Queen Mary University of London
School of Electronic Engineering and Computer Science

Automating Quantitative Information Flow

Jonathan Heusser

Thesis submitted for the Degree of Doctor of Philosophy of the University of London

13th June 2011

I declare that the work in this thesis is performed entirely by myself during the course of my PhD studies at Queen Mary, University of London and has not been submitted for a degree at this or any other university.

Abstract

Unprecedented quantities of personal and business data are collected, stored, shared, and processed by countless institutions all over the world. Prominent examples include sharing personal data on social networking sites, storing credit card details in every store, tracking customer preferences of supermarket chains, and storing key personal data on biometric passports.

Confidentiality issues naturally arise from this global data growth. There are continously reports about how private data is leaked from confidential sources where the implications of the leaks range from embarrassment to serious personal privacy and business damages.

This dissertation addresses the problem of automatically quantifying the amount of leaked information in programs. It presents multiple program analysis techniques of different degrees of automation and scalability.

The contributions of this thesis are two fold: a theoretical result and two different methods for inferring and checking quantitative information flows are presented.

The theoretical result relates the amount of possible leakage under any probability distribution back to the order relation in Landauer and Redmond's lattice of partitions [35]. The practical results are split in two analyses: a first analysis precisely infers the information leakage using SAT solving and model counting; a second analysis defines quantitative policies which are reduced to checking a k-safety problem. A novel feature allows reasoning independent of the secret space.

The presented tools are applied to real, existing leakage vulnerabilities in operating system code. This has to be understood and weighted within the context of the information flow literature which suffers under an apparent lack of practical examples and applications. This thesis studies such "real leaks" which could influence future strategies for finding information leaks.

Acknowledgments I am very thankful to my supervisor Pasquale Malacaria for leading and supporting my work, and for the countless meetings and chats we had about our research.

I also thank Jani Nurminen and Teemu Suopelto for reviews and the many talks we had about my research over the years. More thanks to more reviewers and coworkers: Han Chen, Yewande Lawson, Samia Faruq, and Tom Powell.

Contents

1 Introduction **7**

1.1 Thesis outline and contributions 8

2 Preliminaries **10**

2.1 While language syntax and semantics 10

2.2 Information flow and noninterference 11

2.3 Information theory 13

2.4 Partition and equivalence relations 16

2.5 Estimating a multinomial distribution 17

2.6 Summary of model assumptions 18

3 Lattice of Information **20**

3.1 Quantifying interference 20

3.1.1 Computing entropy and leakage 22

3.2 Lattice of information and leakage 22

3.2.1 Observations and indistinguishability 22

3.2.2 Order relation of equivalence relations 23

3.2.3 Measures on the lattice of information 24

3.3 Relating lattice order and entropy 27

3.4 Measuring leakage of programs 29

3.4.1 Transition system and observations over programs 29

3.4.2 Definition of measuring leakage 30

3.4.3 Lattice operations and leakage 32

3.5 Reasoning about loops in programs 35

3.5.1 Analytical approach 35

3.5.2 The random variable `NIterations` . . . 37
3.5.3 Basic loop leakage definitions . . . 39
3.5.4 Examples . . . 40
3.6 Loops in the lattice of information . . . 43
3.6.1 Algebraic interpretation . . . 44
3.6.2 Loops with collisions . . . 45

4 Exact Leakage Quantification **48**
4.1 Analysis . . . 49
4.1.1 Objective . . . 49
4.1.2 Algorithm . . . 49
4.1.3 Example . . . 50
4.2 Safe upper bound . . . 51
4.3 Case studies . . . 53
4.3.1 Square root . . . 53
4.3.2 Prime numbers . . . 55
4.4 Review of technique . . . 58

5 AQUA Tool **60**
5.1 Automation by applying self-composition . . . 62
5.1.1 K-safety and quantitative information flow . . . 64
5.2 Inferring QIF by SAT solving and model counting . . . 64
5.2.1 Core algorithms . . . 65
5.2.2 Implementation . . . 67
5.2.3 Worked example . . . 70
5.3 Experiments . . . 74
5.3.1 Comparison to Chapter 4 . . . 76
5.4 Application: database queries . . . 77
5.4.1 Database inference by examples . . . 78
5.5 Review of technique . . . 81
5.5.1 Improvements . . . 83

6 Applying Model Checking to Verify Leakage Policies **84**
6.1 Model of programs and distinctions . . . 86

6.1.1 Checking policies and k-safety . . . 87
6.2 Encoding distinction-based policies . . . 88
6.2.1 Bounded model checking . . . 90
6.2.2 Driver . . . 91
6.3 Checking policies in practice . . . 92
6.3.1 Modelling low input . . . 93
6.3.2 Modelling environments . . . 95
6.3.3 Modelling confidential values . . . 96
6.4 Experimental results . . . 97
6.4.1 Linux kernel . . . 98
6.4.2 Authentication checks . . . 103
6.5 Review of technique . . . 105
6.5.1 Improvements . . . 106

7 Related Work **107**
7.1 QIF tools . . . 107
7.1.1 Model checking & constraint solving . . . 107
7.1.2 Interval abstraction . . . 108
7.1.3 Network flow capacity & dynamic analysis . . . 111
7.1.4 Sampling channel capacity . . . 111
7.2 QIF theory . . . 112

8 Conclusion **114**

Chapter 1

Introduction

Unprecedented quantities of personal and business data are collected, stored, shared, and processed by countless institutions all over the world. Prominent examples are sharing personal data on social networking sites, storing credit card details in every store, tracking customer preferences of supermarket chains, and storing key personal data on biometric passports.

Data breaches – malicious or unintentional release of confidential information – naturally arise from this global data growth. There are multiple institutions in the US alone which track and research these breaches. For example, Verizon released the "2010 Data Breach Investigations Report" in collaboration with the US Secret Service which studies and categorises a selection of over 900 breaches and in excess of 900 million leaked data records in the last six years. The most affected sectors are the Financial Services (33%) followed by Hospitality (23%); however all industrial sectors are affected.

This global problem asks for solutions. Verizon's report mentions, amongst others, the following mitigation efforts: (1) ensure essential controls are met (2) eliminate unnecessary data and keep tabs on what is left (3) test and review web applications. How this is achieved in practice is however an unsolved challenge.

One strategy to prevent or at least contain such data breaches is to check the software which handles confidential information for leaks. The upcoming research area of quantitative information flow tries to achieve exactly that.

Ultimately, the goal is that quantifying information leakage is part of the software development cycle and appears as simple as type checking programs. However, the research is not yet there.

This dissertation addresses the question on how to automatically quantify the amount of leaked information in programs. It presents multiple program analysis techniques and implementations of different degrees of automation and scalability.

The chapters tell the story of increasingly more useful and applied analyses. It starts off with the foundational theory behind the works; the following chapters each describe an analysis which is motivated by the previous chapter. It finishes with an elegant and surprisingly simple algorithm to check information leakage policies in much more complex code than previously achieved.

1.1 Thesis outline and contributions

Chapter 2 covers preliminary technical details and motivates assumptions made throughout the thesis.

Chapter 3 describes the algebraic basis of quantitative information flow analysis. We show how random variables and programs can be represented as partitions. These partitions are shown to be useful building blocks to capture how programs leak confidential information and the subsequent chapters always compute the partition representation of programs. The chapter contains a fundamental result between the order of partitions and the possible leakage under any distribution.

Chapter 4 describes a first analysis which runs on a simple while language. It computes the leakage by complete enumeration of the whole confidential variable range. It implements the loop leakage formula of [38] and presents a safe upper bound for the leakage. Also, its inefficiency motivates more refined analyses.

Chapter 5 describes algorithms and an implementation which automatically and statically quantifies the exact leakage for a subset of ANSI C programs. The algorithms are based on SAT solving and model counting and implement a reachability analysis.

The chapter provides a small benchmark which demonstrates its performance and scalability for difficult and known instances of the relevant literature. Also, we describe an application of the tool which measures leakage in aggregated database queries. This unconventional application to an outside field merely shows a possible, different use of quantifying information leaks.

Chapter 6 describes a method which does not precisely compute leakage anymore. The idea is to bound leakage and ask instead the question whether a program satisfies or violates a given leakage bound. This is an instance of *checking* information leakage instead of *inferring* it which proves to be a more feasible approach. One novelty of this approach is that the analysis is independent of the confidential variable size. This allows the analysis to run on much more complex data types. A driver formulates the leakage policy and a model checker checks for violation.

The applications in this chapter are reported leakage vulnerabilities in the Linux Kernel and different authentication routines. Device drivers with known leakage vulnerabilities are checked for different leakage policies. Additionally, the officially applied patches are proved to reduce the information leakage.

Chapter 7 describes other existing tools and techniques of authors who work on quantifying leakage. It also lists all relevant techniques used within this thesis and refers to their original publications.

This research is sponsored by the EPSRC grant EP/F023766/1 with the title “Model Checking and Program Analysis for Quantifying Interference”. The main focus of the grant is the development of tools for quantifying interference.

Chapter 2

Preliminaries

This chapter introduces basic notation and theory used throughout the thesis. It also serves to define the implicit computational and model assumptions made in the rest of the document.

2.1 While language syntax and semantics

Throughout this work, concepts and analyses are explained in terms of the following while-language unless mentioned otherwise. The syntax and denotational semantics of commands of the language are shown in tables 2.1 and 2.2.

$$
\begin{aligned}
&C := \text{skip} \mid x = E \mid C;\ C \mid \text{if } B\ C\ C \mid \text{while } B\ C \\
&E := x \mid n \mid E\ +\ E \mid E\ -\ E \mid E\ *\ E \\
&B := \neg B \mid B \wedge B \mid B \vee B \mid E < E \mid E == E
\end{aligned}
$$

$$C \in \text{Cmd},\ E \in \text{Exp},\ B \in \text{BExp},\ x \in \text{Var},\ n \in \mathbb{N}$$

Table 2.1: Syntax of While language

Members of the set of states Σ are denoted $\sigma : \text{Var} \to \mathbb{N}$, with Var being the set of variable identifiers. Arithmetic expressions are interpreted as the map $[\![E]\!] : \Sigma \to \mathbb{N}$ and boolean expressions have the interpretation $[\![B]\!] : \Sigma \to$

$$
\begin{aligned}
[\![skip]\!] &= \lambda\sigma.\sigma \\
[\![x = E]\!] &= \lambda\sigma.\sigma[[\![E]\!]\sigma/x] \\
[\![C_1; C_2]\!] &= [\![C_2]\!] \circ [\![C_1]\!] = \lambda\sigma.[\![C_2]\!]([\![C_1]\!]\sigma) \\
[\![\text{if } B\ C_1\ C_2]\!] &= \lambda\sigma. \begin{cases} [\![C_1]\!]\sigma, & \text{if} [\![B]\!]\sigma = 1 \\ [\![C_2]\!]\sigma, & \text{if} [\![B]\!]\sigma = 0 \end{cases} \\
[\![\text{while } B\ C]\!] &= \text{lfp } F \text{ where } F(f) = \lambda\sigma. \begin{cases} (f \circ [\![C]\!])\sigma, & \text{if} [\![B]\!]\sigma = 1 \\ \sigma, & \text{if} [\![B]\!]\sigma = 0 \end{cases}
\end{aligned}
$$

Table 2.2: Denotational Semantics of While language

$\{0, 1\}$. A command C is a state transformer map $[\![C]\!] : \Sigma \to \Sigma$ with the usual interpretation of least fix point of a function F for loops [64]. We further assume that in general commands are terminating, but in some contexts nontermination can be included as observable state.

A program P is a sequence of commands using sequential composition. For an initial store σ, $[\![P]\!]\sigma$ computes the final variable store. As we only consider input/output semantics, $[\![P]\!]$ can be seen as set of all executions $[\![P]\!] = \{(\sigma, [\![P]\!]\sigma), (\sigma', [\![P]\!]\sigma'), \dots\}$ where every element of the set is a single execution (input/output tuple). A subset $T \subseteq [\![P]\!]$ is a selection of executions.

2.2 Information flow and noninterference

This section introduces concepts from the secure information flow literature used in this thesis. Denning introduced a lattice model of information flow where variables are partitioned into security labels [23]. The confidential variables have a "high" security label H and the public variables have a "low" security label L. All low variables are publicly observable while the high variables contain confidential information and are kept secret. A partial order describes the allowed flows in a system, where $L \leq H$ allows flows up the lattice and disallows flows from H to the lower label L.

Goguen and Meseguer describe, in a general automaton framework, the idea of *noninterference* as a requirement for checking if a certain security pol-

icy holds [26]. Their definition works on groups of users where, given noninterference, each group can not see the effect of commands used by any other group.

Volpano, Smith, and Irvine described the lattice model of information flow and noninterference in a language-based setting [63]. The authors proved the soundness of Denning's analysis using a type system which coincides with the idea of noninterference.

Definition 1 (Noninterference)**.** *For any two states σ_1 and σ_2 which agree on the values of all low variables $v \in L$, $\sigma_1(v) = \sigma_2(v)$, also satisfy the same equality after the execution of a terminating program P:*

$$(P(\sigma_1))(v) = (P(\sigma_2))(v)$$

This definition is checked syntactically using type system rules where the security labels are part of the type.

From an automation viewpoint a more useful and precise way of checking secure information flows in a program is the semantic approach by Leino and Joshi [36]:

Definition 2 (Secure information flow – Leino-Joshi)**.** *With the assignment of an arbitrary value to all high variables denoted by HH, a program P is noninterfering if the following equation is satisfied:*

$$HH; P; HH = P; HH$$

Here the fragment $HH; P$ describes a program running on an arbitrary high value, where $P; HH$ indicates that after the program is run all high values are "forgotten". The equality of the two, as in the equation above, together with a $; HH$ on both sides indicates that only low variables are known after the execution and that the execution does not depend on the starting value of the high variables.

This semantic approach has multiple benefits especially with our quantitative goals in mind: it is less conservative than a type system approach. A type system will have to reject any program which contains a sub program which

on its own is noninterfering. It can be applied to any language construct with definable semantics, including nondeterminism and it is more useful in the context of automated verification than a type-based approach.

Leino and Joshi's definition will be used in later chapters to motivate a verification approach to quantifying information leaks.

2.3 Information theory

This section contains a very short review of some basic definitions of Information Theory; additional background is readily available in the standard textbook by Cover and Thomas [19].

Definition 3 (Entropy)**.** *Given a space of events $X = (x_i)_{i \in N}$ with the probability mass function $p(x_i)$ for event x_i, Shannon's entropy is defined as*

$$H(X) \triangleq -\sum_{x_i \in X} p(x_i) \log p(x_i) \tag{2.1}$$

Informally, the entropy measures the average information content of the set of events: in the extreme case of an event with probability 1 the entropy will be 0 and on the other hand, if the distribution is uniform with every event equally likely then the entropy is maximal, i.e. $\log |N|$. In the literature the terms information content and uncertainty in this context are often used interchangeably. Both terms refer to the number of possible distinctions on the set of events.

The cardinality of the space of events is important in this work, thus we are going to prove the information theoretical result when entropy attains its maximal value.

$$H(X) \leq \log |N| \tag{2.2}$$

Proof. Jensen's Inequality is used for the proof which states that for any convex function f and random variable X the following holds

$$E[f(X)] \geq f(E[X])$$

with $E[\cdot]$ being the expected value of random variable X. For a discrete random variable with probability mass function $p(x_i)$ this is $\sum_X x_i\ p(x_i)$.

Shannon entropy itself can also be seen as such an expectation, or mean value, of a function $f(u)$ where $f = \log_2$ and $u = \frac{1}{p(x)}$. As entropy is concave we invert Jensen's inequality and get

$$H(X) = E[\log_2(\frac{1}{p(x)})] \leq \log_2(E[\frac{1}{p(x)}])$$

then by noticing that $E[\frac{1}{p(x)}] = \sum_N p(x)\frac{1}{p(x)} = |N|$ we arrive at

$$H(X) \leq \log_2 |N|$$

where the equality holds only when $\frac{1}{p(x)}$ is constant, thus enforcing uniform distribution. □

Definition 4 (Joint Entropy)**.** *Given two random variables X and Y, the joint entropy $H(X,Y)$ measures the uncertainty of the joint random variable (X,Y). It is defined as*

$$H(X,Y) \triangleq -\sum_{x\in X, y\in Y} p(X=x, Y=y)\log p(X=x, Y=y)$$

where $p(X=x, Y=y)$ is the joint probability mass function defined as

$$p(X=x|Y=Y)\ p(Y=y)$$

Definition 5 (Conditional Entropy)**.** *Conditional entropy $H(X|Y)$ measures the uncertainty about X given the knowledge of Y. It is defined in terms of the chain rule*

$$H(X|Y) \triangleq H(X,Y) - H(Y)$$

The higher $H(X|Y)$ is, the lower is the correlation between X and Y. It is easy to see that if X is a function of Y, then $H(X|Y) = 0$ (there is no uncertainty on X knowing Y if X is a function of Y) and if X and Y are independent then $H(X|Y) = H(X)$ (knowledge of Y does not change the uncertainty on X if they are independent) .

Definition 6 (Mutual Information)**.** *Mutual information $I(X;Y)$ is a measure of how much information random variables X and Y share. It is defined in terms of conditional entropy*

$$I(X;Y) \triangleq H(X) - H(X|Y) = H(Y) - H(Y|X)$$

Thus the information shared between X and Y is the information of X (respectively, Y) from which the information about X given Y has been deduced. This quantity measures the relationship between X and Y. For example X and Y are independent iff $I(X;Y) = 0$.

Mutual information is a measure of binary *interaction*. Conditional mutual information, a form of ternary interaction will be used to quantify *leakage*.

Definition 7 (Conditional Mutual Information)**.** *Conditional mutual information measures the relationship between two random variables X and Y conditioned on a third random variable Y. It is defined as*

$$I(X;Y|Z) \triangleq H(X|Z) - H(X|Y,Z) = H(Y|Z) - H(Y|X,Z)$$

where $H(Y|X,Z)$ is read as $H(Y|(X,Z))$.

A further important quantity based on mutual information is the channel capacity.

Definition 8 (Channel Capacity)**.** *Given two random variables H and O which represent the input and output of a system respectively, the channel capacity is defined as the maximal mutual information between H and O. It is defined as*

$$\max_{\mu} I_{\mu}(O;H)$$

where μ is the distribution on the input which maximises the mutual information.

For a single random variable O, $CC(O)$ denotes the channel capacity of O as follows

$$CC(O) \triangleq \max_{\mu} H_{\mu}(O)$$

Here, μ is the distribution on the inputs of O which make the outputs of O most uniform, as a result of equation 2.2.

2.4 Partition and equivalence relations

Partitions are heavily used to represent fundamental structures in this thesis and the terms *partition* and *equivalence relation* are used interchangeably from now on.

A partition Π of a set S is a family of subsets of S such that the following conditions are satisfied

- Every block in Π is nonempty
- Every element in S is contained in exactly one block of Π

where a block of Π is simply an element of Π.

Given the set $S = \{a, b, c, d\}$ then the following is an example partition of that set

$$\Pi = \{\{a\}\{b, c\}\{d\}\}.$$

An equivalence relation on a set S satisfies a reflexive, symmetric, and transitive relation on S. For example, the function $\lambda n.\ \mod n$ describes an equivalence relation $\sim_{\mod n}$ on a set of integers S for any integer n. Taking $S = \{1, 2, 3, 4\}$ and $n = 2$ then

$$\sim_{\mod 2} = \{\{1, 3\}\{2, 4\}\}$$

where $1 \sim_{\mod 2} 3$ denote that the two integers are seen as equivalent or related under the given relation. Also, an equivalence class of $s \in S$ with respect to an equivalence relation $\sim$ is defined as $[s]_{\sim} = \{s' \in S | s' \sim s\}$. Thus, an equivalence relation determines a partition of a given set where each block of the partition is described by an equivalence class of the equivalence relation.

Equally, a partition Π of a set S determines a specific equivalence relation on that set. This equivalence relation is described for every $s, s' \in S$. $s \sim s'$ if and only if s and s' are in the same block in partition Π.

2.5 Estimating a multinomial distribution

Probability distributions are not a focus of the work presented in this thesis. Uniform distribution is mostly used to assign probabilities to outcomes and it is often just assumed that $\frac{x_i}{n}$ is the "right" estimate for the probability of an observed event, where x_i is the number of occurrences of event i and n the total number of observations.

Usually, an event is that a program evaluates to value o with input h, where for example if 10 out of 100 tested inputs lead to value o then the probability of that event is assumed to be $\frac{10}{100}$. The following derivation shows that this is a reasonable estimate.

The multinomial distribution is suitable to model the probabilities of k equivalence classes by their cardinality. We assume to have k possible outcomes, the random variables $X_i = x_i$ describe the number of times outcome i was observed over n trials. The probabilities are described by $p_1, \ldots, p_k$, where $p_i = P(X_i = x_i)$. The probability mass function is

$$f(x_1, \ldots, x_k; n, p_1, \ldots, p_k) = P(X_1 = x_1, \ldots, X_k = x_k) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k}$$

with $\sum_{i=1}^{k} x_i = n$.

We assume the data is given, in other words we have observed the vector $x = (x_i, \ldots, x_k)$. The task now is to assign the most likely probabilities to these counts. This can be shown using a maximum likelihood estimate, by starting with the log likelihood, where $l(p_1, \ldots, p_k) = f(x_1, \ldots, x_k; n, p_1, \ldots, p_k)$:

$$L = \log l(p_1, \ldots, p_k, \lambda) = \log n! - \sum_k \log x_i! + \sum_k x_i \log p_i + \lambda(1 - \sum_k p_i)$$

which is simply the logarithm of the probability mass function and an additional lagrange multiplier encoding the constraint of a probability distribution. This likelihood has to be maximised by setting all derivatives over all arguments to 0. This is simplified greatly because the first two summands are

constant and results in:

$$\frac{dL}{dp_i} = \frac{x_i}{p_i} - \lambda = 0$$
$$\frac{dL}{d\lambda} = 1 - \sum_k p_i = 0$$

To find the value of λ we multiply the first derivative by p_i and plug in the sum over k:

$$\begin{aligned} &\sum_k x_i - \lambda \sum_k p_i = 0 \\ =&\sum_k x_i - \lambda\ 1 = 0 \\ =&n = \lambda \end{aligned}$$

Thus, we can substitute the newly found λ in the first derivative and arrive at the maximum likelihood estimate

$$p_i = \frac{x_i}{n} \tag{2.3}$$

This estimator of equivalence class probabilities can be shown to be unbiased and consistent; unbiased because its expectation is the true probability and consistent because the more data points there are the better the estimate.

2.6 Summary of model assumptions

Unless specified otherwise the following assumptions hold throughout the document: all languages are deterministic; all programs terminate; semantics consider input/output states only; all random variables follow uniform distribution; logarithms are base 2.

Table 2.1 summarises the notation and symbols used in this thesis.

