, information printed to the standard output stream (via `System.out`) or written to the `OutputStream` of a `File` or `Socket` object is typically observable. Locating these occurrences is achievable efficiently with a static analysis of the program's bytecode (similarly to how we identified the occurrence of input to a program via the standard input stream in Section 7.3.3 (p. 193) by analysing its bytecode for references to `System.in`). Provided that the parameters passed to these objects' methods were not too high-entropy to preclude a successful statistical information flow analysis of the program, LeakWatch could automatically infer that these method calls were points

where information becomes observable, with the parameters being the observable information that occurs.

9.1.3 Explaining why Information Flow Occurs in Programs

In the introductions to Chapters 6 and 8 (pp. 144 and 199) we emphasised that our information flow model identifies only that a flow of information occurs, but cannot explain *why* it occurs. To partially address this shortcoming, it may be possible to use feature selection algorithms from the machine learning literature to automatically identify which subset of the publicly-observable information maximises the chosen information leakage measure. This would require modifications to the information flow model; *e.g.*, to determine precisely which variables belong to the set of variables whose values maximise the flow of information in the program, it would be necessary to track the names of variables from which publicly-observable information arises. Given mutual information's popularity as a filter metric in the machine learning literature, there is a large body of existing work that could be applied directly to our model if mutual information were the chosen information leakage measure, but it is not clear how much of this work would be applicable to min-entropy leakage (or other measures).

9.2 Thesis Summary

This thesis has presented several advances in the field of quantitative information flow analysis.

Our primary contribution is the development of a novel "point-to-point" quantitative information flow model with a system model in which the occurrence of secret and publicly-observable information is unrestricted. This is an improvement over previous information flow models, whose system models generally make unsatisfactory assumptions about the behaviour of real systems: that they contain single pieces

of high-security and low-security information, process high-security information provided before execution and output low-security information upon termination, or a combination of these restrictions.

Our model quantifies the amount of information learned about a system's secrets by an attacker with knowledge of the system's behaviour and the ability to inspect the publicly-observable output it produces. This is achieved by modelling the execution of a system as a discrete-time Markov chain (DTMC) and computing the joint probability distribution of the secret and observable information that occurs in each accepting (*i.e.* terminating) state of the DTMC. This joint probability distribution can be used to compute a wide range of information-theoretic leakage measures to quantify the flow of information in the system; in this thesis we have focused on computing the min-entropy leakage from the system's secret information to its publicly-observable information and the mutual information of these two collections of information — different measures model attackers with different abilities and goals, and one can be chosen that best reflects the desired attacker model. We have demonstrated via many examples that our model is ideal for analysing the flow of information in small or medium-sized probabilistic programs and protocols, and have produced a software tool for doing so.

Given that this system model is based on DTMCs, and that execution of a system is characterised as the exploration of a DTMC, we eventually encounter the usual *state space explosion* problem; this prevents us from analysing complex systems using this information flow model. Our second contribution is a relaxed variant of our initial information flow model, in which the joint probability distribution of the secret and publicly-observable information that occurs in a system is estimated via sampling. This introduces the usual complications of sampling that would impede a meaningful information flow analysis of the system: ensuring that a sufficiently-sized sample has been collected, accounting for noise in the sample, *etc.* With the help of theoretical results from the statistics literature, we show that it is possible (and feasible) to accu-

rately approximate the mutual information and min-entropy leakage measures from samples of systems; our technique is limited by the required sample size relative to the amount of unique secret and publicly-observable information that occurs in the sample, but provided that a sufficiently large sample can be collected, it gives credible results. We have demonstrated that this relaxed model can be used to analyse the flow of information in much more complex programs, and have presented a software tool for doing so automatically for programs written in Java.

List of References

Mário S. Alvim, Miguel E. Andrés, and Catuscia Palamidessi. Information Flow in Interactive Systems. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, volume 6269 of *Lecture Notes in Computer Science*, pages 102–116, Paris, France, August–September 2010. Springer.

Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 141–153, Oakland, California, USA, May 2009. IEEE Computer Society.

William R. Bevier, Richard M. Cohen, and William D. Young. Connection Policies and Controlled Interference. In *Proceedings of the Eighth IEEE Computer Security Foundations Workshop (CSFW '95)*, pages 167–176, Kenmare, County Kerry, Ireland, June 1995. IEEE Computer Society.