$E[\cdot]$	Expected value
$H(\cdot), H(\cdot\|\cdot)$	(Conditional) Shannon entropy
$I(\cdot\ ;\ \cdot), I(\cdot\ ;\ \cdot\|\cdot)$	(Conditional) Mutual information
X, Y, O, H, L	Random variables
h, l, o	Program variables
$[\![C]\!]$	Input-output semantics of command C
Σ	Set of all states/atoms in the lattice
σ_i, σ'	Members of Σ
$\{\{a, b\} \cdots\}$	Partition with block containing states a, b
$\sim, \simeq, \approx$	Equivalence relations
$\mathcal{I}(\Sigma)$	Set of all partitions
$\top, \bot$	Top and bottom element in the lattice
$\sqcup, \sqcap$	Lattice join and meet
$\sqsubseteq$	Partial order on partitions

Figure 2.1: Notation

Chapter 3

Lattice of Information

This chapter describes information leakage in programs using information theory. It introduces and motivates the use of partitions as a central element in the computation of leakage and recasts existing work [38] in terms of partitions.

The content has been published in the proceedings of Formal Methods for Quantitative Aspects of Programming Languages (SFM 2010) in LNCS [41]. Previous research on the same content has been presented at the Workshop Quantitative Analysis of Software (QA 2009) which had proceedings in a Berkeley Technical Report [28], and has been presented at the Workshop on Programming Language Interference and Dependence (PLID 2009).

For the syntax and semantics of language code used in this chapter, please refer to section 2.1.

3.1 Quantifying interference

This section describes the work by Clark, Hunt, and Malacaria [13] as a brief introduction to quantifying interference using information theory, or what will also be called leakage.

The starting point is the usual assumption that program variables are partitioned into two sets, H "high" and L "low". High variables contain confidential or secret information which are never directly visible to an attacker. Low variables on the other hand are publicly observable at any point during

the execution, depending on the power of an attacker. *Leakage* is the amount of information one can learn about H by observing L.

The leakage quantity is measured using information theory. This chapter describes the connection between information theoretical measures of leakage and programs.

The most central element of information theory are random variables. In our setting, a random variable is a map $X : D \to R$, where D and R are finite sets and D has a probability distribution μ associated with it. We let x range over the values which X can take on, and similarly elements in the sample space D are denoted d.

The probability that X takes on value x, written $p(x)$, is simply the sum of the probabilities in X's preimage

$$p(x) \stackrel{def}{=} \sum_{d \in X^{-1}(x)} \mu(d)$$

Next, as entropy is a function of $p(x)$

$$H(X) = -\sum_{x} p(x) \log_2 p(x)$$

it follows that the preimage of X completely defines its entropy. Or again in other words, the random variable X partitions its sample space D into blocks which are indistinguishable to an observer who only sees $X = x$. This partitioning is formalised by the kernel of a function $f : A \to B$ which is the equivalence relation $\simeq_f$ where

$$a_1 \simeq_f a_2 \iff f(a_1) = f(a_2). \tag{3.1}$$

Thus, if two random variables have the same kernel they are said to be observationally equivalent and they also agree on their entropy.

We are going to adopt the deterministic model of information flow from [13]. This model describes the information flow from a joint input $\langle \text{High}, \text{Low} \rangle$ (where for readability the random variables are written as words instead of capital letters) to the output random variable $O = f(\langle \text{High}, \text{Low} \rangle)$ where f

is a deterministic function representing a program. The information flow, or leakage from High to O is shown to be the conditional entropy of O given Low

$$F_{\text{Low}}(\text{High} \rightsquigarrow O) = H(O|\text{Low})$$

This is the case in systems defined by simple imperative programs without nondeterministic language constructs.

3.1.1 Computing entropy and leakage

Ultimately, we are interested in computing the entropy $H(O|\text{Low})$ which quantifies the leakage in a program. One key aspect of the entropy calculation is that there is a qualitative and quantitative part. The qualitative part is the equivalence relation derived from the random variable $O|\text{Low}$, the quantitative part is computing $p(o|l)$ for all $o \in O, l \in \text{Low}$. These two parts however can be separated. First the equivalence relation can be computed, then given μ, the quantification can take place. This is the overarching strategy in this work and we mainly focus on computing the qualitative part while always keeping enough quantitative properties of $O|\text{Low}$ intact.

This chapter describes the general structure of such equivalence relations in more detail and connects them with observations and programs.

3.2 Lattice of information and leakage

3.2.1 Observations and indistinguishability

The concept of an *observation* is a key element in our model. The leakage quantity can essentially be seen as the result of a backwards reasoning process of an attacker, from the outputs – or observations – of a program to the high values at the input of the program.

Given a specific observable $O = o$ an attacker who would like to find the high part of this computation is left with no choice but to assume that it was one of the elements in the preimage $O^{-1}(o)$. The set of high values in the

preimage are in the context of equation 3.1 *indistinguishable* from each other. Thus, one observable describes an equivalence class of high values.

Naturally, one can always define a least informative observation (no observations can be made) and a most informative observation (every element is distinguishable). An equivalence class is then the set of high values that are indistinguishable by the corresponding observation.

3.2.2 Order relation of equivalence relations

Given a system with a set of possible states Σ, the set of all possible partitions over Σ is a *complete lattice*: the Lattice of Information (LoI) [35]. The order on partitions is given by refinement: a partition is "above" another if it is more informative, i.e. each block in the lower partition is larger or equal to a block in the partition above.

An alternative view of the same structure is in terms of *equivalence relations*. There is a simple translation between an equivalence relation and a partition: an equivalence relation defines the partition whose blocks are equivalences classes. The terms equivalence relation and partition are used synonymously.

Let us define the set $\mathcal{I}(\Sigma)$ which stands for the set of all possible equivalence relations on a set Σ. The ordering of $\mathcal{I}(\Sigma)$ is now defined as

$$\approx \sqsubseteq \sim \leftrightarrow \ \forall \sigma_1, \sigma_2 \ (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_1 \approx \sigma_2) \tag{3.2}$$

where $\approx, \sim \in \mathcal{I}(\Sigma)$ and $\sigma_1, \sigma_2 \in \Sigma$. Furthermore, the join $\sqcup$ and meet $\sqcap$ lattice operations are the intersection of relations and the transitive closure union of relations, respectively. Thus, higher elements in the lattice can distinguish more while lower elements in the lattice can distinguish less states. It easily follows from (3.2) that $\mathcal{I}(\Sigma)$ is a complete lattice.

We assume this lattice to be finite; this is motivated by the finite bit width of program variables: a k bit variable has 2^k possible values. This assumption can be generalised to an infinite lattice which is however not considered in this work.

A typical example of how these equivalence relations can be used in an

information flow setting is the following [68]. Let us assume the set of states Σ consists of a tuple $\langle h, l\rangle$ where l is a *low* variable and h is a confidential *high* variable. One possible program can be described by the equivalence relation

$$\langle h_1, l_1\rangle \approx \langle h_2, l_2\rangle \leftrightarrow l_1 = l_2$$

That is the observer can only distinguish states which agree on the low variable part. Clearly, a more revealing program is one which distinguishes any two states from one another, or

$$\langle h_1, l_1\rangle \sim \langle h_2, l_2\rangle \leftrightarrow l_1 = l_2 \wedge h_1 = h_2$$

The $\sim$-program reveals more information than the $\approx$-program by comparing states, therefore $\approx \sqsubseteq \sim$.

3.2.3 Measures on the lattice of information

Let us attempt to quantify the amount of information provided by a point in the lattice of information.

One possibility is to take the cardinality of a partition P as its measure, which results in block counting: $|P|$ = "number of blocks in P". This measure is 1 for the least informative partition and its maximal value is reached by the top partition. This choice of measure reflects the lattice order because if $A \sqsubseteq B$ then $|A| \leq |B|$. However, the important property of "additivity" for measures, the inclusion-exclusion principle, is not satisfied. In terms of sets, the inclusion-exclusion principle states that the number of elements in a union of sets is the sum of the number of elements of the two sets minus the number of elements in the intersection[1]. In this example, the inclusion-exclusion principle is expressed as

$$|A \sqcup B| = |A| + |B| - |A \sqcap B|$$

[1]The principle is universal e.g. in propositional logic the truth value of $A \vee B$ is given by the truth value of A plus the truth value of B minus the truth value of $A \wedge B$.

To demonstrate the contradiction, let us take the following two partitions

$$A = \{\{1,2\}\{3,4\}\},\ B = \{\{1,3\}\{2,4\}\}$$

then their join and meet will be

$$A \sqcup B = \{\{1\}\{2\}\{3\}\{4\}\},\ A \sqcap B = \{\{1,3,2,4\}\}.$$

The counting principle from above is in this case not satisfied

$$|A \sqcup B| = 4 \neq 3 = |A| + |B| - |A \sqcap B|$$

Another problem with the cardinality $|\cdot|$ is that when $\mathcal{I}(\Sigma)$ is considered as a lattice of random variables the measure may end up being too coarse, as it discards all probabilities. To address this problem we introduce more abstract lattice theoretic notions.

A valuation on $\mathcal{I}(\Sigma)$ is a real valued map $\nu : \mathcal{I}(\Sigma) \to \mathbb{R}$ which satisfies the following properties:

$$\nu(X \sqcup Y) \ = \nu(X) + \nu(Y) - \nu(X \sqcap Y) \tag{3.3}$$

$$X \sqsubseteq Y \ \text{ implies } \ \nu(X) \leq \nu(Y) \tag{3.4}$$

A join semivaluation is a weak valuation, i.e. a real valued map where property 3.3 is relaxed to an inequality

$$\nu(X \sqcup Y) \leq \nu(X) + \nu(Y) - \nu(X \sqcap Y) \tag{3.5}$$

for every element X and Y in a lattice [51]. The property 3.4 is order-preserving: a higher element in the lattice has a larger valuation than elements below itself. The property 3.5 is a weakened inclusion-exclusion principle.

Proposition 1. *The map*

$$\nu(X \sqcup Y) \triangleq H(X,Y) \tag{3.6}$$

is a join semivaluation on $\mathcal{I}(\Sigma)$.

Proof. The tricky part is to prove that inequality 3.5 is satisfied. Since it is true that

$$H(X,Y) = H(X) + H(Y) - I(X;Y)$$

it will be enough to prove that

$$H(X \sqcap Y) \leq I(X;Y)$$

This can be proved by noticing the following two facts

1. $H(X \sqcap Y) = I(X \sqcap Y; X)$ this is clear because $I(X \sqcap Y; X)$ measures the information shared between $X \sqcap Y$ and X and because $X \sqcap Y \sqsubseteq X$ such measure has to be $H(X \sqcap Y)$

2. $I(X \sqcap Y; X) \leq I(Y; X)$ this is clear because $X \sqcap Y \sqsubseteq Y$ hence there is more information shareable between Y and X than between $X \sqcap Y$ and X

Putting those two items together, it follows that

$$H(X \sqcap Y) = I(X \sqcap Y; X) \leq I(Y; X).$$

□

An important result proved by Nakamura [51] gives a particular importance to Shannon entropy as a measure on $\mathcal{I}(\Sigma)$. He proved that the *only* probability-based join semivaluation on the lattice of information is Shannon's entropy. It is easy to show that a valuation itself is not definable on this lattice, thus Shannon's entropy is the best approximation to a probability-based valuation on this lattice.

Other measures can be used, which are however less mathematically appealing. Min-Entropy, used recently by Smith in an information flow context [57], could be considered as it seems like a good, complementing measure. While Shannon entropy intuitively results in an "averaging" measure over a

probability distribution, the Min-Entropy H_∞ takes on a "worst-case" view: only the maximal value $p(x)$ of a random variable X is considered

$$H_\infty(X) = -\log \max_{x \in X} p(x)$$

where it is always the case that $H_\infty(X) \leq H(X)$.

Another useful tool when working with partitions is considering conditioning of two partitions X and Y. The conditional partition $Y|X = x$ (where x is a block in X) is the intersection of all blocks in Y with x; given $Y|X = x$ a probability distribution is achieved by normalising the probabilities, with normalisation factor $p(x)$.

The notation $Y|X = x$ is justified because $H(Y|X = x)$ is the usual notion of information theoretical entropy of the variable Y given the event $X = x$. Formally,

$$Y|X = x \equiv \{y \cap x | y \in Y\} \tag{3.7}$$

and the probability distribution associated to $Y|X = x$ is

$$\{\frac{p(y \cap x)}{p(x)} | y \in Y\}. \tag{3.8}$$

3.3 Relating lattice order and entropy

The choice of using partitions as basic building block for the quantification of leakage and to only assume uniform distribution may seem overly restrictive or not expressive enough. The following two results however justify that decision. The first states that the order relation imposes bounds on the possible entropy of a random variable for any distribution. The second one restates the first under the worst case distribution, the channel capacity.

Theorem 1.

$$X \sqsubseteq Y \iff \forall \mu. H_\mu(X) \leq H_\mu(Y)$$

Proof.

$\Longrightarrow$ direction follows from entropy being a semi valuation.

$\Longleftarrow$ direction by contraposition. We want to prove that

$$X \not\sqsubseteq Y \implies \exists \mu . H_\mu(X) \not\leq H_\mu(Y)$$

by taking two partitions where a block of Y is refined in X

$$Y = \{\{1,2\}\{3,4\}\{5\}\{6\}\}$$
$$X = \{\{1,2\}\{3\}\{4\}\{5\}\{6\}\}$$

where it is clear that $X \not\sqsubseteq Y$. Next, as choice of μ we assign zero probability to all elements of the input space with the exception of the refining elements 3 and 4 which receive probabilities 0.1 and 0.9 respectively. Thus it is easy to confirm that

$$-(0.1+0.9)\log_2(0.1+0.9) < -0.1\log_2(0.1) - 0.9\log_2(0.9)$$
$$H(Y) < H(X)$$

In general, when blocks $X_1, \ldots, X_i$ refine block Y_1 then choosing the probability distribution $\mu(Y_1) = 1$ leads to $H(Y) = 0 < H(X)$. □

Here, $CC(X)$ is the channel capacity of X as defined in section 2.3. The channel capacity is achieved by the distribution which makes the probability of equivalence classes uniform[2].

Proposition 2.

$$X \sqsubseteq Y \implies CC(X) \leq CC(Y)$$

Proof. Y refines X, thus Y has equal or more equivalence classes than X. Its larger cardinality implies larger channel capacity because channel capacity is $\log_2$ of the cardinality (see equation 2.2 in chapter 2). □

[2]not to be confused with uniform input distribution

3.4 Measuring leakage of programs

3.4.1 Transition system and observations over programs

The previous sections defined the lattice and lattice order in terms of arbitrary states Σ. To model programs we assume states to be part of a transition system (Σ, I, F, T) where

1. Σ is the set of states,
2. I is the set of initial states ($I \subseteq \Sigma$),
3. F is the set of final states,
4. $T \subseteq \Sigma \times \Sigma$ is a the transition relation.

Let us define a successor function for a state $\sigma \in \Sigma$

$$\mathrm{Post}(\sigma) = \{\sigma' \in \Sigma \mid (\sigma, \sigma') \in T\}$$

A state σ is in F if $\mathrm{Post}(\sigma) = \emptyset$. A path is a finite sequence of states $\pi = \sigma_0\sigma_1\sigma_2 \ldots \sigma_n$ such that $\sigma_0 \in I$ and $\sigma_n \in F$. Also, a state is a tuple $\sigma = \sigma_H \times \sigma_L$ of the pair of confidential input H and low input L. The input-output semantic valuation mapping $[\![P]\!] : \Sigma \to \Sigma$ evaluates P for a given initial state and outputs its final state.

Now, an observation over a program P is the equivalence relation on high initial states $\sigma_H \in I$ induced by $[\![P]\!]$. A particular equivalence class will be called an observable. Hence an observable represents a set of states indistinguishable to an attacker making that observation.

A program may or may not have initial low variables whose values are controlled by an attacker. This control is formalised as the ability to set the low part of a state before the execution of the program. For the general case, we assume that the attacker has control over some low variables, where one particular equivalence relation $\simeq_l$ with value l for the initial low state as follows

$$\forall \sigma, \sigma' \in I.\ \sigma_H \simeq_l \sigma'_H \iff [\![P]\!](\sigma) = [\![P]\!](\sigma') \wedge \sigma_L = \sigma'_L = l \tag{3.9}$$

Thus the equivalence relation relates the high states which under a particular low value l result in the same observable output.

We denote this interpretation of a program P in $\mathcal{I}(\Sigma)$ as defined by the equivalence relation (3.9) by $\Pi_l(P)$. The relation $\Pi_l(P)$ is nothing else than the kernel of the semantics of P.

In the simple case where an attacker can not set the low state, the equivalence relation reduces to

$$\forall \sigma, \sigma' \in I.\ \sigma_H \simeq \sigma'_H \iff [\![P]\!](\sigma) = [\![P]\!](\sigma') \tag{3.10}$$

This equivalence relation will be referred to as $\Pi(P)$. To ease readability this equivalence relation is mostly used throughout the chapters unless the attacker is specifically given the choice to initialise the low initial states.

The distinction between $\Pi(P)$ and $\Pi_l(P)$ is only important if the attacker has the ability to compare equivalence relations resulting from multiple low values, or if the attacker has external knowledge that one particular low value leads to an especially informative equivalence relation.

3.4.2 Definition of measuring leakage

As the lattice framework and program model are now connected, we can develop the notion of leakage. Let us start with the following intuitive statement:

The leakage of confidential information of a program is defined as the difference between an attacker's uncertainty about the secret before and after observing the outputs of the program.

For a Shannon-based measure, the above statement is traditionally [13] expressed in terms of conditional mutual information. The uncertainty about the secret by the attacker before observations is $H(H|L)$ and the uncertainty after observations is $H(H|L, \Pi(P))$. Using the definition of conditional mutual information, leakage is defined as

$$H(H|L) - H(H|L, \Pi(P)) = I(H; \Pi(P)|L)$$

We can now simplify the above definition as follows

$$\begin{aligned} I(\Pi(P); H|L) &= H(\Pi(P)|L) - H(\Pi(P)|L, H) \\ &=_A H(\Pi(P)|L) - 0 = H(\Pi(P)|L) \\ &=_B H(\Pi(P)) \end{aligned} \tag{3.11}$$

where equality A holds because the program is deterministic and B holds when the program only depends on the high inputs, for example when all low variables are initialised in the code of the program. The leakage for such programs is then defined as

Definition 9 (Leakage)**.** *The (Shannon-based) leakage of a program P is defined as the (Shannon) entropy of the partition $\Pi(P)$.*

Notice that the above definition can easily be adapted to other real valued maps from the lattice of information, providing possibly different definitions of leakage: $\Pi(P)$ provides a very general representation that can be used as the basis for several quantitative measures likes Shannon's entropy, Renyi entropies or guessability measures.

Definition 10 (Run)**.** *A run of a program P is defined to be a single realisation of equation 3.9 resulting in one equivalence relation $\Pi(P)$ or $\Pi_l(P)$.*

Definition 11 (Evaluation)**.** *An evaluation of a program P is defined to be the execution of the program from an initial to a final state, according to its semantics.*

Please note the following distinction which is made in this work: a program P, before it is evaluated, always starts with a well-defined initial state, such that there are no uninitialised variables. This implies that the Shannon-entropy of a program is always just the entropy of its partition without conditioning on the low variables. This is necessary to distinguish the entropy of a single-run of the program (the equivalence relation resulting from one fixed initial state) with the entropy of multiple runs of the program within the same framework. There are three cases to distinguish:

- Single-run leakage which is described by $\Pi(P)$ or for a particular choice of l as $\Pi_l(P)$
- Single-run maximising leakage which can be achieved by a powerful (or lucky) attacker in one run which results in the maximal equivalence relation (in terms of cardinality) of $\Pi_l(P)$
- Multi-run leakage where an attacker is able to combine the leakage of multiple runs by taking the least upper bound of the resulting partitions $\bigsqcup_L \Pi_l(P)$ for some set of low inputs L.

Unless mentioned otherwise, the leakage of a program is the single-run leakage.

As corollary to theorem 1 the order of programs relates to the amount of leakage by the following result

Corollary 1. *Let P_1, P_2 be two programs depending only on the high inputs. Then $\Pi(P_1) \sqsubseteq \Pi(P_2)$ iff for all probability distributions on states in LoI, $H(\Pi(P_1)) \leq H(\Pi(P_2))$.*

The relation between order and leakage is an interesting result because it underlines how fundamental the order relation in the lattice of information is to reason about leakage. While the purely qualitative view of partitions is clearly more coarse than a quantitative view (because it lacks the possibility of assigning different probability weights to elements in an equivalence class) it is still fundamentally restricting the amount of leakage that is possible in a quantitative setting.

This result justifies the measure used in later chapters to bound leakage.

3.4.3 Lattice operations and leakage

The lattice $\sqcup$ join operation can be used to combine leakage of different programs or different runs of the same program, under the assumption of shared universe of values. First, let us give an example program for $\Pi(P)$ and $\Pi_l(P)$. Given the program

```
if (h==0) o=0; else o=1;
```

where the variable h ranges over $\{0, 1, 2, 3\}$. The equivalence relation $\Pi(P)$ associated to the above program is then

$$\Pi(P) = \{\underbrace{\{0\}}_{o=0}\underbrace{\{1, 2, 3\}}_{o=1}\}$$

$\Pi(P)$ effectively partitions the domain of the variable h, where each disjoint subset represents an output. The partition reflects the idea of what a passive attacker can learn of secret inputs by *backwards* analysis of the program, from the outputs to the inputs.

A simple example of $\Pi_l(P)$ with h ranging over $\{0, 1, \ldots, 7\}$ is the following

```
if(l == 5) {
   if(h < 4)
      o = h;
   else
      o = 0;
} else
   o = 0;
```

$\Pi_l(P)$ then represents different program runs each with a different choice of l from the set $L = \{0, 1, \ldots\}$

$$\{\bot, \bot, \bot, \bot, \bot, \{\{0\}\{1\}\{2\}\{3\}\{4, 5, 6, 7\}\}, \bot, \ldots\}$$

Every choice of l leads to the bottom partition except $l = 5$ where the first 4 high values can be distinguished. Clearly, an attacker who can choose any l for $\Pi_l(P)$ is much more powerful than one who has to start from fixed initial values. An attacker which has the ability to maximise the leakage in one run would choose $\Pi_5(P)$, on the other hand, an attacker who can do multiple runs could just "scan" the resulting partitions until he learned enough.

The next definitions show how to combine leakage of different programs and different program runs of the same program using the join operation in $\mathcal{I}(\Sigma)$.

Definition 12. *Given two programs P_1, P_2 which use the same set of variables*

and low initialisations l_1, l_2 *the leakage of both programs combined is*

$$\Pi_{l_1}(P_1) \sqcup \Pi_{l_2}(P_2) \tag{3.12}$$

If P_1 and P_2 are the same program then 3.12 represents the leakage of *two runs* and can be generalised for any number of runs

Definition 13. *Given program* P *and a given choice of inputs* $L = \{l_1, l_2, \dots\}$ *the partition of* $|L|$ *runs is the join of the partitions* $\Pi_l(P)$ *for all* l

$$\bigsqcup_{l \in L} \Pi_l(P) \tag{3.13}$$

For some applications, as will be shown in a later chapter, it is useful to syntactically create a new program which just combines a number of programs. Thus, the syntactic transformation encodes all program paths of all programs in a single program. This is more amendable for program analyses which need to reason about multiple program runs.

Definition 14. *Given programs* P_1, P_2 *a single program* $P_{1 \sqcup 2}$ *exists such that*

$$\Pi(P_{1 \sqcup 2}) = \Pi(P_1) \sqcup \Pi(P_2) \tag{3.14}$$

Given programs P_1, P_2, we define $P_{1 \sqcup 2} = P_1'; P_2'$ where the primed programs P_1', P_2' are P_1, P_2 with all variables renamed to achieve disjoint variable sets. If the two programs are syntactically equivalent, then this results in self-composition [5]. For example, consider the two programs

$$P_1 \equiv \texttt{if (h == 0) x = 0 else x = 1}, \quad P_2 \equiv \texttt{if (h == 1) x = 0 else x = 1}$$

with their partitions $\Pi(P_1) = \{\{0\}\{h \neq 0\}\}$ and $\Pi(P_2) = \{\{1\}\{h \neq 1\}\}$. The program $P_{1 \sqcup 2}$ is the concatentation of the previous programs with variable renaming

$$\begin{aligned} P_{1 \sqcup 2} \equiv\ & \texttt{h}' = \texttt{h}; \texttt{if (h}' == 0\texttt{) x}' = 0 \texttt{ else x}' = 1; \\ & \texttt{h}'' = \texttt{h}; \texttt{if (h}'' == 1\texttt{) x}'' = 0 \texttt{ else x}'' = 1 \end{aligned}$$

The corresponding lattice element is the join, i.e. intersection of blocks, of the individual programs P_1 and P_2

$$\Pi(P_{1\sqcup 2}) = \{\{0\}\{1\}\{h \neq 0, 1\} = \{\{0\}\{h \neq 0\}\} \sqcup \{\{1\}\{h \neq 1\}\}.$$

3.5 Reasoning about loops in programs

To understand the leakage behaviour of a system in more detail the input/output semantics approach might be too coarse. This is especially the case for loops which introduce circular dependencies between program points. Traditionally, this requires externally provided loop invariants which capture the necessary behaviour of the loop and verification property in question.

Up until recently, leakage inference for loops took a safe but very imprecise approach. As soon as there was a confidential variable contained in the guard or body of the loop then everything of the confidential information was considered leaked [13]. A subsequent paper then provided the first precise leakage semantics for loops [38] which separates the leakage analysis in two parts: the leakage of the loop guard and body. This section will summarise that theory and then show an equivalent interpretation in the lattice of information. Chains of elements in the lattice are seen as loop iterations and the leakage is the entropy of the least upper bound of such chains.

3.5.1 Analytical approach

One possible way to analyse loop leakage is by breaking down the leakage in different contributing parts in the program source code. Both the guard and the body of a loop can be separate sources of leaks. It has been shown [38] that those are two of the three components needed to provide a precise quantitative analysis. The three components are:

guard: the information of the number of iterations of the loop

body: the information of the output given the knowledge of the number of iterations

collisions: the information of the number of iterations given knowledge of the output

The idea is that the leakage of a looping program, denoted $L(P)$, is given by the sum of information leaked by the guard and the body minus the ambiguity given by the collisions. In terms of random variables (which will be formally defined later on) this can be expressed as follows [40]:

$$L(P) = \underbrace{H(\texttt{NIterations}(P))}_{guard} + \underbrace{H(P|\texttt{NIterations}(P))}_{body} - \underbrace{H(\texttt{NIterations}(P)|P)}_{collisions}$$

Let us illustrate this formula with an example and its corresponding state transformer in table 3.1

```
l=0;
while(l < h) {
   if (h==2) l=3; else l++;
}
```

and suppose `h,l` are two bit variables with range $\{0, 1, 2, 3\}$ and all values of `h` are equally likely. Then the loop terminates in 0 iterations with probability 0.25 (i.e. only when `h=0`); it terminates in 1 iterations with probability 0.5 (i.e. only when `h=1` and `h=2`), it never terminates after 2 iterations and finally it terminates in 3 iterations with probability 0.25 (i.e. only when `h=3`). This information about the partitioning of the inputs by the number of iterations the loop needs to terminate makes up the first ingredient of $L(P)$:

$$\underbrace{H(\texttt{NIterations}(P))}_{guard} = H(0.25, 0.5, 0.25)$$

For the body leakage, notice that for iterations 0 and 3 no uncertainty is left about the secret (0 bits of information, see table 3.1), and only in the case of 1 iteration the body leaks the information that `h=1` or `h=2` through its two distinguishable outputs (1 bit of information). This amounts to:

$$\underbrace{H(P|\texttt{NIterations}(P))}_{body} = 0.25 * 0 + 0.25 * 0 + 0.5 * 1$$

$$h = 0 \xrightarrow{0} l = 0$$
$$h = 1 \xrightarrow{1} l = 1$$
$$h = 2 \xrightarrow{1} l = 3$$
$$h = 3 \xrightarrow{3} l = 3$$

Table 3.1: State transformer: start $\overset{iteration}{\longrightarrow}$ end

For the collisions, notice that the output `l=3` can be the result of 1 or 3 iterations, both resulting in the same output `l=3`. This setup generates 1 bit of uncertainty about the number of iterations (it could be one or three iterations) with probability 0.5. This gives the last element of the leakage formula:

$$\underbrace{H(\texttt{NIterations}(P)|P)}_{collisions} = 0.5 * 1$$

For this particular program the leakage is then

$$H(0.25, 0.5, 0.25) + 0.25 * 0 + 0.25 * 0 + 0.5 * 1 - 0.5 * 1 =$$
$$H(0.25, 0.5, 0.25) = 1.5$$

That 1.5 is the correct amount leaked can be checked with the following intuition. An attacker observing the output of the program observes `l=0` in which case he knows that `h=0`; he observes `l=1` in which case he knows that `h=1`; he observes `l=3` in which case he knows that `h=2` or `h=3`. These three observations have probability $(0.25, 0.25, 0.5)$ and so the leakage given the observations is $H(0.25, 0.5, 0.25) = 1.5$

We are now going to make this argument formal following [38].

3.5.2 The random variable `NIterations`

Given a looping program $P \equiv$ `while e M` that only depends on a high input variable h let us associate the random variable `NIterations`, or $\texttt{NIt}_\texttt{P}$ in

shorthand, to P.

$\mathtt{NIt_P}$ is the random variable where its values denote "number of iterations the loop terminates in". The associated distribution $p(\mathtt{NIt_P} = n)$ is the sum of the probabilities of all values of h such that for those values P terminates in n iterations.

$$p(\mathtt{NIt_P} = n) = \sum \{p(h = v) | P(v) \text{ terminates in } n \text{ iterations}\}$$

The next proposition shows that the analytical approach of separating guard and body leakage results in the same leakage as in definition 3.11.

Proposition 3.

$$H(\Pi(P)) = H(\mathtt{NIt_P}) + H(\Pi(P)|\mathtt{NIt_P}) - H(\mathtt{NIt_P}|\Pi(P))$$

Proof. We use the information theoretical equality

$$H(Y) = H(X) + H(Y|X) - H(X|Y) \tag{3.15}$$

which is true by definition of the conditional entropy

$$H(X) + H(Y|X) - H(X|Y) = H(X) + H(Y,X) - H(X) - H(X|Y) =$$
$$H(X) + H(Y,X) - H(X) - H(X,Y) + H(Y) = H(Y)$$

The result then follows with replacing $X = \mathtt{NIt_P}, Y = \Pi(P)$. □

The elements on the right hand side of equation 3.15 have the following meaning

1. $H(\mathtt{NIt_P})$ is the leakage of the guard
2. $H(\Pi(P)|\mathtt{NIt_P})$ is the leakage of the body
3. $H(\mathtt{NIt_P}|\Pi(P))$ is the measure of the *collisions* of the loop

A collision is an observable value that can be generated in different iterations of the loop.

The random variable $\texttt{NIt}_\texttt{P}$ can be approximated by the random variable $\texttt{NIt}_{\texttt{P}n}$ which restricts the number of possible values from $0, \ldots, n$, where the last value n has the meaning "the loop terminates in $> n$ iterations". The probabilities associated to $\texttt{NIt}_{\texttt{P}n}$ are also an approximation of the probabilities of $\texttt{NIt}_\texttt{P}$. They are defined by

$$p(\texttt{NIt}_{\texttt{P}n} = m) = \begin{cases} p(\texttt{NIt}_\texttt{P} = m) & \text{if } m \leq n, \\ 1 - \sum\{p(\texttt{NIt}_\texttt{P} = s) | s > n\} & \text{otherwise.} \end{cases}$$

3.5.3 Basic loop leakage definitions

Definition 15 (Leakage of collision free loop)**.** *The leakage of a collision free loop* `while e M` *up to n iterations is given by*

$$W(e, M)_n = H(\texttt{NIt}_{\texttt{P}n}) + H(\Pi(P)|\texttt{NIt}_{\texttt{P}n})$$

Proposition 4. $\forall n \geq 0,\ \ W(e, M)_n \leq W(e, M)_{n+1}$

Proof. The proof can be decomposed by showing that $H(\texttt{NIt}_{\texttt{P}n}) \leq H(\texttt{NIt}_{\texttt{P}n+1})$ which is true because $\texttt{NIt}_{\texttt{P}n+1}$ *refines* the distribution $\texttt{NIt}_{\texttt{P}n}$. To prove the other component of the inequality, i.e. $H(\Pi(P)|\texttt{NIt}_{\texttt{P}n}) \leq H(\Pi(P)|\texttt{NIt}_{\texttt{P}n+1})$ consider the event e' as the "loop terminates in $n+1$ iterations". Using the definition of conditional entropy this simplifies to

$$\begin{aligned} H(\Pi(P)|\texttt{NIt}_{\texttt{P}n}) &= \sum_{\texttt{NIt}_{\texttt{P}n}=e} p(e) H(\Pi(P)|\texttt{NIt}_{\texttt{P}n} = e) \\ &\leq \sum_{\texttt{NIt}_{\texttt{P}n}=e} p(e) H(\Pi(P)|\texttt{NIt}_{\texttt{P}n} = e) + p(e') H(\Pi(P)|e') \\ &= \sum_{\texttt{NIt}_{\texttt{P}n+1}=e} p(e) H(\Pi(P)|\texttt{NIt}_{\texttt{P}n+1} = e) \\ &= H(\Pi(P)|\texttt{NIt}_{\texttt{P}n+1}) \end{aligned}$$

□

Using proposition 3 we can define the leakage of a loop as

$$\lim_{n\to\infty} \mathtt{W}(\mathtt{e},\mathtt{M})_n - H(\mathtt{NIt_P}|\Pi(P)) \tag{3.16}$$

which when there are no collisions simplifies to

$$\lim_{n\to\infty} \mathtt{W}(\mathtt{e},\mathtt{M})_n \tag{3.17}$$

3.5.4 Examples

Let us apply the previous theory to the analysis of two looping programs. The high input variable is a k-bit variable assuming possible values $0,\ldots,2^k-1$ (i.e. no negative numbers).

Unbounded leakage with decreasing rate. Consider the following simple loop with an increasing counter `l`:

```
l=0;
while (l != h) {
     l=l+1;
}
```

No high variables appear in the body of the loop so there is no leakage in the body, i.e

$$\lim_{n\to\infty} H(\Pi(P)|\mathtt{NIt_P}_n) = 0$$

Therefore we only need to study the behaviour of

$$\lim_{n\to\infty} H(\mathtt{NIt_P}_n)$$

The events associated to the random variable $\mathtt{NIt_P}_n$ are:

$$(\mathtt{NIt_P}_n = i) = \begin{cases} 0 = h, \mathtt{if}\ i = 0 \\ \\ 0 \neq h, \ldots, i \neq h \wedge i+1 = h,\ \mathtt{if}\ i > 0 \end{cases}$$

thus every event is equally likely, i.e. $p(\texttt{NIt}_{\texttt{P}n} = i) = \frac{1}{2^k}$. The entropy over all possible guards is then

$$\lim_{n\to\infty} H(\texttt{NIt}_{\texttt{P}n}) = H(\frac{1}{2^k}, \ldots, \frac{1}{2^k}) = \log(2^k) = k$$

As expected all k bits of a variable are leaked in this loop, for all possible k; however, 2^k iterations are required to reveal k bits. We conclude that this is an unbounded covert channel with decreasing rate $\frac{k}{2^k}$.

Bounded leakage with constant rate. The next example is a loop with a decreasing counter and a slightly different guard expression

```
l=20;
while (h < l) {
     l=l-1;
}
```

Again, since the body of the loop does not contain any high variable, the body part of the leakage is 0

$$\lim_{n\to\infty} H(\Pi(P)|\texttt{NIt}_{\texttt{P}n}) = 0$$

Thus we only need to study the leakage of the guard.

After executing the program, `l` will be 20 if `h` ≥ 20 and will be `h` if $0 \leq$ h < 20, i.e. `h` will be revealed if its value is in the interval $0 \ldots 19$.

The events associated to $\texttt{NIt}_{\texttt{P}n}$ are:

$$(\texttt{NIt}_{\texttt{P}n} = i) = \begin{cases} h < 20 - i \wedge h \geq 20 - (i+1) \equiv & \\ \text{h = 20-(i+1),} & i > 0 \\ \\ h \geq 20, & i = 0 \end{cases}$$

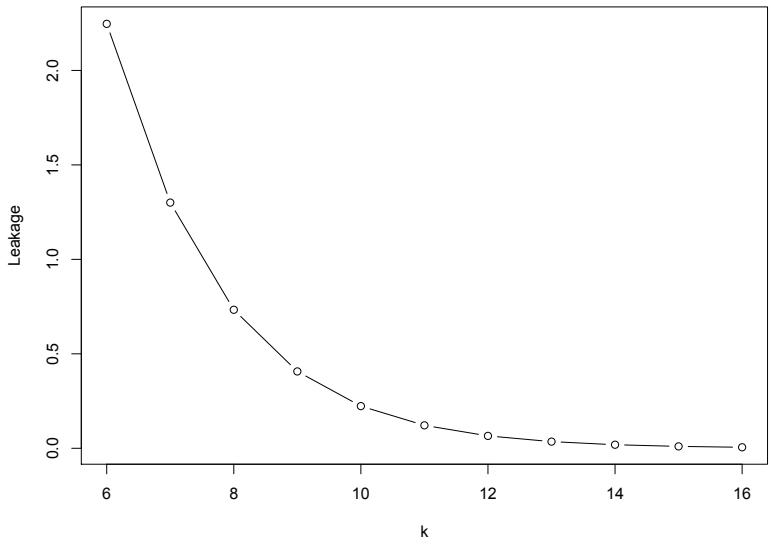

Figure 3.1: Leakage in `l=20; while (h < l) {l=l-1}`

and

$$p(\mathtt{NIt}_{\mathtt{P}n} = i) = \begin{cases} \frac{2^k-20}{2^k} \text{ if } i = 0 \\ \frac{1}{2^k} \text{ if } 0 < i \leq 20 \\ 0 \text{ if } i > 20 \end{cases}$$

The leakage is then given by

$$\lim_{n\to\infty} H(\mathtt{NIt}_{\mathtt{P}n}) =$$
$$H(\frac{2^k-20}{2^k}, \frac{1}{2^k}, \ldots, \frac{1}{2^k}, 0, \ldots, 0) =$$
$$-\frac{2^k-20}{2^k}\log(\frac{2^k-20}{2^k}) - 20(\frac{1}{2^k}\log(\frac{1}{2^k}))$$

This function is plotted in Figure 3.1 for $k = \{6 \ldots 16\}$. The interesting element in the graph is how it shows that for k around 6 bits the program is unsafe (more than 2.2 bits of leakage) whereas for k from 14 upwards the program is safe (around 0 bits of leakage).

However, the uniform distribution is not the channel distribution. The capacity of this channel is 4.3923 and is achieved by the distribution where the only values with non zero probability for `h` are in the range $\{0 \dots 20\}$ and have uniform distribution[3].

3.6 Loops in the lattice of information

This section presents a natural interpretation of the previous analysis of loops in the lattice of information. The key result is that leakage of loops is the semivaluation of the least upper bound of a chain of elements in the lattice of information, where the chain is the interpretation of the different iterations of the loop.

To understand the ideas let us consider again the program

```
l=0;
while(l < h) {
   if (h==2) l=3; else l++;
}
```

and let us now study the partitions it generates. The loop terminating in 0 iterations will reveal that `h=0` i.e. the partition $W_0 = \{\{0\}\{1,2,3\}\}$; termination in 1 iteration will reveal `h=1` if the output is 1 and `h=2` if the output is 3 i.e. $W_1 = \{\{1\}\{2\}\{0,3\}\}$; the loop will never terminate in 2 iterations i.e. $W_2 = \{\{0,1,2,3\}\}$; in 3 iterations it will reveal that `h=3` given the output 3, i.e. $W_3 = \{\{3\}\{0,1,2\}\}$. Let us define $W_{\leq n}$ as $\sqcup_{n \geq i \geq 0} W_i$, we have then

$$W_{\leq 1} = W_{\leq 2} = W_{\leq 3} = \{\{0\}\{1\}\{2\}\{3\}\}$$

We also introduce an additional partition C to cater for the collisions in the loop: the collision partition is $C = \{\{0\}\{1\}\{2,3\}\}$ because the inputs `h=2` and `h=3` generate the same output in different number of iterations. Given these partitions, the loop leakage is then

[3]We are ignoring the case where $k < 5$ where the capacity is less than 4.3923

$$H(\sqcup_{n\geq 0} W_{\leq n} \sqcap C) = H(\{\{0\}\{1\}\{2,3\}\})$$

Notice now that the analytic and lattice interpretation give the same result: assuming uniform distribution we get

$$\underbrace{H(0.25, 0.5, 0.25)}_{guard} + \underbrace{0.5\ H(0.5, 0.5)}_{body} - \underbrace{0.5\ H(0.5, 0.5)}_{collisions} = 1.5$$
$$= H(\{\{0\}\{1\}\{2,3\}\})$$

We can interpret looping programs in the lattice of information as least upper bounds of increasing sequences; for some loops (those with collisions) this is not immediately true: however we will show that all loops can be interpreted as the meet of the least upper bound of an increasing sequence and a point in the lattice representing the collisions.

3.6.1 Algebraic interpretation

Given a loop W, let W_n be the program W up to the nth iteration. The random variable associated to W_n is a partition where only the outputs of W up to the nth iteration are distinguished. Thus, W_{n+1} will refine W_n by introducing additional blocks.

As a simple example of a collision free program consider the "linear search" program P below

```
l=0;
while (l < h) {
   l=l+1;
}
```

We get the following corresponding family of partitions of states P_n:

$$P_n = \{\{0\}, \{1\} \ldots, \{n-1\}, \{x|\ x \geq n\}\}$$

The following proposition establishes the relation between collision free loops and the chain W_n being increasing:

Proposition 5. *For all n, $W_n \sqsubseteq W_{n+1}$ iff the loop W is collision-free.*

Proof. The direction $\Rightarrow$ follows immediately from the definition of collision. For the $\Leftarrow$ suppose $W_n \not\sqsubseteq W_{n+1}$, then at least a block in W_{n+1} is not a refinement of a block in W_n, e.g. $\{\{a\},\{b,c\}\}$ in W_n and $\{\{a,b,c\}\}$ in W_{n+1} and by definition of W_n either $\{\{a\}$ or $\{b,c\}\}$ (w.l.g. we can say is $\{a\}$) corresponds to an output o after $\leq n$ iterations. Then $\{\{a,b,c\}\}$ in W_{n+1} corresponds to a collision, namely the collision which sends a, b, c to the same output o in a different number of iterations. □

Proposition 6. *The random variable W of a collision-free loop is the Kleene fixpoint $\sqcup_{n\geq 0} W_n$ of the chain $(W_n)_{n\geq 0}$.*

Proof. The result follows from Proposition 5 and the fact that the number of states is finite. □

Theorem 2. *Given a collision-free loop* `while e M`*, the leakage* $\lim_{n\to\infty} W(e,M)_n$ *is equal to the semivaluation $H(\sqcup_{n\geq 0} W_n)$.*

Proof. This follows from Proposition 6. □

Also note that in this case the chain $W_0 \sqsubseteq W_1 \sqsubseteq \ldots$ satisfies the ascending chain condition. There exists an integer n such that $W_m = W_n$ for all $m > n$, because W_{i+1} destructively refines or "splits" a finite block of W_i into smaller equivalence classes.

3.6.2 Loops with collisions

Let us look at the colliding program shown in Figure 3.2. It consists of two iterations, represented by functions f_1 and f_2.

The exact partition for this program is

$$P = \{\{a, a'\}, \{x, x', y\}, \{c\}\}$$

The chain of partitions associated to the program is the following:

$$W_1 = \{\{a, a'\}, \{x, x'\}, \{y, c\}\}$$
$$W_2 = \{\{a, a'\}, \{x, x', y\}, \{c\}\}$$

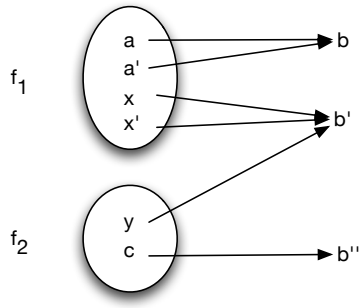

Figure 3.2: Two iterations with one collision at b'

We see that W_2 extends the block containing x, x' with y because all three of them have the same image b'. This reflects the idea of collisions, namely that two (or more) elements of the codomain of two different iteration functions, here f_1 and f_2, coincide. The result is that their inverse images are indistinguishable from one another and therefore end up being in the same block, here $\{x, x', y\}$. Then, W_2 is equal to P. However, because W_2 extends a block in W_1 this is not an ascending chain anymore; actually by choosing a distribution assigning probability 0 to c, we can see that $H(W_1) > H(W_2)$ and therefore theorem 2 is false in case of collisions.

To address this problem we first introduce a trick to transform a sequence of partitions into an ascending chain of partitions: given a sequence of partitions $(W_i)_{i\geq 0}$ define the sequence $(W_{\leq i})_{i\geq 0}$ by

$$W_{\leq i} = \sqcup_{j\leq i} W_j$$

It is easy to see that $(W_{\leq i})_{i\geq 0}$ is an increasing chain.

Define now the *collision equivalence* of a loop W as the reflexive and transitive closure of the relation $\sigma \simeq_C \sigma'$ iff σ, σ' generate the same output from different iterations.