Fabrizio Biondi, Axel Legay, Pasquale Malacaria, and Andrzej Wąsowski. Quantifying Information Leakage of Randomized Protocols. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, volume 7737 of *Lecture Notes in Computer Science*, pages 68–87, Rome, Italy, January 2013a. Springer.

Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, and Andrzej Wąsowski. QUAIL: A Quantitative Security Analyzer for Imperative Code. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 702–707, Saint Petersburg, Russia, July 2013b. Springer.

Michele Boreale and Michela Paolini. On Formally Bounding Information Leakage by Statistical Estimation. In *Proceedings of the 17th International Information Security Conference (ISC 2014)*, volume 8783 of *Lecture Notes in Computer Science*, pages 216–236, Hong Kong, China, October 2014. Springer. To appear.

David R. Brillinger. Some data analyses using mutual information. *Brazilian Journal of Probability and Statistics*, 18(2):163–182, December 2004.

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. Technical report, France Telecom R&D, November 2002. URL http://asm.ow2.org/current/asm-eng.pdf.

Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. OpenPGP Message Format, 2007. URL http://tools.ietf.org/html/rfc4880.

CBS News. PlayStation Network breach has cost Sony $171 million, May 2011. URL http://www.cbsnews.com/news/playstation-network-breach-has-cost-sony-171-million/. Accessed on September 24, 2014.

Konstantinos Chatzikokolakis, Tom Chothia, and Apratim Guha. Statistical Measurement of Information Leakage. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, Lecture Notes in Computer Science, pages 390–404, Paphos, Cyprus, March 2010. Springer.

David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

Han Chen and Pasquale Malacaria. Studying Maximum Information Leakage Using Karush-Kuhn-Tucker Conditions. In *Proceedings of the 7th International Workshop on Security Issues in Concurrency (SecCo'09)*, volume 7 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–15, Bologna, Italy, September 2009. Open Publishing Association.

Tom Chothia and Vitaliy Smirnov. A Traceability Attack against e-Passports. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security (FC 2010)*, volume 6052 of *Lecture Notes in Computer Science*, pages 20–34, Tenerife, Canary Islands, Spain, January 2010. Springer.

Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. A Tool for Estimating Information Leakage. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 690–695, Saint Petersburg, Russia, July 2013a. Springer.

Tom Chothia, Yusuke Kawamoto, Chris Novakovic, and David Parker. Probabilistic Point-to-Point Information Leakage. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 193–205, New Orleans, Louisiana, USA, June 2013b. IEEE Computer Society.

Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. LeakWatch: Estimating Information Leakage from Java Programs. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS 2014), Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 219–236, Wrocław, Poland, September 2014. Springer.

David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative Information Flow, Relations and Polymorphic Types. *Journal of Logic and Computation*, 15(2): 181–199, April 2005.

David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, August 2007.

CNET. Expert: Sony attack may have been multipronged, May 2011. URL http://www.cnet.com/news/expert-sony-attack-may-have-been-multipronged/. Accessed on September 24, 2014.

Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2nd edition, 2006.

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, May 1982.

Dorothy E. Denning and Peter J. Denning. Data Security. *ACM Computing Surveys*, 11 (3):227–249, September 1979.

Peter Elias, Amiel Feinstein, and Claude E. Shannon. A Note on the Maximum Flow Through a Network. *IRE Transactions on Information Theory*, 2(4):117–119, December 1956.

Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing Flexibility in Information Flow Control for Object-Oriented Systems. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, California, USA, May 1997. IEEE Computer Society.

Forbes. Data Breach Bulletin, June 2014. URL http://www.forbes.com/sites/katevinton/2014/06/23/data-breach-bulletin/. Accessed on September 24, 2014.

John E. Freund. *Introduction to Probability*. Courier, 1973.

Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MIFARE Classic. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS 2008)*, volume 5283 of *Lecture Notes in Computer Science*, pages 97–114, Málaga, Spain, October 2008. Springer.

Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, April 1982. IEEE Computer Society.

James W. Gray. Toward a Mathematical Foundation for Information Flow Security. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 21–34, Oakland, California, USA, May 1991. IEEE Computer Society.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, June 2009.

Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. In *Proceedings of the 2011 Marktoberdorf Summer School*, Marktoberdorf, Germany, August 2011.

Jonathan Heusser and Pasquale Malacaria. Measuring Insecurity of Programs. Technical report, Department of Computer Science, Queen Mary University of London, UK, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.222.3297.