We are now ready to relate the leakage of arbitrary loops with semivaluations on LoI.

Theorem 3. *The leakage of an arbitrary loop as in definition 3.16 is equivalent*

to semivaluating the meet of the least upper bound of its increasing chain $W_{\leq n}$ and its collision partition C, i.e.

$$\lim_{n\to\infty} \mathtt{W}(\mathtt{e},\mathtt{M})_n - H(\mathtt{NIt_P}|\Pi(P)) = H(\sqcup_{n\geq 0} W_{\leq n} \sqcap C)$$

Proof. Notice first that increasing chains x_n with a maximal element in a lattice do distribute, i.e.:

$$(\sqcup_{n\geq 0} x_n) \sqcap y = \sqcup_{n\geq 0}(x_n \sqcap y)$$

Assuming distributivity the argument is then easy to show:

$$(\sqcup_{n\geq 0} W_{\leq n} \sqcap C) = \sqcup_{n\geq 0} (W_{\leq n} \sqcap C)$$

Notice now that $(W_{\leq n} \sqcap C)_{n\geq 0}$ is a chain cofinal to the sequence $(W_n)_{n\geq 0}$ and so we can conclude that
$\sqcup_{n\geq 0}(W_{\leq n} \sqcap C)$ is the partition whose semivaluation corresponds to $W(e, M)$.
□

Notice the generality of the lattice approach: we can replace Shannon entropy H with any real valued map from the lattice of information F and we get a definition of leakage for loops as follows:

$$F(\sqcup_{n\geq 0}(W_n \sqcap C)).$$

Chapter 4

Exact Leakage Quantification by Complete Enumeration

We start the first chapter of automated techniques to calculate leakage by the most straightforward brute-force technique: complete enumeration of all possible confidential inputs to a program and evaluating its leakage, usually assuming uniform input distribution. This approach is an exact method as it calculates the correct entropy quantity by running the program on all inputs. This simple analysis is included for three reasons

- to motivate the use of smarter approximate methods presented in later chapters
- it has proven to be a useful tool during research and development of more advanced tools and was the starting point of my research
- a *safe* upper bound on the exact leakage has been formalised in cases where not all inputs have been treated

The analysis has been implemented in a tool and has been tested on 82 different programs over the years. Two of those programs and their analysis will be covered in this chapter as case studies.

Input: Program text P,Value range H
Output: $\Pi(P)$, $\texttt{NIt}_\texttt{P}$
$\Pi(P) = \emptyset$
$\texttt{NIt}_\texttt{P} = \emptyset$
for $h \in H$ **do**
  $o, n \leftarrow P(h)$
  $\Pi(P) \leftarrow \texttt{attach}(o,\ \Pi(P),\ \{h\})$
  $\texttt{NIt}_\texttt{P} \leftarrow \texttt{attach}(n,\ \texttt{NIt}_\texttt{P},\ \{h\})$
end

Algorithm 1: Iteratively calculating partitions $\Pi(P)$ and $\texttt{NIt}_\texttt{P}$

4.1 Analysis

4.1.1 Objective

The objective of this analysis is to calculate the exact loop leakage for a simple while language (as presented in section 2.1) with a single loop in the program text. The leakage calculation should implement the formulas from section 3.5.1. Please refer to that section for the notation used.

4.1.2 Algorithm

For the complete enumeration we iteratively compute two partitions representing $\Pi(P)$ and $\texttt{NIt}_\texttt{P}$ according to the algorithm 1. Input to the algorithm are the program P and the range of secret values H. The program is run with only the secret value h as input; its return value is the computed output observation o and the last iteration count of the loop n.

The algorithm uses the helper function `attach`(*label*, *partition*, *block*) which adds block *block* to the equivalence class with label *label* from partition *partition*. It always returns the updated partition. The semantics of `attach` are self-explanatory, e.g.

$$\texttt{attach}(i, \emptyset, \{1,2\}) = \{\{1,2\}_i\}$$
$$\texttt{attach}(i, \{\{0\}_j\{1,2\}_i\}, \{3,4\}) = \{\{0\}_j\{1,2,3,4\}_i\}$$

Following from that, the program and loop leakage of the resulting partitions can be calculated easily. According to proposition 3, the program leakage is $H(\Pi(P))$ and the loop leakage is calculated by the conditioning: leakage of the guard is $H(\texttt{NIt}_\texttt{P})$, leakage of the body $H(\Pi(P)|\texttt{NIt}_\texttt{P})$ and collision leakage is $H(\texttt{NIt}_\texttt{P}|\Pi(P))$.

The conditional entropy is calculated using the conditioning of partitions from equations 3.7 and 3.8 and by its definition

$$H(\Pi(P)|\texttt{NIt}_\texttt{P}) = \sum_N p(\texttt{NIt}_\texttt{P} = n) H(\Pi(P)|\texttt{NIt}_\texttt{P} = n).$$

Here, we assume that $\texttt{NIt}_\texttt{P} = n$ is a shorthand for selecting the block with label n from $\texttt{NIt}_\texttt{P}$. Thus, every block is selected and the entropy of the conditional is calculated according to equation 3.8. As we consider uniform distribution the probability reduces to block counting: $p(\texttt{NIt}_\texttt{P} = n) = \frac{|\texttt{NIt}_\texttt{P}=n|}{\sum_{i \in N} |\texttt{NIt}_\texttt{P}=i|}$ (see section 2.5).

4.1.3 Example

Let us take the following example with range of $\texttt{h} \in H = \{0, 1, 2, 3\}$.

```
l=0;
while(l<h) {
  if(h==2 || h==3)
    l=3;
  else
    l++;
}
```

The algorithm 1 finds the following two partitions

$$\Pi(P) = \{\{0\}_0\{2,3\}_3\{1\}_1\}$$
$$\texttt{NIt}_\texttt{P} = \{\{0\}_0\{1,2,3\}_1\}$$

To calculate the leakage of the guard we have to only consider the block $\texttt{NIt}_\texttt{P} = 1 \equiv \{1, 2, 3\}$, thus $p(\texttt{NIt}_\texttt{P} = 1) = \frac{3}{4} = 0.75$. Plugging in equation 3.8 results in

the following partition for the conditioning

$$(\Pi(P)|\mathtt{NIt_P} = 1) = \{\{1\}\{2,3\}\}$$

which results in the entropy

$$H(\Pi(P)|\mathtt{NIt_P} = 1) = 0.75H(\frac{1}{3}, \frac{2}{3})$$

Notice that there is no collision leakage because $\mathtt{NIt_P} \sqsubseteq \Pi(P)$. If there was leakage it would work exactly in the same way as the leakage in the body, except that we would also need identifying labels describing which output value belongs to which equivalence class of $\Pi(P)$ for the selection and intersection process. See section 3.5.1 for more information.

Thus the leakage of this loop amounts to

$$\begin{aligned} H(\Pi(P)) &= H(\frac{1}{4}, \frac{2}{4}, \frac{1}{4}) = 1.5 \\ H(\Pi(P)) &= H(\mathtt{NIt_P}) + H(\Pi(P)|\mathtt{NIt_P}) = H(\frac{1}{4}, \frac{3}{4}) + 0.75H(\frac{1}{3}, \frac{2}{3}) \\ &= 1.5 \end{aligned}$$

4.2 Safe upper bound

An information theoretical upper bound can be provided for $\Pi(P)$ if the analysis is *not* completely run on the set of all inputs; this is most often the case when the algorithm is aborted before normal termination. This upper bound is safe, which means that it is always greater than or equal to the true, precise leakage. Let us assume that all variables have k bits thus the space of the confidential variable H is 2^k.

The reasons for not completing the enumeration are obvious

- $P(h)$ could be an expensive function to evaluate
- 2^k could be very large
- some h could cause $P(h)$ to not terminate

Let us assume that P is only run on the first m inputs, thus $m < 2^k$ then the entropy of $\Pi(P)$ is

$$H(\Pi(P)_m) = H(p_1, \ldots, p_u, q)$$

with $u \leq m$ and $q = 1 - \sum_{i=1}^{u} p_i$. Remember that using the partition-view as intuition, p_i is the sum of the probability of all inputs leading to the output representing the ith equivalence class. There are $2^k - m$ inputs untreated sharing a probability of q. The worst-case assumption is that every remaining input will result in a new, distinguishable observation with probability $\frac{q}{2^k - m}$, i.e. distributing the remaining probability q over the untreated inputs.

Proposition 7. *Let k be the size of the secret, and the entropy calculated so far for m inputs is*

$$H(\Pi(P)_m) = H(p_1, \ldots, p_u, q) \tag{4.1}$$

with $u \leq m$ and $q = 1 - \sum_{i=1}^{u} p_i$. A safe leakage upper bound is then calculated as follows

$$H(\hat{\Pi}(P)_m) = H(p_1, \ldots, p_u, \frac{q}{2^k - m}, \ldots, \frac{q}{2^k - m}) \tag{4.2}$$

with $\hat{\Pi}(P)_m$ being the partition which distributes the remaining probability q over the untreated inputs.

Proof. The choice of uniform fractions $\frac{q}{2^k - m}$ is motivated by equation 2.2 which is shown to maximise entropy. The probabilities p_i are calculated using the maximum likelihood estimator $\frac{n_i}{2^k}$ with n_i being the cardinality of the ith equivalence class. For proving the upper bound we also assume that every p_i is "complete", i.e. no other inputs lead to the same output. This is generally not the case but does not influence the upper bound.

There are two extreme cases:

I. no untreated input adds a new observation. The leakage is then bound by $\log_2(u)$ which is clearly less or equal to the upper bound.

II. assume $m = u$, i.e. when every input so far generated a unique observation, and also assuming that the remaining inputs show the same behaviour, then the precise leakage and our upper bound in 4.2 coincide.

□

As an illustrative example, let us take the parameters $2^k = 8, m = 4$ and the partition

$$\Pi(P)_m \equiv \{\{1,2\}\{3,4\}\underbrace{\{5,6,7,8\}}_{\text{“}q\text{”}}\}$$

Thus $q = \frac{4}{8} = 0.5, 2^k - m = 8 - 4 = 4$ and each new singleton class is assigned the probability $\frac{0.5}{4} = \frac{1}{8}$. The upper bound partition is then

$$\hat{\Pi}(P)_m \equiv \{\{1,2\}\{3,4\}\{5\}\{6\}\{7\}\{8\}\}$$

and its entropy is

$$H(\hat{\Pi}(P)_m) = H(\frac{1}{4}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}) = 2.5$$

out of the possible $\log(8) = 3$ bit.

4.3 Case studies

The following two case studies show the application of the technique developed in this chapter. The examples also serve as a way to compare human intuition of leakage through code review with automatically computed leakage. We learn that manual code review is not appropriate to calculate or estimate leakage which is an argument for tool support.

4.3.1 Square root

The integer square root is defined as follows

$$sqrt(x) = \lfloor \sqrt{x} \rfloor$$

There are many different ways of implementing this definition in a while language. One of the simplest is

```
l=0;
```

```
while(l*l <= h) {
     l++;
}
l--;
```

which calculates `l=sqrt(h)`, therefore `l` contains the largest integer such that `l*l <= h`. Our interest is to know how many bits of the secret h leak into l. This example is straightforward, half the bits of h minus the remainder removed by the floor operation are revealed by l after the execution of the program. In this case, calculating the leakage is feasible and a code reviewer can accurately reason about the revealed quantity.

But what about other implementations; do they leak the same amount and can we rely on our intuition to deduce the leakage? The next program is an equivalent implementation, computing `res=sqrt(num)`.

```
curLog=16; res=0; flag=1;
nextNum=0; resAdd=0;
while(curLog > 0 && flag == 1) {
   nextNum = num - (res << curLog);
   curLog = curLog-1;
   if(nextNum > 0) {
      resAdd = (1 << curLog);
      nextNum = nextNum - (resAdd << curLog);
      if(nextNum > 0) {
         num = nextNum;
         res = res | resAdd;
      } else {
         if(nextNum == 0) {
            res = res | resAdd;
            flag=0;
}}}}
```

This version is so complex that, for most programmers it is hard to even guess what this program is doing. Calculating the leakage from `num` to `res` without tool-support is very difficult and error-prone. However, our tool re-

vealed that this implementation leaks *all* of the input bits to the output[1]; further, it showed that the variables `nextNum` and `flag` are acting as covert channels and are responsible for the high leakage. This is reflected in the following analysis output, with `num` being a 5 bit variable:

```
Guard leak: 0.3998 Body leak: 4.6002
Collision: 0.0000 Loop leaks: 5.0000
```

The 5 bit leakage means that we can uniquely identify the input by observing the output values alone. However, when we remove `nextNum` and `flag` from the calculation we get:

```
Guard leak: 0.3998 Body leak: 2.2234
Collision: 0.0000 Loop leaks: 2.6232
```

Reducing the overall leakage from 5 to 2.62 bits – almost half the original leakage.

To summarise, different implementations of the same algorithm can have very different leakage behaviour and we can not always rely on intuition. Further, we showed that automated quantitative analysis could help programmers identify leaking components in algorithms and could provide ways to minimise such information leakages by eliminating side-channels.

4.3.2 Prime numbers

Consider the following program:

```
l=2;
while (h % l > 0) {
  l++;
}
```

which computes the smallest divisor of a secret h. Again, we are interested in the information leaked from the secret `h` to the public variable `l`. In that respect, this program is clearly unsafe: whenever h is prime the whole secret will be disclosed. But how much do non-prime inputs contribute to the leakage?

[1] All observable variables together build the output

One possible way to approach this question is by using the Prime Number Theorem which states that

> *The proportion of primes less than a k bit number h is asymptotic to $1/\ln h$.*

By this theorem one could derive the following estimate for the leakage: For all k the above program leaks 1.94 bits.

The idea behind this estimation is the following: the probability of a prime within 2^k possible values can be approximated by $\frac{1}{\ln(2^k)}$ and in this case it leaks everything as stated above; so far this gives a leakage of $\frac{k}{\ln(2^k)} = 1.442$. The remaining 0.5 to reach the total of 1.94 bits is obtained by observing that half of the numbers are divisible by 2. The knowledge that a number is divisible by 2 consists of 1 bit: if the least bit of a number is 0 then it is divisible by 2. Multiplying this single bit by the probability of being in that set yields the required 0.5 bit.

However, this argument has two flaws:

1. It does not take into account the non-primes which are divisible by any number other than 2. Incorporating the leakage generated by these numbers may require more substantial mathematics

2. It does not takes into account that the program does not terminate if the secret is 1

Using the algorithm 1 we can compute the real leakage. Below is an output of the system computing the leakage of the program for a 3 bit secret:

```
[h -> 1] caused timeout
<l = 2 h = 4 > --0--> <l = 2 h = 4 >
<l = 2 h = 6 > --0--> <l = 2 h = 6 >
<l = 2 h = 2 > --0--> <l = 2 h = 2 >
<l = 2 h = 0 > --0--> <l = 2 h = 0 >
<l = 2 h = 3 > --1--> <l = 3 h = 3 >
<l = 2 h = 5 > --3--> <l = 5 h = 5 >
<l = 2 h = 7 > --5--> <l = 7 h = 7 >
Analysis:
```

```
Guard leak: 1.6645 Body leak: 0.0000
Collision: 0.0000 Loop leaks: 1.6645
Upper Bound Analysis
Remaining (1-p) = 0.1250
Safe upperbound: 2.0000
```

The output starts by listing input/output variable configurations of the program, e.g. the second line means the program on inputs $l = 2, h = 4$ terminated after 0 iterations with the values $l = 2, h = 4$, i.e. no computation took place.

The output of the analysis tool explains the actual leakage of the program. For a 3 bit secret a leakage of 1.6645 is calculated: in the three cases when h is prime ($\frac{3}{7}$ of the cases) the observer will learn the whole secret ($\log(7)$ bits of information), whereas in the other 4 cases one will know that the secret is one of the 4 possible non prime numbers ($\log(7) - \log(4)$ bits of information). The leakage from the observable output is hence

$$\frac{3}{7}\log(7) + \frac{4}{7}\log(\frac{7}{4}) = 1.6645$$

The first line of the output (`[h -> 1] caused timeout`) indicates that when the variable h was initialised with the value 1 then the program did not terminate in the allowed time. The *Upper Bound Analysis* is using the upper bounds of Proposition 7 exactly in such cases when one or more inputs cause (observable) non-termination:

- The lower bound 1.6645 is the leakage where non-terminating inputs are ignored, i.e. the non-terminating inputs contribute 0 to the leakage.
- The upper bound 2 is obtained by considering the case where the non-terminating input contributes maximally to the leakage as proved in

Secret size (bits)	Leakage (terminating)	Upper Bound	Time
3	1.6645	2	0.010s
4	1.989	2.2028	0.014s
8	3.0890	3.1138	0.118s
10	3.3897	3.3976	0.960s
16	3.9298	3.9302	7m7s

Table 4.1: Leakage for the Primes program

Proposition 7:

$$\begin{aligned} q = \frac{1}{8},\ 2^k = 8,\ m = 7,\ \frac{q}{2^k - m} &= \frac{1}{8} \\ \Pi(P) &= \{\{0,2,4,6\}\{3\}\{5\}\{7\}\{\mathbf{1}\}\} \\ &= H(\frac{4}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{\mathbf{1}}{\mathbf{8}}) = 2 \end{aligned}$$

Table 4.1 shows the leakage, refined upper bounds, and runtime of the analysis for different sizes of the secret. The runtime column demonstrates well how this analysis does not scale. This is only a one dimensional secret space (one secret variable) of maximally 16 bit, and a very short program. Clearly, the runtime is prohibitively long.

To summarise, in respect to this example it has become clear that tool support is needed to calculate leakage. Even very short programs, like the one presented, can expose difficult leakage behaviours which are too error-prone to be calculated by humans. This example also demonstrated that the new bounds calculate meaningful results even in the presence of non-termination inputs.

4.4 Review of technique

This chapter described a *dynamic* analysis to evaluate the leakage of a program written in a while language. However, the analysis is not limited to while languages but can be performed on any code which is *executable*. Also it calculates the *precise* leakage, including the leakage breakdown in case of loops.

Unfortunately, the runtime of the algorithm depends on the execution time of the program multiplied by the size of the secret, which is very expensive in general. Ideally, the runtime would be independent of either one of the two variables.

Apart from the precise leakage calculation, the features described above present drawbacks. The fact that the code has to be executable restricts the programs which can be analysed. Either a whole program has to be analysed, which is executable but consists of a large number of lines of code, or an executable slice of a program has to be generated which is a whole research area in itself. Thus, the analysis of, for example, a module or individual function out of a more complex program, such as a kernel module, is not possible.

Another subtle difficulty introduced by the enumeration of the secret input is that the input has to be easily enumerable. While this is trivial for simple datatypes like integers, if more complex datastructures, which by definition follow a certain structure, were used as secret input then this enumeration is not straightforward anymore.

Finally, the dynamic analysis brings with it the obvious disadvantages when it comes to security applications: if the code to be analysed is critical to a system and one would want to know beforehand if there is leakage or not, then this analysis is not suitable either.

The next chapters try to address these shortcomings by developing more advanced analyses.

Chapter 5

AQUA Tool

This chapter describes a different tool developed for this thesis which addresses some of the weaknesses of the previous approach to automatically calculating leakage. Most importantly, this analysis has three objectives:

- Employ static analysis techniques instead of dynamic execution
- Address scalability issues
- Move away from while-languages to ANSI-C

To achieve the objectives the new technique exploits the original definition of noninterference and applies symbolic verification methods instead of the explicit execution of the program.

The main idea for this work stems from multiple recent papers [59, 4, 62] which describe how the noninterference property can be checked using conventional program verification techniques such as verification of safety properties. The most important source is the work from Backes, Köpf, and Rybalchenko [4].

Let us start with some observations about noninterference. A program P is noninterfering if it satisfies the well-known equation from Joshi and Leino

$$HH; P; HH = P; HH$$

5.0

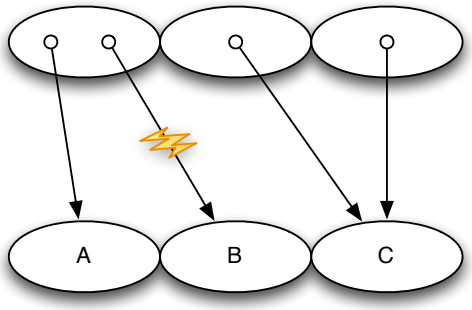

Figure 5.1: Distinction in class B as Non-Interference violation

which can be interpreted as: running a program does not create more distinctions on the secret values.

It is known that if a program is non-interfering then it has no leakage or equivalently $\Pi(P) = \bot$ [14]. Thus, it should be possible to use the violation of this property as a way to quantify leakage.

A violation of noninterference is shown in the figure 5.1. Each oval describes an equivalence class and the four dots inside the top figure are the elements in the confidential space. Let us take the top[1] figure as an initial partition of the secret and the bottom figure as $\Pi(P)$. A violation of noninterference is the arrow to the B equivalence class which creates a distinction between two related secret values, i.e. A and B now distinguish the two input values in the first equivalence class of the initial partition.

Every violation of NI is potentially an additional distinction between two secret values. However, unless the program leaks everything, this does not need to be the general case, because two secret values might be distinguished but happen to fall in an existing equivalence class.

Definition 16 (NI violation)**.** *A noninterference violation (or NI violation) is a counterexample to the NI property.*

However, to completely find the partition $\Pi(P)$ using NI violations alone a large number of counterexamples would need to be found. For k bit variables,

[1]Not to confuse with $\top$ partition

$2^k(2^k-1)/2$ pairs of inputs would need to be evaluated, much larger than the actual secret space. Therefore, the approach has to be modified to be practical.

The approach in this chapter is based on the assumption that there *is* leakage. Using a combination of techniques, we compute two characteristics of the partition $\Pi(P)$ separately:

- Number of distinct outputs in the program with respect to the confidential variables, i.e. number of equivalence classes
- Sizes of inverse images of every distinct output, i.e. equivalence class sizes

For example, let us take the one-liner program $P(h) = h \ \% \ 4$ where h is 4 bit. Then $\Pi(P)$ is

$$\{\{4, 8, 12, 16\}\{1, 5, 9, 13\}\{2, 6, 10, 14\}\{3, 7, 11, 15\}\}$$

which results in 4 distinct outputs (since the program is calculating modulo 4) and all equivalence classes have size 4 as well. The tool presented in this chapter, called AQuA (Automated Quantitative Analysis), calculates these two quantities automatically from the source code of the program; the resulting partition is then ready to be quantified.

At the end of the chapter, a benchmark shows the performance of the tool on examples from the literature and other relevant pieces of code. The application section describes how AQuA could be applied to the area of statistical databases to reason about leakage of combined queries.

This work has been published at the proceedings of the FAST (Formal Aspect of Security and Trust) workshop [29] in LNCS.

5.1 Automation by applying self-composition

A property is a set of execution traces where checking if a trace is a member of the property does not depend on any other traces in the property. A safety property declares that something bad can never happen [34] and it can be expressed as an invariance argument, or a predicate on states.

Definition 17 (Safety Property)**.** *A safety property is a set of traces of a program P where there exists some predicate $\phi(\cdot)$ which every execution trace satisfies.*

$$S \subseteq [\![P]\!].\ \forall s \in S.\ \phi(s)$$

A safety problem is then the decision on the membership to a safety property.

Noninterference is not a safety property [43] however Terauchi and Aiken [59] showed that it is *almost* a safety property and defined it as *2-safety property*. Such a property can be refuted by observing two finite traces. The same authors also showed that Noninterference can be checked using self-composition (by Barthe et. al. [5]) which reduces a 2-safety problem into a safety problem.

Self-composition is a simple program transformation which enables the program P to be evaluated only once by sequencing a copy of P, named P', where all variables have been replaced by fresh copies which do not appear in P simply as follows

$$P; P'$$

By executing this self-composed program two "runs" of the program P are performed in one piece of code.

Using this technique, we can check 2-safety with the (un-)reachability of some label. This is demonstrated as follows in pseudo code following Backes, Köpf, and Rybalchenko [4]

```
if(l == l' && h ≃ h')
   P(h,l); P'(h',l')
   if(l != l') ERROR
```

The assumption that an attacker can not learn anything about the secret (i.e. for all `h,h'. h ≃ h'`) can be falsified by finding an execution path which reaches the `ERROR` label. The reachability of this label can be efficiently checked by off-the-shelf model checkers.

Thus, this approach is efficient for finding violations of the noninterference property, or in other words of the initial assumption that all secret values are indistinguishable.

5.1.1 K-safety and quantitative information flow

The concept of 2-safety can be generalised to *k-safety* where k is the number of traces needed to refute the property [16, 65]. Again, this property can be reduced to a normal safety property and checked accordingly .

However, a negative result by Yasuoka and Terauchi [65] showed that inferring, i.e. computing, quantitative information flow is not k-safety for any k. Thus one can not simply use self-composition like in the noninterference case to precisely calculate leakage.

To overcome this problem, we reformulate the reachability problem in a way which assumes that there is leakage in P, i.e. that the secret input is not the bottom partition. Starting from this assumption, the code from above is modified as follows

```
assume(l = l' && h = i)
P(h,l); P'(h',l')
assert(l != l')
```

The initial assumption on the high equivalence is dropped and instead the original high variable `h` is initialised with value `i`. Also, the primed variable `h'` is left *uninitialised*. As a next step, this program is fed to a SAT solver, which when satisfiable found a model for this program where `h'` is assigned a value which is in a different equivalence class than `h = i`.

Iterating and extending this algorithm in the right way will necessarily find the equivalence relation from equation 3.9.

5.2 Inferring QIF by SAT solving and model counting

The previous section showed the core idea behind the analysis which takes the "two program runs view" from Joshi and Leino's interpretation of noninterference and the translation to a constrained program which finds distinguishable equivalence classes in the secret space. This section describes two algorithms which perform the partition discovery.

The two step process is best explained using the recurring password example with 4 bit variable width and the secret input variable `pwd`:

```
if(pwd == 4) { return 1; } else { return 0; }
```

The first step of the method is to find a *representative* input for each possible output. In our case, AQuA could find the set $\{4, 5\}$, for outputs 1 and 0, respectively. This is accomplished using a SAT-based fixed point computation.

The next step runs on that set of representative inputs. For each input in that set, the number of possible inputs are counted which lead to the same implicit, distinct output. This step is accomplished using model counting.

5.2.1 Core algorithms

The method consists of two reachability analyses, which can be run either one after another or interleaved.

The first analysis finds a set of inputs to which the original program produces distinct outputs for. That set has cardinality of the number of possible outputs for the program. The second analysis counts the set of all inputs which lead to the *same* output. This analysis is run on all members of the set of the first analysis. Together, these two analyses discover the partition of the input space according to the outputs of a program.

To a program P we associate two modified programs $P_{\neq}$ and $P_{=}$, representing the two reachability questions. The two programs are defined as follows

$$P_{\neq}(i) \equiv h = i; P; P'; \texttt{assert}(\texttt{l}! = \texttt{l}')$$
$$P_{=}(i) \equiv h = i; P; P'; \texttt{assert}(\texttt{l} = \texttt{l}')$$

The program P is self-composed [5, 59] and is either asserting low-equality or low-inequality on the output variable and its copy. Their argument is the initialisation value for the input variable. This method works on any number of input variables, but we simplify it to a single variable to ease readability.

The programs $P_{\neq}$ and $P_{=}$ are unwound into propositional formula and then translated in Conjunctive Normal Form (CNF) in a standard fashion.

Input: $P_{\neq}$
Output: S_{input}
$S_{input} \leftarrow \emptyset$
$h \leftarrow random$
$S_{input} \leftarrow S_{input} \cup \{h\}$
while $P_{\neq}(h)$ *not unsat* **do**
  $(l, h') \leftarrow$ Run SAT solver on $P_{\neq}(h)$
  $S_{input} \leftarrow S_{input} \cup \{h'\}$
  $h \leftarrow h'$
  $P_{\neq} \leftarrow P_{\neq} \wedge l' \neq l$
end

Algorithm 2: Calculation of S_{input} using $P_{\neq}$

Input: $P_{=}$, S_{input}
Output: M
$M = \emptyset$
while $S_{input} \neq \emptyset$ **do**
  $h \leftarrow s \in S_{input}$
  $\#models \leftarrow$ Run $allSAT$ solver on $P_{=}(h)$
  $M = M :: \{\#models\}$
  $S_{input} \leftarrow S_{input} \setminus \{s\}$
end

Algorithm 3: Model counting of equivalence classes in S_{input}

$P_{\neq}$ is solved using a number of SAT solver calls using a standard reachability algorithm (SAT-based fixed point calculation).

Algorithm 2 describes this input discovery. In each iteration it discovers a new input h' which does not lead to the same output as previous the input h. The new input h' is added to the set S_{input}. The observable output l is added to the formula as blocking clause, to avoid finding the same solution again in a different iteration. This process is repeated until $P_{\neq}$ is unsatisfiable which signifies that the search for S_{input} elements is exhausted.

Given S_{input} (or a subset of it) as result of Algorithm 2, we can use $P_{=}$ to count the sizes of the equivalence classes represented by S_{input} using model counting. This process is displayed in Algorithm 3 and is straightforward to understand.

The algorithm calculates the size of the equivalence class $[h]_{P_=}$ for every h in S_{input} by counting the satisfying models of $P_=(h)$. The output M of Algorithm 3 is the partition $\Pi(P)$ of the original program P.

Proposition 8 (Correctness)**.** *The set S_{input} from Algorithm 2 contains a representative element for each possible equivalence class of $\Pi(P)$. Algorithm 3 calculates $\{[s_1]_{P_=}, \ldots, [s_n]_{P_=}\}$ with $s_i \in S_{input}$ which, according to (3.9), is $\Pi(P)$.*

Proof. Algorithm 2 terminates when $P_{\neq}$ is unsatisfiable. In every iteration of the algorithm one distinct valuation of the boolean formula representing a distinct output is removed by the blocking clause $l' \neq l$; thus the assertion in $P_{\neq}$ fails once all representative inputs leading to distinct outputs have been found. As the inputs lead to distinct outputs, the model counting step in Algorithm 3 find the size of every equivalence class. □

Proposition 9 (Algorithm 2 leakage bounds)**.** *If Algorithm 2 completed with an unsatisfiable $P_{\neq}$ then the channel capacity of P is bound from above by $\log_2(|S_{input}|)$.*

If the algorithm is in any other iteration, then the channel capacity of P is bound from below by $\log_2(|S_{input}|)$.

Proof. Channel capacity is shown to be reached by the cardinality of the set of events in equation (2.2). The set S_{input} is by definition the cardinality of possible unique outputs of program P when Algorithm 2 terminates, which proves the bound from above.

In the case where the algorithm is interrupted early, program P has at least as many outputs as the cardinality of S_{input} as SPEAR is sound and complete. □

5.2.2 Implementation

The implementation builds up on a toolchain of existing tools, together with some interfacing, language translations, and optimisations. See Figure 5.2 for an overview.

AQUA has the following main features:

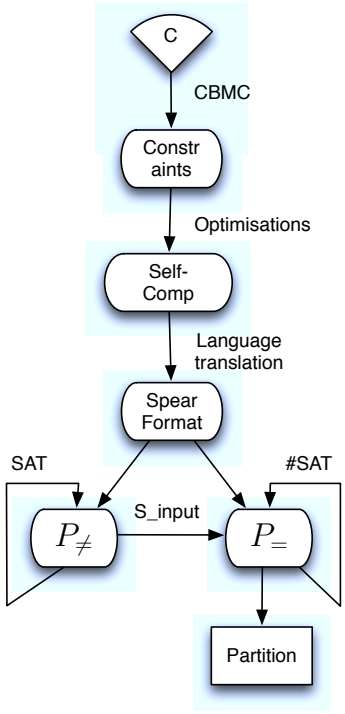

Figure 5.2: Translation steps

- runs on a subset of ANSI C without memory allocation and with integer secret variables
- no user interaction or code annotations needed except command line options
- supports non-linear arithmetic and integer overflows

AQuA works on the equational intermediate representation of the CBMC bounded model checker [17]. C code is translated by CBMC into a program of constraints which in turn gets optimised through standard program analysis techniques into cleaned up constraints[2]. This program then is self-composed and user-provided source and sink variables get automatically annotated.

[2]CBMC adds some constraints which distorts the model counting.

In a next step, the program is translated into the bit-vector arithmetic SPEAR format of the SPEAR theorem prover [2]. At this point, AQUA will spawn the two instances, $P_{=}$ and $P_{\neq}$, from the input program P.

Algorithms 2 and 3 get executed sequentially on those two program versions. However, depending on the application and cost of the SAT queries, once could also choose to execute them interleaved, by first calculating one input to the program $P_{=}$ and then model counting that equivalence class. Executing the algorithms in this order could be beneficial because it allows to calculate the entropy from below where intermediate results of the analysis already provide a bound on the entropy.

For Algorithm 2, SPEAR will SAT solve $P_{\neq}$ directly and report the satisfying model to the tool. The newly found inputs are stored until $P_{\neq}$ is reported to be unsatisfiable.

For Algorithm 3, SPEAR will bit-blast $P_{=}$ down to CNF which in turn gets model counted by either RELSAT [6] or C2D. C2D is only used in case the user specifies fast model counting through command line options. While the counting is much faster on difficult problems than RELSAT, the CNF instances have to be transformed into a *d-DNNF* tree which is very costly in memory. This is a trade-off between time and space. In most instances, RELSAT is fast enough, except in cases with multiple constraints on more than two secret input variables. The decision which backend SAT solver to use is left as choice to the user.

Once the partition $\Pi(P)$ is calculated, the user can choose which measure to apply.

Loops

The first step of the program transformations is treating loops in an unsound way, i.e. a user needs to define a fixed number of loop unwindings. This is a inherent property of the choice of tools used, as CBMC is a bounded model checker, which limit the number of iterations down to what counterexamples can be found. While this is a real restriction in program verification – as bugs can be missed in that way – it is not as crucial for our quantification purposes.

In such cases Algorithm 2 detects at one point an input which contains all inputs beyond the iteration bound. Using the principle of maximum entropy, this "sink state" can be used to always safely over-approximate entropy (for example see proposition 7 in section 4.2 for an explanation).

Let us assume we analyse a binary search examples with 15 unwindings of the loop and 8 bit variables. AQuA reports the partition

```
Partition:
{241}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}: 256
```

where the number in the brackets are the model counts. The analysis output describes 15 singleton blocks and one sink block with a model count of the remaining 241 unprocessed inputs. When applying a measure, the 241 inputs could be distributed in singleton blocks as well which would over-approximate (and in this case actually exactly find) the leakage of the input program.

Proposition 10 (Sound loop leakage)**.** *For any unwinding bound n of program P, its leakage can be overapproximated from the resulting partition $\Pi(P)_n$ by distributing its "sink state" block into singleton blocks.*

Proof. Let us assume partition $\Pi(P)_n$ is the result of n unwindings of P, and $\Pi(P)_m$ is m unwindings of P, where $m \geq n$. If every element of the "sink state" block $b \in \Pi(P)_n$ is distributed in individual blocks, the partition denoted as $\hat{\Pi}(P)_n$, then $\Pi(P)_m \sqsubseteq \hat{\Pi}(P)_n$. From property 3.4 of the semivaluation definition it follows that $H(\Pi(P)_m) \sqsubseteq H(\hat{\Pi}(P)_n)$. □

5.2.3 Worked example

Let us demonstrate each step in the analysis on a small but non-trivial example. The leakage of the polynomial $5h^2 + 2h$ is analysed by the simple translation to a C program

```
int main() {
   int l,h;

   l = 5*h*h+2*h;
}
```

where as usual `h` is the confidential variable and all variables are 8 bit wide.

Constraint translation. CBMC translates this program in the following constraint form

```
l@1#1 == 2 * h@1#0 + 5 * h@1#0 * h@1#0
```

$P_{\neq}$ and $P_{=}$ translation. AQuA translates these constraints into Spear format. The program here is $P_{=}$, while $P_{\neq}$ would be the same program with the last line inverted.

```
v 1.0
d l11__:i8 l11sp2__:i8 l11sp0__:i8 h10__:i8 l11sp1__:i8 l11:i8
  l11sp2:i8 l11sp0:i8 l11sp1:i8 h10:i8
p = h10 23:i8 # initialisation of the secret to 23
c l11sp1 * 5:i8 h10
c l11sp0 * 2:i8 h10
c l11sp2 * l11sp1 h10
c l11 + l11sp0 l11sp2
c l11sp1__ * 5:i8 h10__ # start of self-composition
c l11sp0__ * 2:i8 h10__
c l11sp2__ * l11sp1__ h10__
c l11__ + l11sp0__ l11sp2__
p = l11__ l11 # equality assertion
```

We notice the naive self-composition where the copy variables are identified by double underscores. Also, we assume that the initialisation of 23 is a representative input for some equivalence class.

Model Counting. Spear translates this program into CNF which is then model counted by invoking an all SAT solver such as Relsat. The SAT solver will return with an answer such as

```
Number of solutions: 16
Solution 1: 1 3 4 10 11 12
18 20 21 22 23 25 27 28 29 30 32
36 37 38 39 41 43 45 48 51 52 54
```

```
56 57 58 59 60 62 64 65 66 69 70
71 72 74 75 79 85 86 89 90 91 92
94 97 98 103 107 108 110 112 113
114 115 116 118 120 121 123 124
125 126 131 132 133 137 141 145
151 152 155 156 157 158 160 162
163 165 171 172 175 176 177 183
184 189 190 192 193 194 196 199
203 204 206 209 210 215 216 218
228 230 231 232 234 247 248 250
251 252 255 256 257 261 268 269
270 272 273 279 280 282 283 287
293 294 295 296 298 299 300 302
303 304 306 307 308 310
...
```

where each number in the solutions represents an active bit of a variable in the boolean formula. To get the actual members of this equivalence class, all solutions can be mapped back to individual variables (e.g. here the variable `h` consists of the 8 variables `27 28 29 30 31 32 33 34`) and the values are extracted by matching it against the active bits in the solution. Note that the enumeration of equivalence classes is optional and not part of the normal operation of AQuA. In normal operation only equivalence class sizes are calculated which avoids enumeration.

What AQuA users see. All of this is done fully automatically. A user of AQuA only has to execute

```
aqua mult.c
```

and will see the following answer:

```
Program leaks 5.2500 bits (of 8.0000)
Partition:
{4}{4}{4}{8}{4}{16}{4}{4}{4}{8}{4}{4}{4}{4}
{8}{4}{4}{16}{4}{4}{4}{8}{4}{4}{8}{4}{16}{8}
{4}{4}{4}{4}{8}{4}{4}{16}{4}{4}{8}{4}{4}{4}{4}{4}: 256
```

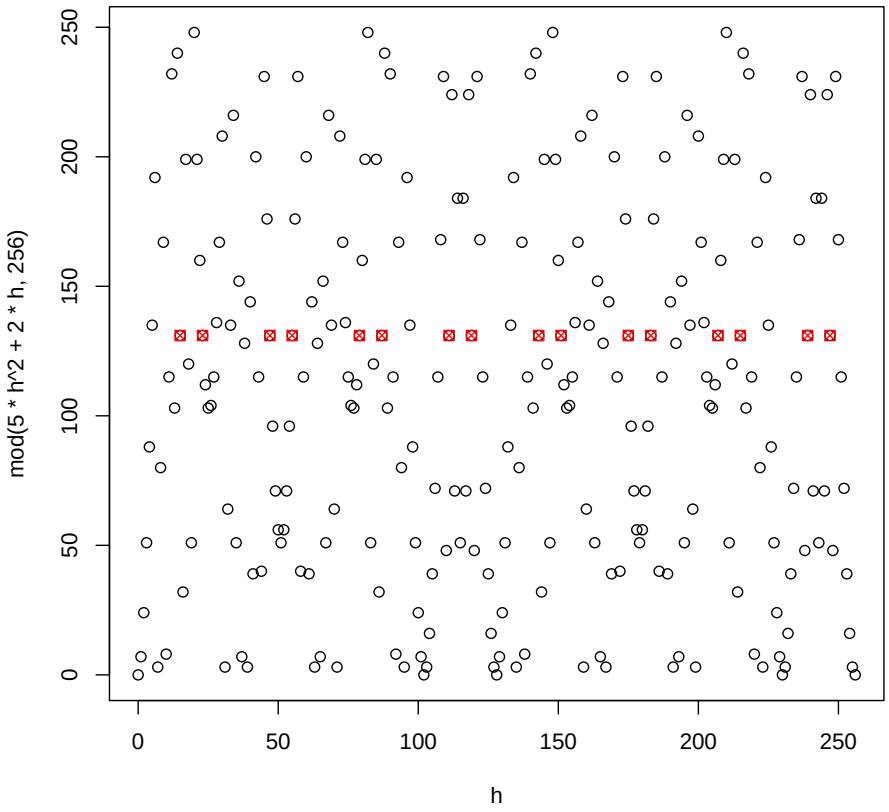

Figure 5.3: Polynomial $5h^2 + 2h$ modulo 256 for $h = \{1, \ldots, 256\}$

What does this actually mean? As the output variable is also a 8 bit variable, we can map the domain of the polynomial to a space modulo 256. This can be seen in figure 5.3. Also, for this example we have extracted an arbitrary equivalence class and its members are

```
47, 175, 239, 111, 79, 207, 143, 15,
183, 247, 55, 119, 215, 87, 23, 151
```

As they are all in the same equivalence class, these secrets should be indistinguishable for an attacker observing the output of the polynomial. A simple test in *R* verifies this; the values above are stored in the vector `ps`.

Program	$\#h$	range	Σh bits	$P_{\neq}$ Time	$P_{\neq} + P_{=}$ Time	SPEAR LOC
CRC8_1h.c	1	8 *bit*	8	17.36s	32.68s	370
CRC8_2h.c	2	8 *bit*	16	34.93s	1m18.74s	763
sum3.c†	3	$0 \dots 9$	9.96 (10^3)	0.19s	0.95s	16
sum10.c⋆	10	$0 \dots 5$	25.84 (6^{10})	1.59s	3m30.76s	51
nonlinear.c	1	16 *bit*	16	0.04s	13.46s	20
search30.c*	1	8 *bit*	8	0.84s	2.56s	186
auction.c†⋆	3	20 *bit*	60	0.06s	16.90s	42

Table 5.1: Performance examples. * 30 loop unrollings; † from [4]; ⋆ counted with *C2D* Machine: Linux, Intel Core 2 Duo 2GHz

```
> 5*ps^2+2*ps
 [1]  11139 153475 286083  61827 31363 214659 102531
[11]  1155 167811 305539 15235 71043 231555 38019 2691 114307
> mod(5*ps^2+2*ps,256)
 [1] 131 131 131 131 131 131 131 131
 131 131 131 131 131 131 131 131
```

While all the values evaluate to different points in the polynomial (as it is a strictly increasing function) they all collapse to the same value under modulo 256. Thus, they are indistinguishable when stored in a variable of 8 bit. These 16 points have been highlighted in figure 5.3. Naturally, they lie on a horizontal line at 131. This example also shows how precise modelling of integer overflows is crucial for making the analysis work.

5.3 Experiments

Table 5.1 provides a performance benchmark comparing the two algorithms for $P_{\neq}$ and $P_{=}$ on different problems. The run times have been split between Algorithm 2 to calculate $P_{\neq}$ and the total run time; the lines of code (LOC) column is a measure of the size of the generated code in SPEAR format.

The biggest example is a full CRC8 checksum implementation where the input are two `char` variables (16 bit). The program has been chosen because its output is well known and can serve as a check if AQUA calculates the

equivalence class number and sizes correctly. As expected, AQuA always reported a leakage of 8 bit and a uniform bucketing of the inputs over the 256 equivalence classes. Also, as this code produces a large number of outputs, the runtime between the two algorithms are roughly equal.

The examples `sum3.c` and `auction.c` are taken and adapted from the paper [4]. The performance of those examples has not been discussed in the mentioned paper. The auction example is a simple loop where the output is the highest bidder (variable `l`) of an auction, however the actual bids (`h[i]`) are confidential

```
...
l=0;
for(i=0; i<3; i++) {
  if(h[i] > h[l]) l = i;
}
```

As there are only three outputs almost all time is spent in the model counting algorithm. The other example `sum3.c`, and the related `sum10.c`, is a test to see how the tool copes with multiple confidential variables, from 3 to 10. Multiple confidential variables very quickly lead to difficult SAT instances which become intractable. The value range of 10 values in `sum3.c` had to be reduced to 6 values and the instance had to be model counted with the C2D backend tool for it to terminate within reasonable time.

The example `search30.c` is a loop unrolling test. The code itself is very short, however due to 30 loop unrollings and self-composition the tool generated 186 lines of code out of the following code

```
l=0;
while(l < h) {
  l++;
}
```

However, as the generated code consists mostly of control flow decisions, which AQuA handles well, the runtime to calculate the leakage of this program is below 3 seconds.

Program	Σh bits	Enumeration Time	AQuA x times faster
sum3.c	9.96	12.5s	13x
sum10.c	25.84	$\perp$	-
nonlinear.c	13	10m32s	90x

Table 5.2: Runtime comparison to the enumeration tool in Chapter 4

Finally, the example `nonlinear.c` demonstrates the ability to handle integer overflows and non-linear arithmetic correctly (due to the bit vector nature of Spear). The code is the following with the assumption of 16 bit variables

```
tmp = h * h;
if(tmp < 24) l=0;
else if(tmp > 32) l=1;
else l=2;
```

Clearly, it only produces three outputs. The interesting aspect however is that only very few values of `h` squared will lead to an output of `l=2`. When run, AQuA reports the following partition

```
{292}{65240}{4}
```

Where the largest equivalence class must be responsible for output 1, the one with 292 inputs is most likely the one for output 0, and then there is a class with only 4 inputs where the squared and truncated to 16 bit values of `h` should be 25. An enumeration of this equivalence class gives the values $\{32763, 5, 32773, 65531\}$. A simple calculator confirms that all of these numbers squared and modulo 16 are 25, i.e. the only possible integer square between 24 and 32.