Jonathan Heusser and Pasquale Malacaria. Quantifying Information Leaks in Software. In *Proceedings of the Twenty-Sixth Annual Computer Security Applications Conference (ACSAC 2010)*, pages 261–269, Austin, Texas, USA, December 2010. IEEE Computer Society.

Boniface Hicks, Kiyan Ahmadizadeh, and Patrick Drew McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)*, pages 153–164, Miami Beach, Florida, USA, December 2006. IEEE Computer Society.

Boniface Hicks, Dave King, and Patrick McDaniel. Jifclipse: development tools for security-typed languages. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 1–10, San Diego, California, USA, June 2007. ACM Press.

T. E. Hull and A. R. Dobell. Random Number Generators. *SIAM Review*, 4(3): 230–254, July 1962.

Information Commissioner's Office. British Pregnancy Advice Service fined £200,000, March 2014. URL http://ico.org.uk/news/latest_news/2014/british-pregnancy-advice-service-fined-200000-07032014. Accessed on September 24, 2014.

Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, June 1981.

Jaisook Landauer and Timothy Redmond. A Lattice of Information. In *Proceedings of the 6th IEEE Computer Security Foundations Workshop (CSFW '93)*, pages 65–70, Franconia, New Hampshire, USA, June 1993. IEEE Computer Society.

Pasquale Malacaria. Assessing security threats of looping constructs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 225–235, Nice, France, January 2007. ACM Press.

Pasquale Malacaria. Algebraic foundations for quantitative information flow. *Mathematical Structures in Computer Science*, 25(2):404–428, February 2015. To appear.

Stephen McCamant and Michael D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 193–205, Tucson, Arizona, USA, June 2008. ACM Press.

Ziyuan Meng and Geoffrey Smith. Calculating Bounds on Information Leakage Using Two-Bit Patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming*

Languages and Analysis for Security (PLAS 2011), San Jose, California, USA, June 2011. ACM Press.

Rudy Moddemeijer. On estimation of entropy and mutual information of continuous distributions. *Signal Processing*, 16(3):233–248, March 1989.

Chunyan Mu and David Clark. Quantitative Analysis of Secure Information Flow via Probabilistic Semantics. In *Proceedings of the Fourth International Conference on Availability, Reliability and Security (ARES 2009)*, pages 49–57, Fukuoka, Japan, March 2009a. IEEE Computer Society.

Chunyan Mu and David Clark. An Interval-based Abstraction for Quantifying Information Flow. In *Proceedings of the Seventh Workshop on Quantitative Aspects of Programming Languages (QAPL 2009)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 119–141, York, UK, March 2009b. Elsevier.

Chunyan Mu and David Clark. A tool: quantitative analyser for programs. In *Proceedings of the Eighth International Conference on Quantitative Evaluation of Systems (QEST 2011)*, pages 145–146, Aachen, Germany, September 2011. IEEE Computer Society.

Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 228–241, San Antonio, Texas, USA, January 1999. ACM Press.

Andrew C. Myers and Barbara Liskov. Complete, Safe Information Flow with Decentralized Labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, California, USA, May 1998. IEEE Computer Society.

Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 5*, 50 (302):157–175, July 1900.

Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic Quantitative Information Flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

Claude E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, July 1948.

Geoffrey Smith. Principles of Secure Information Flow Analysis. *Malware Detection*, 27:291–307, 2007.

Geoffrey Smith. On the Foundations of Quantitative Information Flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2009)*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302, York, UK, March 2009. Springer.

Geoffrey Smith. Quantifying Information Flow Using Min-Entropy. In *Proceedings of the Eighth International Conference on Quantitative Evaluation of Systems (QEST 2011)*, pages 159–167, Aachen, Germany, September 2011. IEEE Computer Society.

Stanford Report. Stanford students show that phone record surveillance can yield vast amounts of information, March 2014. URL http://news.stanford.edu/news/2014/march/nsa-phone-surveillance-031214.html. Accessed on September 24, 2014.

Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 Seconds: Hijacking with Hitag2. In *Proceedings of the 21th USENIX Security Symposium*, pages 237–252, Bellevue, Washington, USA, August 2012. USENIX Association.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–187, January 1996.

Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1st edition, 1993.

J. Todd Wittbold and Dale M. Johnson. Information Flow in Nondeterministic Systems. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 144–161, Oakland, California, USA, May 1990. IEEE Computer Society.

Andrew Chi-Chih Yao. Theory and Applications of Trapdoor Functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS 1982)*, pages 80–91, Chicago, Illinois, USA, November 1982. IEEE Computer Society.