5.3.1 Comparison to Chapter 4

A direct comparison between the tool performances of the last chapter and this chapter was not possible. Language restrictions of the previous tool (e.g. no arrays) would make it difficult to provide all the examples from table 5.1. However, table 5.2 gives the reader an idea of the performance advantage of the SAT based tool versus direct enumeration.

AQuA was 13 times faster than enumeration for program `sum3.c`. The program `sum10.c` did not terminate within 30 minutes; a downscaled version of `nonlinear.c`, from 16 to 13 bit secret size, was 90 times faster with AQuA against enumeration.

5.4 Application: database queries

This section describes how AQuA could be applied to measure leakage of combined statistical database queries. Database queries are modelled as programs; our analysis tool calculates the partition of states of that program and in turn quantifies the leakage of the encoded queries. This section is not about showcasing the performance of AQuA but to illustrate the potential width of applications of automatically quantifying leakage.

We will use concepts used by Dobkin et al. [24] to describe databases.

Definition 18. *A database D is a function from $1, \ldots, n$ to $\mathbb{N}$. The number of elements in the database is denoted by n; $\mathbb{N}$ is the set of possible attributes.*

A database D can also be directly described by its elements $\{d_1, \ldots, d_n\}$, with $D(i) = d_i$ for $1 \leq i \leq n$. For a database with n number of objects, a query is an n-ary function. Given D, $q(D) = q(d_1, \ldots, d_n)$ is the result of the query q on the database D.

We assume that a database user can choose the function q and restrict its application to some of the elements of $\{d_1, \ldots, d_n\}$, depending on the query structure. However, the user can not see any values the function q runs on.

An arbitrary query is translated by the following transformation

$$Q_1 = q(d_i, \ldots, d_j) \quad \Rightarrow \quad \mathtt{l_1} = \mathtt{e}(\mathtt{h_i}, \ldots \mathtt{h_j})$$

where the function q applied to $(d_i, \ldots, d_j)$ is rewritten to some C expression e[3] on the secret variables $\mathtt{h_i}, \ldots, \mathtt{h_j}$, where $\mathtt{h_n}$ is equal to d_n for all $i \leq n \leq j$; the output is stored in the observable variable $\mathtt{l_1}$. A sequence of queries $Q_1, \ldots, Q_n$

[3]Expressions usually used in statistical database are `sum, count, average, mean, median` etc. Our context is general so any C expression could be used

results in tuples of observable variables $(\mathtt{l}_1, \ldots, \mathtt{l}_n)$. We denote the partition of states for a query Q_i, after the transformation above, as $\Pi(Q_i)$.

5.4.1 Database inference by examples

To measure the degree of database inferences possible by a sequence of queries we define the following ratio, comparing leakage with the respective secret space

Definition 19 (SDB Leakage Ratio)**.** *Given an SDB, let $Q_1, \ldots, Q_n$ be queries, and $h_1, \ldots, h_m$ be the involved secret elements in the database. The percentage of leakage revealed by the sequence of queries is given by*

$$\frac{H(\bigsqcup_{1 \leq i \leq n} \Pi(Q_i))}{H(h_1, \ldots, h_m)} \tag{5.1}$$

In the definition we can use definition 14 to compute $\bigsqcup_{1 \leq i \leq n} \Pi(Q_i)$

Max/Sum Example. Two or more queries can lead to an inference problem when there is an overlap on the query fields. Assume two series of queries:

$$Q_1 = \max(h_1, h_2) \quad Q_2 = \mathrm{sum}(h_3, h_4)$$

The first series of queries ask for the max and sum of two disjoint set of fields. The two queries don't share any common secret fields, so Q_1 does not contribute to the leakage of Q_2.

$$Q'_1 = \max(h_1, h_2) \quad Q'_2 = \mathrm{sum}(h_1, h_2)$$

It is a different picture if the two queries run on the same set of fields, as shown in Q'_1, Q'_2. Intuitively, we learn the biggest element of the two and we learn the sum of the two. The queries combined reveal the values of both secret fields, i.e. $sum - max = min$.

Assuming 2 bit variables, we get the following calculations:

$$H(\Pi(Q_1)) = 1.7490 \quad H(\Pi(Q_2)) = 2.6556 \quad H(\Pi(Q_1) \sqcup \Pi(Q_2)) = 4.4046$$
$$H(\Pi(Q'_1)) = 1.7490 \quad H(\Pi(Q'_2)) = 2.6556 \quad H(\Pi(Q'_1) \sqcup \Pi(Q'_2)) = 3.25$$

Contributor	**Industry**	**Geograph. Area**
C_1	Steel	Northeast
C_2	Steel	West
C_3	Steel	South
C_4	Sugar	Northeast
C_5	Sugar	Northeast
C_6	Sugar	West

Table 5.3: Contributors

Contributing Group	**Amount**
Steel	$h_1 + h_2 + h_3$
Sugar	$h_4 + h_5 + h_6$
...	...
Northeast	$h_1 + h_4 + h_5$
...	...

Table 5.4: Summary Table for Contributors

The measure of how much of the secret the two series of queries revealed is the ratio between the join of the queries to the whole secret space:

$$\frac{H(\Pi(Q_1) \sqcup \Pi(Q_2))}{H(h_1, h_2, h_3, h_4)} = \frac{4.4046}{8.0} \approx 55\% \quad \frac{H(\Pi(Q_1') \sqcup \Pi(Q_2'))}{H(h_1, h_2)} = \frac{3.25}{4.0} \approx 81\%$$

where we have used H, the Shannon entropy as the leakage measure[4]. The 3.25 bits, or 81% of the secret, is the maximal possible leakage for the query, as we still don't know which of the two secrets secret was the bigger one of the two, however "everything" is leaked in a sense, while the first query only reveals 55% of the secret space.

For the enforcement, we could think of a simple monitor which keeps adding up the information released so far for individual users and which would refuse certain queries in order to not reveal more than a policy allows. A policy can be as simple as a percentage of the secret space to be released.

Sum Queries Inference. Consider a database storing donations of contributors to a political party from the steel and sugar industry, contributors coming from several geographical areas. Given Tables 5.3 and 5.4, a user is

[4]Taking a different measure like min entropy we would get 40% and 75% respectively

allowed to make sum queries on all contributors which share a common attribute (Industry or Geographic Area)[5]. Table 5.4 summarises all possible queries, where the amount donated by each contributor C_i is represented by the value h_i.

In this scenario, the owner of the databases wants to make sure that no user can learn more than 50% of the combined secret knowledge of what each contributor donated.

We will look at two users querying the database; the queries of the first user fulfill the requirements of the database owner, the second user (who happens to be contributor C_1) is clearly compromising the database information release requirements.

User 1 is making two queries

$$Q_1 = \text{sum}(h_1, h_2, h_3) \quad Q_2 = \text{sum}(h_4, h_5, h_6)$$

In other words, User 1 is asking for the sum of the contributors from the steel and sugar industry. For simplicity, we assume only 2 bit variables for each contributor h_i. AQuA calculates a partition with 100 equivalence classes, and a Shannon entropy of 5.9685 of total 12 bits.

This results in a ratio of

$$\frac{H(\Pi(Q_1) \sqcup \Pi(Q_2))}{H(h_1, \ldots, h_6)} = \frac{5.9685}{12} \approx 49.73\%$$

which is just within the requirements of 50% information leakage.

User 2, who is contributor C_1, is inquiring the following two queries:

$$Q_3 = \text{sum}(h_4, h_5, h_6) \quad Q_4 = \text{sum}(h_1, h_4, h_5)$$

Here, Q_3 and Q_4 have an overlap in the fields h_4 and h_5. Since User 2 is C_1, the field h_1 is known, so with these two queries, User 2 is able to learn h_6, i.e. $h_6 = Q_3 - Q_4 + h_1$. The substantial knowledge gain of User 2 is revealed in

[5]Example adapted from [24]

the leakage ratio

$$\frac{H(\Pi(Q_3) \sqcup \Pi(Q_4))}{H(h_1, h_4, h_5, h_6)} = \frac{H(\Pi(Q_3) \sqcup \Pi(Q'_4))}{H(h_4, h_5, h_6)} = \frac{4.6556}{6} \approx 77.6\%$$

where in the second equation term h_1 in the denominator disappear because contributor C_1 knows h_1 (similarly $Q'_4 = \text{sum}(h_4, h_5)$)[6]. If our tool was evaluating the information leakage of these queries before the result was reported back to the user, then Q_4 could be denied for User 2.

We can see the previous database as an (easily computable) abstraction of a real database with a large number of entries. In this case C_1 could represent the *set* of contributors form the Steel industry in the Northeast. In this case the leakage ratio would tell us the amount of information the queries leak about the *group* of individual (or set of secret data). We can hence extract valuable information about the threat of a set of queries by automatically computing the leakage on an abstraction of a database. This measure can be combined with more classical query restriction techniques like set size and overlap restriction within a threat monitor. While a precise theory of this monitor is beyond the scope of this work we believe the ideas are sound and workable.

5.5 Review of technique

This chapter described a push-button static analysis tool called AQuA which calculates precise partitions on multiple confidential variables generated by C programs. It is based on a number of different tools and algorithms using SAT solving and model counting.

This approach addresses a number of deficiencies from the approach in the previous chapter such as

- possibility of analysing fragments of source code, from whole functions down to individual lines of code

[6]To understand the numbers 4.6556 comes by the fact that the queries reveal h_6 i.e. 2 bits, plus $\text{sum}(h_4, h_5)$ which is 2.6556 bits

- avoiding complete enumeration of secret space by the choice of only counting
  - number of equivalence classes
  - sizes of equivalence classes
- support for multiple integer confidential variables
- runs on a subset of C instead of a simple while language

AQuA is arguably more scalable than the dynamic approach, as demonstrated in analysing 700+ lines of generated code or calculating the leakage of a 60 bit secret space below 17 seconds. However, predicting or bounding the runtime of this program analysis is hard. Mapping from a set of features of a problem instance (e.g. the description of a program) to the predicted runtime is known as *empirical hardness modelling* [37] where problems as difficult as the ones built by AQuA are, to the best of our knowledge, not within the scope of that research area yet.

Still, there are a number of factors involved when evaluating the performance and scalability of the tool. The runtime depends on

1. number of lines of code and therefore number of variables and clauses in the CNF instance
2. number and size of confidential variables
3. arithmetic operations used in the code
4. number of equivalence classes described by the program

All of this basically reduces to how hard the CNF instances are to solve which is difficult to predict as mentioned above. However, points 3 and 4 deserve further clarification. On point 3: when a program is translated to CNF a technique called bit-blasting is applied which reduces arithmetic bit-vector operations to boolean circuits. This can lead to very complex and difficult to solve circuits for certain operations such as multiplication and modulus. On point 4: the algorithm to find the number of equivalence classes, $P_{\neq}$, adds a

blocking clause for every equivalence class found. The formula grows linearly with the number of equivalence classes which will slow down the analysis in case there are a large number of equivalence classes to be found.

5.5.1 Improvements

The algorithms and techniques used to implement AQuA are completely unoptimised. There are a number of features to add which could easily improve the performance two-fold or more (obviously still dominated by any exponential explosion of states). The two most important points are

- Only naive self-composition has been implemented where a full copy of the program is appended to the original program. More clever compositions could save a lot of unnecessary variables and thus reduce complexity
- Almost all interactions between the different programs are performed via input-output and parsing instead of direct API calls (mostly because these APIs don't exist). Switching to API interactions could dramatically increase the performance.

Chapter 6

Applying Model Checking to Verify Leakage Policies

So far, we have presented automatic methods to compute the precise information leakage in C programs. To reach such precision there is no way around computing the whole partition of the high values one way or the other. The previous chapter showed algorithms to perform this calculation in a more scalable and useful way than just brute-force complete enumeration. Most importantly, the tool is able to calculate the number of equivalence classes independently from calculating the sizes of the equivalence classes. This allows to *bound the leakage* from above and might give a good indication on the nature of the leakage – i.e. large leak or small leak. Still, the leakage computation done in this way will always depend on the secret size; for secret sizes of more than a few bits this is computationally prohibitive.

In this chapter, we take the idea of bounding the leakage a step further. Instead of computing the precise leakage, we ask the simpler question "does the program leak more than M bits" where M is a reasonable, externally provided choice given a larger context. We call such a decision question a *quantitative policy*. The crucial insight is that checking a quantitative policy is a k-safety problem which is bounding the channel capacity based leakage of the program.

Checking whether such a policy holds or is violated allows for a different style of analysis: the confidential information is modelled as *nondeterminism*

and a *driver* code is provided to each program which asserts the number of distinctions which are allowed to be observable for the program. This code is then run by a model checker whose sole purpose is to find a number of confidential values (drawn from a nondeterministic source) which violate such a policy.

This chapter largely consists of the successful application of this quantitative leakage analysis on Linux Kernel device driver code. We chose to apply our method to reported information leakage vulnerabilities in the Linux Kernel and to common authentication routines. All of the covered vulnerabilities are indexed by the standardised vulnerability repository CVE from Mitre[1]. The vulnerability description are quite detailed because this is the first account of applying such verification techniques to quantifying real information leakage bugs, thus it is interesting and important to understand the nature of the leaks.

The drivers can be run by off-the-shelf symbolic model checkers such as CBMC [17], where this is our choice of verification tool. CBMC is a good choice for several reasons: (i) it makes it easy to parse and analyse large ANSI-C based projects (ii) it models bit-vector semantics of C accurately which makes it able to detect arithmetic overflows amongst others, which turns out to be important (iii) nondeterministic choice functions are provided to easily model user input, which also enjoys efficient solving due to the symbolic nature of the model checker (iv) despite being a bounded model checker, CBMC can check whether enough unwindings of the transition system were performed which prove that there are no deeper counterexamples.

Our experiments show that the analysis not only quantifies the leakage but for certain instances also helps understanding the nature of the leak. In particular, the counterexample produced by the model checker, when a leakage property is violated, can provide insights into the cause of the leak. For example, we can extract a public user input from the counterexample needed to trigger a violation.

Another surprising result of our experiment is that in certain circumstances we were able to use our technique to prove whether the official patches provided for the vulnerabilities actually eliminate the information leak. This is achieved

[1]http://cve.mitre.org, CVE is industry-endorsed with over 70 companies actively involved

by point (iv) from above, when the model checking process is complete.

In summary the main technical contributions of this chapters are the following:

1. We present the first quantitative leakage analysis of operating system code.

2. We show how to express Quantitative Information Flow properties that can be efficiently checked using bounded symbolic model checking.

3. We show that the technique not only quantifies leakage in real code but also provides valuable information about the nature of the leak.

4. In some cases we are able to prove that official patches for reported vulnerability do indeed eliminate leakage; these constitute the first positive proofs of absence of QIF vulnerabilities for real-world systems programs.

This work has been published and presented at the Annual Computer Security Applications Conference (ACSAC) 2010 with proceedings published by ACM [30].

6.1 Model of programs and distinctions

A transition system as described in section 3.4.1 is used to model programs. Again, we are interested in the input/output behaviour of a C function where inputs are formal arguments to the function and outputs are either return values or pointer-type arguments. The partition $\Pi_l(P)$ from equation 3.9 in section 3.4.1 is used to describe the mapping between confidential inputs and publicly observable outputs given a low input choice of l.

Formally, we define a *quantitative policy* as a non-negative natural number N. A partition $\Pi_l(P)$ breaches a policy if $|\Pi_l(P)| > N$, where $|\Pi_l(P)|$ is the number of equivalence classes of $\Pi_l(P)$. This number describes the number of distinct outputs of program P, or equivalently the maximal cardinality for $\Pi_l(P)$. We refer to it as the number of *distinctions* on the secret the program makes.

In our model, the question whether a program violates a policy can be formulated by the following decision question

$$\lceil \Pi_l(P) \rceil = \max_{l \in L} |\Pi_l(P)| \leq N \tag{6.1}$$

where the verification task is to find a low input l with respect to N where the cardinality of the resulting partition violates the policy. This search can be interpreted as a powerful attacker who is able to pick the most damaging low input l for a given policy N. If there is no such l where $\Pi_l(P)$ violates the policy then the program satisfies the policy.

The decision whether a program P satisfies such a policy gives a bound on the channel capacity. Malacaria and Chen [39] proved that the channel capacity of P is just $\log_2(\lceil \Pi_l(P) \rceil)$.

The next section will show the relationship between a channel capacity based leakage policy and k-safety, as a mean to easily check such policies.

6.1.1 Checking policies and k-safety

Checking a policy for a fixed N distinctions is a k-safety problem. This has been proved by Yasuoka and Terauchi [66] and previously implied by Malacaria and Chen [39].

A k-safety property has been formally described by [66] as follows

Definition 20 (k-safety property)**.** *A property $Q \subseteq Prog \times \mathbb{N}$ is a k-safety property iff $(P, N) \notin Q$ implies that there exists $T \subseteq [\![P]\!]$ with cardinality $|T| \leq k$ and $\forall P'.T \subseteq [\![P']\!] \implies (P', N) \notin Q$.*

where in this specific case, the property $Q = \{(P, N) \mid \lceil \Pi_l(P) \rceil \leq N\}$.

Informally, the definition can be understood in the following way: a decision can be made if a program satisfies the property or not by observing a counterexample $T \subseteq [\![P]\!]$ with size $|T| \leq k$. Also, for all other programs where T is also a valid trace, they are all not satisfying the property either. Since we consider input/output semantics, $|T|$ is the number of runs necessary to refute the property.

In our case, if a program and natural number N is not in the property, $(P, N) \notin Q$, means that it is possible to count $N + 1$ distinct input/output pairs in some trace T. This makes the policy check a $N + 1$-safety check [66].

Thus, our property states that every program contained in the property does not make more than N distinctions on the output. As this coincides with the channel capacity measure the outcome of the check has the following quantitative implication

Also notice that "verified" in the following proposition assumes a complete analysis.

Proposition 11. *For a program P, maximising low choice l, and policy N, if equation 6.1 is*

- *verified then* $\log_2(N)$ *is an upper bound*
- *violated then* $\log_2(N + 1)$ *is a lower bound*

on the channel capacity of the program P.

Proof. Immediate through the channel capacity result in equation 2.2 and from definition 20. □

The next section will use the fact that every k-safety problem can be reduced to a normal safety problem using self-composition as a way to encode policies in a driver function.

6.2 Encoding distinction-based policies

A program violates a quantitative policy if it makes more distinctions than what is allowed in the policy. A leaking program is one breaching the policy $N = 1$ in the above definition.

We take ideas from assume-guarantee reasoning [52] to encode such a policy in a driver function, which tries to trigger a violation, i.e. producing a counterexample, of the policy. If the policy states that the function `func` is not allowed to make more than 2 distinctions then this is modelled as shown in

```
int h1,h2,h3;
int o1,o2,o3;

h1 = input();  h2 = input(); h3 = input();

o1 = func(h1);
o2 = func(h2);
assume(o1 != o2); // (A)

o3 = func(h3);
assert(o3 == o1 || o3 == o2); // (B)
```

Program 1: Example driver checking for 2 distinctions

Program 1. This driver only has a high component as a state, which is passed to the function `func` where the policy is tested on.

Drivers always have a similar structure: we model the secret by a nondeterministic choice function `input()` as a placeholder for all possible values of that type; then for a policy of checking for N distinctions, the function under inspection is called N times. The crucial step (A) is the use of the `assume` statement after the calls: the driver assumes that, in this case, there are two different return values found already. The function is called an $N + 1$th time and at (B) the driver asserts that the next output is either one of the previously found outputs.

The `assume` statement only considers execution paths which satisfy the given boolean formula, all other paths are rejected. Further, the bounded model checker used will try to find a counterexample to the negated assertion claim, which is only satisfiable if and only if a counterexample exists. An unsatisfiable formula means that the original claim holds, i.e. the program conforms to the policy. The verification condition generated by the bounded model checker for the policy in Program 1 is:

$$\texttt{o1 != o2} \implies \texttt{(o3 == o1 || o3 == o2)}$$

Where the bounded model checker tries to find a counterexample (execution path) using the negated claim such that the following holds

$$\texttt{o1 != o2} \land \texttt{o3 != o1} \land \texttt{o3 != o2}$$

Input: Function `func`, types `t,t',t"`, comparison `eq_t`, bound k, threshold N
Output: Driver.c

```
t o_1, ..., o_n, o_n+1;
t' h_1, ..., h_n, h_n+1;
t'' l;

h_1 = input(); ... h_n = input(); h_n+1 = input();
l = input();
o_1 = func(h_1, l);
⋮
o_n = func(h_n, l);
assume(!eq_t(o_1, o_2) && !eq_t(o_1, o_3) && ...);

o_n+1 = func(h_n+1, l);
assert(eq_t(o_n+1, o_1) || eq_t(o_n+1, o_2) || ...);
```

Algorithm 4: Template to syntactically generate a driver for an N distinction policy

i.e. that there are three distinctions possible.

Another possibility is that the function `func` does not even make two distinctions, such that the `assume` statement at point (A) is always false, which leads to proving the policy (or any policy) vacuously true, because for any assertion `Q` the verification condition is true, i.e. `false` $\Longrightarrow$ `Q`.

6.2.1 Bounded model checking

We use the bounded model checker CBMC to verify or falsify a policy. CBMC encodes an ANSI-C program into a propositional formula by unwinding the transition relation and user defined specifications up to some bound. This formula is only satisfiable if there exists an error trace violating the specification.

The tool can also check if the unwinding bound is sufficient by introducing *unwinding assertions*, which are assertions on the negated loop guards. This ensures that no longer counterexample can exist than the used bound. To *prove* any properties the analysis has to pass unwinding assertions, otherwise it can only be used as a way to find counterexamples up to the unwinding

bound.

The C program gets encoded into constraints C and the property – user defined assertions – are encoded in P. Then the model checker tries to find a satisfiable assignment to the formula

$$C \wedge \neg P$$

where P is an accumulation of the assumptions and assertions made in the program text. Thus if there are two `assume` statements in the driver with expressions E_1 and E_2 and one `assert` statement with expression Q then P is

$$P \equiv E_1 \wedge E_2 \implies Q.$$

6.2.2 Driver

A general template for a driver is described in Algorithm 4. The inputs to the algorithm are the function `func` to be analysed, possibly up to three different types for the input/output pair $\langle (h, l), o \rangle$, and a comparison function `eq_t` which returns true if the arguments of type `t` are equal, where `t` is the type of the observation of function `func`. This comparison function could be as simple as `==` of C, or a more complex function, such as `memcmp`, if `t` is an array or string. Also note that the observations `o_i` do not need to be only return values, but can also be variables passed by reference to `func`.

Proposition 12 (Correctness of driver template)**.** *If the driver template in Algorithm 4 is successfully verified up to a bound k (i.e. the negated claim is unsatisfiable) then the function* ***`func`*** *does not make more than N distinctions on the output within the bound k. Formally, we state that the correctness of the driver implies the correctness of the policy specified. We prove this by showing that the structure of the driver encodes a limit on the maximum possible distinctions made by the function* ***`func`****.*

$$\begin{aligned} o_1 \neq o_2 \wedge o_1 \neq o_3 \wedge \cdots \wedge o_{n-1} \neq o_n \\ \implies o_{n+1} = o_1 \vee \cdots \vee o_{n+1} = o_n \end{aligned}$$

Proof. We prove the proposition by showing that the triple `assume(E); S; assert(Q)` generates the above implication through weakest precondition semantics on the structure of the driver:

$$\begin{aligned} & wp(assume(E); S, Q) \\ & wp(assume(E), wp(S, Q)) \\ E & \implies wp(S, Q) \end{aligned}$$

with $S \equiv o_{n+1} = \texttt{func}(h_{n+1}, l)$, $Q \equiv o_{n+1} = o_1 \vee \cdots \vee o_{n+1} = o_n$, and $E \equiv o_1 \neq o_2 \wedge o_1 \neq o_3 \wedge \cdots \wedge o_{n-1} \neq o_n$ we get by substitution:

$$\begin{aligned} E & \implies Q[\texttt{func}(h_{n+1}, l)/o_{n+1}] \\ o_1 \neq o_2 \wedge o_1 \neq o_3 \wedge \cdots \wedge o_{n-1} \neq o_n & \implies \texttt{func}(h_{n+1}, l) = o_1 \vee \cdots \vee \texttt{func}(h_{n+1}, l) = o_n \end{aligned}$$

□

Thus, we can make the following claims on the result of the model checking process: For a given bound k and a policy,

- if the model checker finds a counterexample then the policy is violated, i.e. the program makes more distinctions than specified.
- if the process ends with a successful verification of the policy without unwinding assertions then the policy holds up to an unwinding of k. This result is complete *up to bound k*.
- if the process ends with a successful verification of the policy *with* unwinding assertions then the policy holds for any number of iterations. This result is complete.

6.3 Checking policies in practice

The steps in checking a program or function for the compliance with a quantitative policy are as follows:

1. Define the input state (h, l) and output state o in the code, i.e. the confidential input h, the low input l, and the observation o
2. Define the maximum number of distinctions allowed by the policy and an unwinding factor k
3. Generate a driver function using the template in Algorithm 4
4. Run CBMC on the driver. If the driver is successfully verified, potentially increase the unwinding factor or add the unwinding assertion

6.3.1 Modelling low input

A crucial aspect of the analysis is to model low user input, which is often responsible for triggering a bug which causes the information leak. These bugs only happen on a very restricted number of execution paths and could be exploited by a malicious user choosing a special user input. This scenario generally applies when studying many CVE reported information leakage vulnerabilities.

Let us look at the following simplified code in Program 2, which contains an integer underflow, taken from the vulnerability CVE-2007-2875 in the linux kernel.

At first, it seems not possible that the point (C) where the secret `h` gets returned is ever executed; exactly that check is done in (A) which reduces the variable `nbytes` to be within the bound `bufsz`. However, due to wrong choice and combination of types, the subtraction in (B) causes an underflow in `nbytes` for a very large `ppos` value. Unfortunately, `ppos` is a user controlled input variable, such that when its value is chosen carefully, point (C) is reached.

In this example, a state in the system is the tuple (h, l) which represents the arguments to the function `underflow`, i.e. the formal parameters `h` and `ppos`; observations are the return values of this function. The generated driver can automatically find the low part of a state which triggers such subsequent information leaks, because the analysis instructs the model checker to find *any* possible execution path satisfying the assumptions and assertions on the

```
typedef long long loff_t;
typedef unsigned int size_t;
int underflow(int h, loff_t ppos) {
  int bufsz;
  size_t nbytes;
  bufsz=1024;
  nbytes=20;

  if (ppos + nbytes > bufsz) // (A)
       nbytes = bufsz - ppos; // (B)
  if(ppos + nbytes > bufsz) {
     return h; // (C)
  } else {
     return 0;
  }
}
```

Program 2: Integer underflow causing a leak

outputs, given nondeterministic high values and fixed low inputs. As SAT-based model checking is precise down to the individual bit, it will find a low input which triggers the underflow and uncovers the leak.

CBMC generates a counterexample falsifying a policy of e.g. no leakage and thereby having triggered the integer underflow. The following excerpt of the counterexample

```
State 14 file underflow.c line 40 function main
------------------------------------------------
  underflow::main::1::l=1706688912 (00000000...
....
State 35 file underflow.c line 13 function underflow
------------------------------------------------
  underflow::underflow::1::nbytes=4027596816 (11110000...
```

shows that a low input of `l=1706688912` lead to an `nbytes` which underflowed from the previous value 20.

Clearly, for such leaks to be detected it needs bit-level precise reasoning, just like SAT-based bounded model checkers support.

6.3.2 Modelling environments

In model checking, the environment, such as library function calls or generally functions and data structures which have no implementation, need to be modelled in a way which allows for the property to be verified. Out of the box, CBMC replaces function calls without implementation with nondeterministic values.

As our analysis needs to check for equality on inputs and outputs of functions a certain number of common library functions have to be modelled in a way which preserves their original semantics. For example, the usual library C functions `memcmp`, and `strcmp` are implemented in a way which return 0 if their arguments are equal and a value not equal to 0 if they are not equal. The functions `memset` and `memcpy` actually set an array of integers or characters to a certain value or to the content of another array. The same applies to linux kernel utility functions such as `copy_to_user` and `copy_from_user` which copy memory blocks to or from user space.

For example, a `memcmp` implementation is shown in Program 3.

```
int memcmp(void *s1, void *s2, unsigned int n) {
  int i;
  char *us1,*us2;

  us1 = (char*) s1;
  us2 = (char*) s2;
  for(i=0;i<n;i++) {
     if(us1[i] != us2[i]) return -1;
  }
  return 0;
}
```

Program 3: Simplified `memcmp` model

These library functions do not have to follow the exact semantics of their original implementation but merely have to be precise enough to represent the necessary distinctions on the confidential variables. For example, the given `memcmp` only returns the values 0 and -1 while the original libc function has more complex return values. However, those return values are not needed for

the purposes of the analysis as the only condition we rely upon is that if the function returns 0 then all characters in both strings `s1` and `s2` are equal. How to write these models needs understanding of the code which the analysis will be run on.

A properly modelled environment contains all of the following points

- All types, type definitions, and preprocessor constants are provided
- Models for all library functions are provided
- All pointers referring to global structures have been initialised
- The confidential variables or structures have been assigned nondeterministic values

It is clear that this involves a significant amount of manual effort to provide a good environment. However, to put the effort in perspective, if there is a large amount of work necessary to provide an environment then it can most likely be reused for analysing different functions in the code base; if the code to be analysed is small then the work to provide the model is small.

6.3.3 Modelling confidential values

The big advantage and difference between checking and inferring quantitative information flow is that checking is independent of the range of confidential values and solely depends on the number of distinctions which need to be checked. Thus, it does not matter whether the confidential variable contains 1024 bits or 100 megabytes as long as the source the values are drawn from allows for enough distinct values to violate the property. Confidential variables are modelled using nondeterministic choice functions which are provided by CBMC. Every primitive type has its own function returning nondeterministic values, such as `nondet_int()`, `nondet_char()`, etc.

The simplest, runnable driver is shown in Program 4. The driver encodes a policy of 2 distinctions and the function `f` returns nondeterministic integers between 0 and 1 where the variable `tmp` implicitly models the confidential

```
int f() {
  int tmp = nondet_int() % 2;
  __CPROVER_assume(tmp >= 0);
  return tmp;
}

int main() {
  int o1,o2,o3;

  o1 = f();
  o2 = f();
  __CPROVER_assume(o1 != o2);

  o3 = f();
  assert(o3 == o1 || o3 == o2);
}
```

Program 4: Simple implementation of template in Algorithm 4

values. The function `f` satisfies the policy as long as it does not produce more confidential values than the drivers checks for.

For more complex types (structures) allocation functions have to be provided to nondeterministically initialise all of the fields of the types. An example allocation function used in one of the linux kernel experiments is the one in Program 5.

6.4 Experimental results

We applied our technique to CVE reported information leakage vulnerabilities in the Linux Kernel. In the experiments we checked for policy violations and proved whether official patches resolve the information leakage. Notice that proving the absence of information leakage could also be done by a noninterference check, which is the special case of a policy $N = 1$. We also analysed authentication routines of the Secure Remote Password protocol (SRP) and of a Internet Message Support Protocol implementation. A summary of the results is shown in Table 6.1. The leakage is reported in the second last column where $> \log_2(N)$ means that more than $\log_2(N)$ bits leaked, i.e. the policy N

```
struct sockaddr_at* alloc_sockaddr_at() {
  int i;
  struct sockaddr_at* tmp = (struct sockaddr_at*)
                            malloc(sizeof(struct sockaddr_at));

  for(i = 0; i < 8; i++) {
    tmp->sat_zero[i] = nondet_char();
  }
  tmp->sat_len = nondet_uchar();
  tmp->sat_family = nondet_uchar();
  tmp->sat_port = nondet_uchar();

  return tmp;
}
```

Program 5: Allocation function for a `sockaddr_at` type

Description	CVE Bulletin	LOC	$k^\star$	Patch Proof	$log_2(N)$	Time
AppleTalk	CVE-2009-3002	237	64	✓	>6 bit	1h39m
tcf_fill_node	CVE-2009-3612	146	64	✓	>6 bit	3m34s
sigaltstack	CVE-2009-2847	199	128	✓	>7 bit	49m50s
cpuset†	CVE-2007-2875	63	64	×	>6 bit	1m32s
SRP getpass	–	93	8	✓	≤1 bit	0.128s
login_unix	–	128	8	–	≤2 bit	8.364s

Table 6.1: Experimental Results. $\star$ Number of unwindings † From Section 6.3.1

has been violated; equally, $\leq \log_2(N)$ means the policy N has been verified. These two cases correspond to lower and upper bounds on the leakage.

6.4.1 Linux kernel

We define information leakage in the kernel always as parts of the kernel memory which gets mistakenly copied to user space, i.e. the virtual memory allocated to conventional applications. Clearly, this should not happen as anything allocated in the kernel space is not meant to be seen by users (except within the bounds of normal user/kernel interactions), especially in multi-user systems like Linux. Thus, in all examples the kernel memory is modelled as

nondeterministic values.

The interface between user and kernel space are system calls or syscalls in short. Syscalls, like normal functions, have a number of arguments and a return value where the kernel can transfer data structures or single values back and forth. This is the crucial point in the system where information leakage is most common.

AppleTalk. The specific vulnerability CVE-2009-3002 in the appletalk network code shows a quite common cause of information leakage: a user requests, by a syscall, that a structure gets filled with values and returned to user land. The developer however forgot to assign values to all fields in the struct, thus these missing fields get "filled" with unspecified kernel memory, as it is allocated on the stack. This CVE security bulletin actually comprises six different vulnerable network protocol implementations, all following the same leakage pattern, probably as result of copy&paste programming. We will only present the affected code of the AppleTalk implementation – the same kind of analysis applies to all six vulnerabilities.

In this case the structure returned to the user, thus the observable, is shown in Program 6. The leaking function is `atalk_getname` in `net/appletalk/ddp.c`

```
struct sockaddr_at {
    u_char sat_len, sat_family, sat_port;
    struct at_addr      sat_addr;
    union {
        struct netrange r_netrange;
        char            r_zero[8];
    } sat_range;
};
#define sat_zero sat_range.r_zero
```

Program 6: Complex observation struct leads to leak from `sat_zero`.

is shown in Program 7.

In that function, the structure `sat` gets filled with values provided by the kernel, at the end the whole structure is copied via `memcpy` to the address of the `uaddr` pointer, which is, indirectly via the syscall `getsockname`, copied back to user land. However, the field `sat.sat_zero` has not been initialised,

```
int atalk_getname(struct socket *sock,
      struct sockaddr *uaddr, int *uaddr_len, int peer) {
        struct sockaddr_at sat;

        // Official Patch. Comment out to trigger leak
        //memset(&sat.sat_zero, 0, sizeof(sat.sat_zero));
        ⋮ // sat structure gets filled
        memcpy(uaddr, &sat, sizeof(sat));
        return 0;
}
```

Program 7: Function introducing the leak for CVE-2009-3002.

thus a number of bytes of kernel memory are not overwritten and get copied back to the user.

The secret is implicitly modelled by allocating the `sat` structure with non-deterministic values; observations are also of type `sockaddr_at`. The driver uses as parameter `eq_t` the library function `memcmp` to compare memories.

Running the model checker on this driver for a 6 bit policy generated a counterexample within 1 hour and 39 minutes. Once the official patch was applied which sets the `sat` structure to 0 with `memset`, our driver successfully verified the policy in about the same time with unwinding assertions, thus it proved that the patch stops the leak.

tcf_fill_node. This information leak occurs in the netlink subsystem of the kernel. In Program 8, the function `tcf_fill_node` prepares a `struct tcmsg` to be sent back to the user. However, the programmer made a typing mistake and assigned the field `tcm__pad1` twice instead of assigning `tcm__pad2` the second time.

This leaks kernel memory from `tcm__pad2` back to user space. Here, we again modelled kernel memory implicitly by the memory allocated for `tcm` through the function `NLMSG_DATA`, which initialised the fields of the struct with nondeterministic values. The observation is the filled out variable `tcm`, the low user input is a simple integer variable not mentioned here for clarity.

The official patch which was applied to fix the leak is simply changing the last line of Program 8 to `tcm->tcm__pad2=0`. We were again able to prove

```
struct tcmsg *tcm;
...
nlh=NLMSG_NEW(skb, pid, seq, event, sizeof(*tcm), flags);
tcm=NLMSG_DATA(nlh);
tcm->tcm_family = AF_UNSPEC;
tcm->tcm__pad1 = 0;
tcm->tcm__pad1 = 0; // typo, should be tcm__pad2 instead.
```

Program 8: Function excerpt introducing the leak for CVE-2009-3612.

that this patch successfully fixes the security hole and with out it the program violates a leakage policy of 6 bits.

Without the patch, a counterexample is found within 3 minutes and 34 seconds; with the patch, the program is verified within the same time.

sigaltstack. The leakage for this vulnerability is intricate and only manifests itself on 64-bit processors. On such a system, the struct `stack_t`, as shown in Program 9, will be padded to a multiple of 8 bytes because on 64-bit systems `void*` and `size_t` are both 8 bytes (instead of 4 bytes for 32-bit systems), while an integer type remains 4 bytes. Thus, the size of `stack_t` is padded to 24 bytes, while on a 32-bit system it remains unpadded at 12 bytes.

```
typedef struct sigaltstack {
        void __user *ss_sp;
        int ss_flags; // 4 bytes padding on 64-bit
        size_t ss_size;
} stack_t;
```

Program 9: Structure with padding depending on architecture.

The syscall `do_sigaltstack` in `kernel/signal.c` copies such a structure back to userland via the copy function `copy_to_user`, however it does not clear the padding bytes, thus those are leaked to the user on a 64-bit system. In the function visible in Program 10, the high input is the structure `oss` and the low output is the argument `uoss`.

CBMC supports modelling of 64-bit widths however that is not enough to automatically measure the padding bytes. This is because the `sizeof` operator in CBMC returns only the sum of all sizes without eventual bit alignments.

```
int do_sigaltstack (const stack_t __user *uss,
                stack_t __user *uoss, unsigned long sp) {
     stack_t oss;
      ... // oss fields get filled
      if (copy_to_user(uoss, &oss, sizeof(oss)))
             goto out; ....
```

Program 10: Leakage through copying whole structures including padding.

This is solved in our approach by providing a model of the `copy_to_user` function, just like e.g. an implementation of `memcpy` is provided, which checks if the length parameter is aligned according to the architecture (4 bytes for 32 and 8 bytes for 64). If there are padding alignments then these will be chosen to be filled with nondeterministic integer values modulo the number of padding bytes.

In Program 10, this would translate to the following: `sizeof(oss)` counts 20 bytes as the size of the structure. However, this does not account for the padding bytes, and our `copy_to_user` model does the following calculation:

```
pad = ALIGN - (sizeof(oss) % ALIGN);
if(pad == ALIGN)
    padding = 0;
else
    padding = ((unsigned int) nondet_int()) % (1 << (pad*8))
```

where `ALIGN` is chosen to be 4 or 8 depending on the architecture used. In a 64-bit system, this translates to $8 - (20\%8) = 4$ bytes for `pad` which are represented by the `padding` variable.

With this setup, we were able to verify that on a 32-bit system the Program 10 does not leak anything, while on a 64-bit system this violates a policy of e.g. 7 bits. A counterexample was found within 49 minutes and 50 seconds. We were also able to prove that the official patch removes the padding leak. The patch in this case was not to copy the whole struct but copying the three struct members separately through the function `__put_user`, where the padding does not come into play.

cpuset. The crucial part of this vulnerability has already been discussed

in Section 6.3.1. Our analysis finds the right low input which triggers the integer underflow. The actual code however does not simply return the secret as shown in the section mentioned above, but it copies `nbytes` number of bytes from a buffer `ctr->buf` at offset `*ppos`. Because of the underflow, `nbytes` and

```
if (*ppos + nbytes > ctr->bufsz)
      nbytes = ctr->bufsz - *ppos;
if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
     return -EFAULT;
```

`*ppos` access memory way out of the actual buffer and thus disclose kernel memory. However our analysis of this vulnerability requires at the moment too much manual intervention to model memory access outside of the allowed bound (i.e. `ctr->buf + *ppos`).

One elegant way of addressing this problem would be by modifying CBMC itself; CBMC could for example return nondeterministic values for such out-of-bound memory accesses which would implicitly model the access to confidential data.

6.4.2 Authentication checks

We analysed parts of the authentication routines of the secure remote password suite (SRP) and the Unix password authentication of Cyrus' Internet Message Support Protocol daemon (IMSPD).

SRP. To demonstrate that confidential variables and observations can be used flexibly, we checked that there is no leakage in the password request function in `libsrp/t_getpass.c`.

The confidential input is the password entered by the user when being prompted at the login; the observations are the *echos* of the terminal of typed characters. Whether the terminal echos the typed characters or not depends on which mode the console is in. The environment modelling the console and its modes had to be provided to check this program.

In Program 11, the function `t_getpass` first gets the current mode of the console by the function `GetConsoleMode`; then it sets a new console mode by

```
_TYPE( int ) t_getpass (char* buf, unsigned maxlen,
                                    const char* prompt) {
    DWORD  mode;

    GetConsoleMode( handle, &mode );
    SetConsoleMode( handle, mode & ~ENABLE_ECHO_INPUT );
    if(fputs(prompt, stdout) == EOF ||
        fgets(buf, maxlen, stdin) == NULL) {
        SetConsoleMode(handle,mode);
        return -1;
    } ....
```

Program 11: Side-effect of `mode` decides on echo output of `fgets`

inverting the bit
`ENABLE_ECHO_INPUT` in the mode through the function
`SetConsoleMode` which clearly disables the echo of input read from standard input. The function `GetConsoleMode` is modelled by nondeterministically setting the mode to any integer value, the function `SetConsoleMode` sets a global mode variable to its second argument. The function `fgets`, which reads a number of bytes from `stdin`, is modelled to return its first argument `buf` completely if the mode is set to echo the input and return a constant value otherwise.

With this setup CBMC proves through our driver that starting from any initial `mode`, the program will always end up with $\log_2(|\simeq_P|) = 0$, i.e. that there is no leakage. We can also successfully check that if the line which disables the echo is removed then the policy is violated.

IMSPD. The function checked in this package is `login_plaintext` in `imsp/login_unix.c`, as shown in Program 12.
The program first tries to receive the stored password context of a user using the function `getpwnam`. If successful, it will compare the stored with the entered password using `strcmp`. If this fails it will set the string `reply` to "wrong password". If authentication is successful it returns 0.

Clearly, this function has three distinguishable observables: (1) it returns 1 (2) it returns 1 and sets `*reply` (3) it returns 0. We modelled the three parameters to the function as low user input and the stored password as con-

```
int login_plaintext(char *user, char* pass,
                          char** reply) {
    ...
    struct passwd* pwd = getpwnam(user);
    if (!pwd) return 1;
    if (strcmp(pwd->pw_passwd,
               crypt(pass, pwd->pw_passwd)) != 0) {
        *reply = "wrong password";
        return 1;
    }
    return 0;
```

Program 12: Login function of IMSPD.

fidential variable. With this setup, we were able to verify, within 9 seconds, that this program conforms to a policy which only allows 3 distinctions on the confidential variable.

6.5 Review of technique

This chapter presented a way to check if an ANSI-C program conforms to a quantitative policy. The checking of a policy is effectively the checking of a k-safety problem where the outcome describes a bound on the worst case leakage (channel capacity).

The biggest advantage of just checking quantitative information flow instead of inferring it precisely is that the analysis is independent of the secret size. The secret is modelled by nondeterministic choice functions which could be seen as pool of values within the range of the secret. This pool only needs to be large enough to draw as many values from it as necessary to falsify a policy – the real size of the secret is unimportant. This made it possible to run the analysis on data structures which are too large for enumeration or on structures which represent unbounded memory.

A disadvantage of this approach is that it needs manual intervention and modelling effort for each code base under analysis. Writing functions and library calls to model the environment is unavoidable. However, other manual work involved like writing allocation and nondeterministic functions for confi-

dential data structures could probably be avoided by extending CBMC.

6.5.1 Improvements

Extending CBMC

There are two parts in the whole process which need manual work: one is the environment generation, the other one is allocating the confidential structures with nondeterministic values. The latter one could probably be completely automated by extending CBMC or by a preprocessing step. The working of the allocation functions are completely mechanical: take all types in the structure and assign them values from the respective `nondet_*` functions.

This would further simplify the amount of effort necessary to check a certain module for leakage.

Driver structure

To increase scalability, the function calls before the `assume` statement could be replaced by an abstraction. This is easily possible because of the assume-guarantee structure of the driver. For example, a simplified driver with $N = 2$ could be written as

```
o1 = f_octagon();
o2 = f_octagon();
assume(o1 != o2);

o3 = f();
assert(o3 == o1 || o3 == o2);
```

where `f_octagon` is a function returning a concrete value chosen nondeterministically from an octagonal abstract domain generated by abstractly interpreting `f`. This would drastically reduce the size and complexity of the boolean formula generated by CBMC without compromising soundness. However, this will introduce spurious violations of a policy which would need refinement.

Chapter 7

Related Work

7.1 QIF tools

7.1.1 Model checking & constraint solving

Recently, Backes, Köpf, and Rybalchenko published an elegant method to calculate and quantify an equivalence relation given a C-like program [4].

Two algorithms are described to discover and quantify the required equivalence relation. The procedure *Disco* starts with an equivalence relation equivalent to the $\perp$ element in the lattice of information, and iteratively discovers and refines the relation by discovering pairs of execution paths which *do* lead to a distinction in the outputs. The corresponding high inputs of those two paths are then split in two different equivalence classes. This process is repeated until no more counter examples are discovered. The procedure *Quant* calculates the sizes of equivalence classes generated by the output of the previous procedure. The result can be normalised to a probability distribution and any probabilistic measure can be applied on it.

Disco is implemented by turning the information flow checking into a reachability problem, as shown by [59]. The program P is self-composed by creating a copy of the code P' with disjoint variable sets (indicated by the primes) and an added low inequality check at the end of the newly created program, where R is the relation to be refined:

```
if(l = l' && (h,h') in R)
   P(h,l)
   P'(h',l')
   if(l != l')
      error
```

If the error state is reachable then that indicates that there exist two paths of the program P with related low and high inputs which produce distinguishable outputs l and l'. This is a violation of the noninterference property and thus a leak of information.

The model checker ARMC is applied to this reachability problem which will output a path to the error label, if reachable. Beside the path, the model checker also returns a formula in linear arithmetic which characterises all initial states from which the error state is reachable. Out of this formula, the two previously related secrets h and h' can be extracted which are then split in two different equivalence classes.

Given the formula from the last step, QUANT calculates the number and sizes of those equivalence classes using a combination of the Omega calculator and the Lattice Point Enumeration Tool. Omega calculates for each equivalence class a linear arithmetic proposition in disjunctive normal form. The enumeration tool then solves these system of linear inequalities for each class, which results in counting the number of elements in the equivalence class.

The so generated equivalence class can then be applied to various entropy formulas. The paper shows as example, among others, a sum query of three secrets. The precision and scalability of the tool entirely depends on the choice of underlying tools. The runtime depends on the number of execution paths of the program under analysis and number of variables involved.

7.1.2 Interval abstraction

Mu and Clark use probabilistic semantics in an abstract interpretation framework to build an automatic analyser [50]. The authors borrow Kozen's semantics for probabilistic programs which interprets programs as a partial measurable functions on a measurable space; these semantics can be seen as a way

to map an input probability distribution to an output probability distribution through the execution of the program under analysis. The entropy measure used is Shannon's entropy was extended to work on "incomplete" random variables, where the entropy is normalised to the coverage of the probability distribution.

To make their analysis tractable, they employ abstract interpretation as their abstraction technique. The interval abstract domain is used to partition the concrete measure space into blocks. Additionally, Monniaux's abstract probabilistic semantics are used to replace the previous concrete semantics. The abstraction overestimates the leakage through *uniformalization*, which provides safe upper bounds on the leakage. The concrete space X is abstracted to a set of interval-based partitions for each program variable, together with a weighting factor α_i, which is the sum of the probabilities of the interval value-range.

The abstract domain is described by a Galois connection $X\langle\alpha,\gamma\rangle X^{\#}$, where the measure space X is abstracted by $X^{\#}$. The abstraction function α is a map from X to sets of interval-based partitions $X^{\#} = \{\langle\alpha_i, [E_i]\rangle\}_{0<i\leq n}$, with n the number of partitions, α_i the weight, $[E_i]$ be the interval based partition of X. The concretisation function γ maps $X^{\#}$ to $\bigcup\{x|x \in [E_i/\eta]\}$, where E_i is a block of the abstract object $X^{\#}$ and η is a sub-partition on each block under uniform distribution.

The corresponding abstract semantics function $[\![\cdot]\!]^{\#}$ transforms the abstract spaces described by $X^{\#}$. The description of the abstract operations are skipped and instead we explain the effects of a conditional on the abstract domain as an example: the test splits the abstract space into two parts according to the two outcomes of the test. The if statement returns the sum of the statements in the two branches where the new intervals of variable values are calculated using interval arithmetic.

Taking an example from the authors paper

```
if(x==0) then y=0 else y=1
```

The analysis starts off with an initial probability distribution for x as 3 bit

variable

$$\begin{pmatrix} 0 \to 0.1 & 1 \to 0.1 & 2 \to 0.1 & 3 \to 0.1 \\ 4 \to 0.2 & 5 \to 0.2 & 6 \to 0.1 & 7 \to 0.1 \end{pmatrix}$$

and y as low security variable under any distribution. An initial partition is then for example

$$\begin{array}{ll} E_1\langle [0,3]_x, [0,7]_y \rangle & \alpha_1 = 0.4 \\ E_2\langle [4,7]_x, [0,7]_y \rangle & \alpha_2 = 0.6 \end{array}$$

After applying the abstract operation for the if statement the abstract domain is transformed to

$$\begin{array}{ll} E'_1\langle [0,3]_x, [0,1]_y \rangle & \alpha'_1 = 0.4 \\ E'_2\langle [4,7]_x, [1,1]_y \rangle & \alpha'_2 = 0.6 \end{array}$$

The working of the interval-arithmetic is clearly visible in the restriction of the intervals for the y variable.

Leakage of loops is done in a standard fashion using least fixpoints and interval widening, additionally the weight on each abstract element is the maximum between the current and previous iteration.

Once the final abstract space has been calculated, the uniformalization transformation guarantees a conservative leakage analysis; this is a process to maximise the entropy for a given abstract space. Again, let us explain this using an example. After some computation, we are given the abstract object on the left; the uniformalized probability distribution on variable l is given on the right.

$$\begin{array}{ll} E'_1\langle [2,2]_y \rangle & \alpha'_1 = 0.3 \\ E'_2\langle [3,3]_y \rangle & \alpha'_2 = 0.4\ , \\ E'_3\langle [4,6]_y \rangle & \alpha'_2 = 0.3 \end{array} \qquad \begin{pmatrix} 2 & \to & 0.3/1 \\ 3 & \to & 0.4/1 \\ 4 & \to & 0.3/3 \\ 5 & \to & 0.3/3 \\ 6 & \to & 0.3/3 \end{pmatrix}$$

Thus, the weight of each interval is divided by the size of the interval. Finally, the leakage upper bound of this space is calculated as $H(0.3, 0.4, 0.3) + 0.3 * \log(3)$.

This work is the first description of an abstract domain for quantitative information flow. The precision of the analysis is clearly limited by the precision of the interval arithmetic and uniformalization. The scalability of the analysis has not been discussed by the authors.

7.1.3 Network flow capacity & dynamic analysis

McCamant and Ernst [45, 46] present multiple techniques to analyse information leakage in large C, C++, and Objective C programs. The authors released as tool FLOWCHECK which computes leakage as the maximum flow between inputs and outputs using a combination of network flow capacities and dynamic binary analysis. The basic tool to model programs is a network flow graph which represents the execution of a program in a form similar to a circuit. Edges represent values, and have their secret bitwidths (how many secret bits that can be transferred by that edge) as capacity. Nodes represent basic operations on those values. Implicit flows, generated by branches and pointers, are integrated in this model through what the authors call *enclosed region*. Such a region is an annotation in the code which abstracts away a block of the program into a single node with given inputs and outputs. Those annotations can be partly inferred and some need manual editing of the source code. The calculation of leakage then reduces to checking the maximum flow in this network.

The tool was applied to a number of large programs, such as the OpenSSH client, and X server, and the Imagemagick tool. Their approach is interesting as it is not based on reachability, like most other analyses, and because it reaches a high level of scalability through making use of dynamic instrumentation tools, such as Valgrind.

7.1.4 Sampling channel capacity

Chatzikokolakis, Chothia, and Guha [10] automatically quantify information flows using a statistical approach which treats probabilistic systems as blackboxes; thus it is not a language-based analysis. Such a blackbox-approach has the advantage that any system, even applications which the attacker has no direct access to, could be analysed for leakage as long as repeated inputs and outputs of the system can be observed.

Once inputs and outputs have been defined, the system is sampled on a number of inputs. The aim is to build a probability transition matrix which reflects the true conditional probabilities of the outputs given the inputs. Given

this estimated matrix, the capacity, i.e. mutual information, of the system is calculated.

The capacity is automatically calculated using an iterative algorithm, which is a more efficient and more automated approach than previously described in the quantitative information flow literature [39].

However, the leakage quantity calculated in this way is the result of an approximation algorithm on a sampled matrix, thus it needs further statistical analysis to gain confidence in the numbers. The authors provide a way to calculate bounds on the true capacity given the estimated result.

The implemented tool is applied to a Mixminion remailer which provides anonymous email forwarding. The goal of the analysis was to check how well the remailer retains the anonymity of the sender. For example one of the tests ran was to check if the ordering of packets entering and leaving the node leaks information about the identity of the sender. The authors were able to show that the estimate is within the bounds of zero leakage, i.e. no loss of anonymity.

7.2 QIF theory

This section cites papers which introduced influential ideas used in this thesis.

Lattice of information. Landauer and Redmond wrote the original lattice of information paper [35]. Knuth's paper gave me the idea to look into lattice valuations in relation to information theory [32]. Nakamura provided the proofs and crucial insights connecting semivaluations and entropy [51]. The papers by Schellekens [53] and Simovici [56] also described similar ideas on entropy and the lattice of partitions. Zdancewic and Myers used the lattice of information in an information flow setting where it was used as model to describe what is distinguishable for an observer [68].

Self-composition. Barthe et al. first described secure information flow by self-composition [5]. In the same paper, the authors also describe non-interference in terms of Computation Tree Logic which could be in itself an interesting research direction.

Terauchi and Aiken extended this work and introduced the idea of secure

information flow being a 2-safety problem [59]. Both papers mention how self-composition lends itself to using model checkers for verifying secure information flows in programs.

K-safety. The concept of k-safety has been introduced by Clarkson and Schneider's Hyperproperties [16] and the two very influential papers on quantitative information flow by Yasuoka and Terauchi [65, 66]. The latter two papers study the verification hardness of exactly quantifying and bounding information leakage. The second paper proves a theorem which shows that for a fixed number of distinctions, calculating a bound on the channel capacity is a k-safety problem. While we independently found and described the same result, the theoretical presentation is clearer in Yasuoka and Terauchi's paper and deserves special mention.

Counterexamples against non-interference. Unno et al. [62] describe an analysis which uses model checking of self-composed programs to find violations of non-interference. The authors also described optimisations to the naive self-composition. Finding non-interference violations via their technique is definitely the first step towards quantifying information leakage. I am also grateful to Hiroshi Unno who provided me with the full source code of their implementation.

Chapter 8

Conclusion

This thesis described the lattice of information and its properties as the underlying structure of quantitative information flow. Furthermore, we presented three techniques to infer and check leakage in programs.

The first approach exhaustively enumerates all possible confidential values in order to calculate the leakage. It is a precise method to calculate the leakage of loops, implementing the theory of [38]. However, state explosion issues make this approach unusable in practice.

In chapter 5, a static analysis translated the program to be analysed to a bit-vector language which in turn gets solved by a SAT solver. This approach splits the task of calculating the precise leakage into finding the number of equivalence classes and then model counting their sizes. It is a more automated analysis, however the scalability issues are not adequately addressed yet.

Finally, the last tool developed in the course of this thesis, describes a way to check bounds on how much information leaked in programs. A property is defined which, if a counterexample exists, implies a lower bound on the channel capacity. Encoding the confidential space as nondeterministic pool of values allows reasoning independent of the size of the secret. The tool has been applied to describe bounds on existing leakage vulnerabilities in the Linux Kernel and also verified that some patches reduce the leakage.

To further improve the quantitative analysis of information leakage one has to find an appropriate notion of *abstraction*. It is a very big challenge however

to find an abstraction appropriate for a quantitative analysis. Abstraction is about reducing the number of details to make the analysis more tractable; quantification, on the other hand, becomes more precise the more details there are in the system. There is a tradeoff to be made between precision and tractability.

Chapter 6 introduced two forms of abstraction. Firstly, it does not precisely quantify the leakage anymore but solely defines a property whose violation or verification implies bounds on the maximal possible leakage. Secondly, it represents confidential values in an abstract way by defining them as a nondeterministic source. This allows a program analysis tool to draw values from this pool "on demand" until the property is violated or verified.

Further improving and understanding abstractions in the context of a quantitative analysis is definitely the right way to go for future research.

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman: Universality of data retrieval languages. POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.

[2] Domagoj Babić and Frank Hutter: Spear Theorem Prover. Proc. of the SAT 2008 Race, 2008.

[3] Michael Backes and Boris Köpf: Formally Bounding the Side-Channel Leakage in Unknown-Message Attacks. Proc. 13th European Symposium on Research in Computer Security (ESORICS)

[4] Michael Backes and Boris Köpf and Andrey Rybalchenko: Automatic Discovery and Quantification of Information Leaks. Proc. 30th IEEE Symposium on Security and Privacy (S& P '09)

[5] Barthe, Gilles and D'Argenio, Pedro R. and Rezk, Tamara:Secure Information Flow by Self-Composition. CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations.

[6] R. Bayardo, R. Schrag: Using CSP look-back techniques to solve real-world SAT instances.In Proc. of AAAI-97. pp. 203-208. AAAI Press/The MIT Press, 1997.

[7] Birkhoff, G.: Lattice theory. Amer. Math. Soc. Colloq. Publ. 25 (1948).

[8] Paul Bratley and Bennett L. Fox and Harald Niederreiter: Implementation and tests of low-discrepancy sequences. ACM Trans. Model. Comput. Simul., Volume 2, Number 3 / 1992.

[9] Pankaj Chauhan and Edmund M. Clarke and Daniel Kroening: Using SAT based Image Computation for Reachability. Carnegie Mellon University, Technical Report CMU-CS-03-151, 2003.

[10] Konstantinos Chatzikokolakis and Tom Chothia and Apratim Guha: Statistical Measurement of Information Leakage. TACAS, 2010, 390-404

[11] Konstantinos Chatzikokolakis and Keye Martin: A monotonicity principle for information theory. Elsevier Science B.V., 2008

[12] David Clark, Sebastian Hunt, Pasquale Malacaria: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security, Volume 15, Number 3 / 2007.

[13] David Clark, Sebastian Hunt, and Pasquale Malacaria: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language, 18(2):181-199, 2005.

[14] David Clark, Sebastian Hunt, Pasquale Malacaria: Quantitative Analysis of the leakage of confidential data. Electronic Notes in Theoretical Computer Science 59, 2002

[15] Michael R. Clarkson, Andrew C. Myers, Fred B. Schneider: Belief in Information Flow. CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations, 2005:31-45

[16] Michael R. Clarkson and Fred B. Schneider: Hyperproperties. Journal of Computer Security, Volume 18, Number 6, 2010:1157-1210

[17] Clarke, Edmund, and Kroening, Daniel, and Lerda, Flavio: A Tool for Checking ANSI-C Programs. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Springer, 168–176, Volume 2988

[18] Patrick Cousot, Radhia Cousot: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. ACM symposium on Principles of programming languages, 1977.

[19] T.Cover, J. Thomas. Elements of Information Theory. Wiley

[20] Adnan Darwiche and Pierre Marquis: A Knowledge Compilation Map. Journal of Artificial Intelligence Research, 2002, 229–264 Volume 17.

[21] D. E. R. Denning: Cyptography and Data Security. Addison-Wesley, 1982.

[22] Dorothy E. Denning, Jan Schlörer, A fast procedure for finding a tracker in a statistical database, ACM Transactions on Database Systems, Volume 5, Issue 1 (March 1980), 88-102

[23] Denning, Dorothy E. and Denning, Peter J.: Certification of programs for secure information flow. Commun. ACM, 20, 7, July, 1977, 504–513

[24] David Dobkin and Anita K. Jones and Richard J. Lipton: Secure databases: Protection against user influence. ACM Transactions on Database Systems, 1979, Volume 4, 97–106.

[25] R. Giacobazzi: Domain Theory in Abstract Interpretation. http://www.cs.ucy.ac.cy/compulog/newpage114.htm

[26] J A Goguen, J Meseguer: Security policies and security models. In Proceedings of the 1982 IEEE Computer Society Symposium on Security and Privacy

[27] James W Gray III: Toward a mathematical foundation for information flow security. Proc. 1991 IEEE Symposium on Security and Privacy. Oakland, California, May 1991.

[28] Jonathan Heusser and Pasquale Malacaria: Quantifying Loop Leakage using a Lattice of Partitions. In Proceedings of Workshop on Quantitative Analysis of Software Berkeley Technical Report No. UCB/EECS-2009-93

[29] Jonathan Heusser and Pasquale Malacaria: Applied Quantitative Information Flow and Statistical Databases. In Proceedings of Formal Aspects in Security and Trust (FAST) 2009, 96–110. Springer

[30] Jonathan Heusser and Pasquale Malacaria: Quantifying Information Leaks in Software. In Proceedings of Annual Computer Security Applications (ACSAC) 2010, ACM.

[31] E.T. Jaynes: Information Theory and Statistical Mechanics. Statistical Physics, 181, 1963.

[32] Kevin Knuth: Valuations on Lattices and their Application to Information Theory. IEEE International Conference on Fuzzy Systems, 2006, 217-224

[33] Boris Köpf and David Basin: An information-theoretic model for adaptive side-channel attacks. CCS '07: Proceedings of the 14th ACM conference on Computer and communications security, 2007, 286-296

[34] Lamport, L.: Proving the Correctness of Multiprocess Programs IEEE Trans. Softw. Eng., Volume 3, Issue 2, 1977, 125-143

[35] Landauer, J., and Redmond, T.: A Lattice of Information. In Proc. of the IEEE Computer Security Foundations Workshop. IEEE Computer Society Press, 1993.

[36] Leino, K. Rustan M. and Joshi, Rajeev: A Semantic Approach to Secure Information Flow. Proceedings of the Mathematics of Program Construction (MPC), 1998, 254–271

[37] Leyton-Brown, Kevin and Nudelman, Eugene and Shoham, Yoav: Empirical hardness models: Methodology and a case study on combinatorial auctions. J. ACM, 56, 4, 2009, 0004-5411, 1–52, ACM, New York, NY, USA.

[38] Pasquale Malacaria: Assessing security threats of looping constructs. Proc. ACM Symposium on Principles of Programming Language, 2007.

[39] Pasquale Malacaria, Han Chen:Lagrange Multipliers and Maximum Information Leakage in Different Observational Models. ACM SIGPLAN Third Workshop on Programming Languages and Analysis for Security. June, 2008.

[40] Pasquale Malacaria: Risk Assessment of Security Threats for Looping Constructs, to appear in the Journal Of Computer Security, 2009.

[41] Malacaria, Pasquale and Heusser, Jonathan: Information Theory and Security: Quantitative Information Flow. Formal Methods for Quantitative Aspects of Programming Languages. Lecture Notes in Computer Science Springer Berlin / Heidelberg, 87-134, 6154

[42] Keye Martin and Ira S. Moskowitz and Gerard Allwein: Algebraic Information Theory For Binary Channels. Electr. Notes Theor. Comput. Sci., 289-306, 2006

[43] John McLean: A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. Proceedings of the 1994 IEEE Symposium on Security and Privacy, 1994, 79–

[44] John McLean: Security models and information flow. Proc. 1990 IEEE Symposium on Security and Privacy. Oakland, California, May 1990.

[45] Stephen McCamant and Michael D. Ernst: A Simulation-based Proof Technique for Dynamic Information Flow. PLAS 2007: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, 2007

[46] Stephen Andrew McCamant: Quantitative Information-Flow Tracking for Real Systems. MIT Department of Electrical Engineering and Computer Science, Ph.D., Cambridge, MA, 2008.

[47] McKenzie, B. J., R. Harries, and T. Bell: Selecting a Hashing Algorithm, Software Practice and Experience 20,2 (Feb 1990), p209.

[48] Jonathan Millen: Covert channel capacity. Proc. 1987 IEEE Symposium on Research in Security and Privacy.

[49] Robin Milner: Is Computing an Experimental Science? *LFCS report ECS-LFCS-86-1, 1986.*

[50] Chunyan Mu and David Clark: Quantitative Analysis of Secure Information Flow via Probabilistic Semantics. ARES, 2009, 49-57

[51] Y. Nakamura: Entropy and Semivaluations on Semilattices. Kodai Math. Sem. Rep 22 (1970), 443 468

[52] A. Pnueli: In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, Logics and Models of Concurrent Systems, volume 13 of NATO ASI series F. Springer-Verlag, 1984.

[53] M. P. Schellekens: A characterization of partial metrizability: domains are quantifiable. Theor. Comput. Sci., 305, 1-3, 2003

[54] Robert Sedgewick: Algorithms in C++ *Addison-Wesley Longman Publishing Co., Inc.*, 1992, Boston, MA, USA

[55] D. Senato, H. Crapo: Algebraic Combinatorics and Computer Science: A Tribute to Gian-Carlo Rota. Springer-Verlag New York, Inc., 8847000785, 2001

[56] Dan Simovici: Metric-Entropy Pairs on Lattices. Journal of Universal Computer Science, vol. 13, no. 11 (2007), 1767-1778

[57] Geoffrey Smith: On the Foundations of Quantitative Information Flow. FOSSACS, 2009, 288-302.

[58] C. E. Shannon and W. Weaver: A Mathematical Theory of Communication. Urbana, IL: Univ. of Illinois press, 1963.

[59] T. Terauchi and A. Aiken: Secure information flow as a safety problem. In SAS, volume 3672 of LNCS, pages 352–367, 2005.

[60] Tong, Matthew C. F.: General Hashing.

[61] E.A.Unger, L.Harn and V.Kumar: Entropy as a Measure of Database Information, Proceedings of Sixth ACSAC, IEEE 1990

[62] Hiroshi Unno, and Naoki Kobayashi and Akinori Yonezawa: Combining type-based analysis and model checking for finding counterexamples against non-interference. PLAS, pages 17–26, 2006

[63] Volpano, Dennis and Irvine, Cynthia and Smith, Geoffrey: A sound type system for secure flow analysis. J. Comput. Secur., 4, 2-3, January, 1996, 167–187

[64] Glynn Winskel: The Formal Semantics of Programming Languages. The MIT Press 1993.

[65] Hirotoshi Yasuoka and Tachio Terauchi: Quantitative information flow - verification hardness and possibilities. In Proceedings CSF 2010.

[66] Hirotoshi Yasuoka and Tachio Terauchi: On Bounding Problems of Quantitative Information Flow. In Proceedings of ESORICS 2010.

[67] Raymond W. Yeung.: A new outlook on Shannon's information measures. IEEE Transactions on Information Theory, vol. 37, no. 3 (1991), 466-474

[68] Steve Zdancewic and Andrew C. Myers: Robust Declassification. In Proc. IEEE Computer Security Foundations Workshop, 2001, 15-23