


3-27-2014

# Two-Bit Pattern Analysis For Quantitative Information Flow

Ziyuan Meng  
zmeng001@fiu.edu

**DOI:** 10.25148/etd.FI14040860

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

 Part of the [Information Security Commons](#), and the [Programming Languages and Compilers Commons](#)

---

## Recommended Citation

Meng, Ziyuan, "Two-Bit Pattern Analysis For Quantitative Information Flow" (2014). *FIU Electronic Theses and Dissertations*. 1326.  
<https://digitalcommons.fiu.edu/etd/1326>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact [dcc@fiu.edu](mailto:dcc@fiu.edu).

FLORIDA INTERNATIONAL UNIVERSITY  
Miami, Florida

TWO-BIT PATTERN ANALYSIS FOR QUANTITATIVE INFORMATION  
FLOW

A dissertation submitted in partial fulfillment of the  
requirements for the degree of  
DOCTOR OF PHILOSOPHY  
in  
COMPUTER SCIENCE  
by  
Ziyuan Meng

2014

To: Dean Amir Mirmiran  
College of Engineering and Computing

This dissertation, written by Ziyuan Meng, and entitled Two-Bit Pattern Analysis for Quantitative Information Flow, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

---

Jinpeng Wei

---

Bogdan Carbunar

---

Jeffrey Fan

---

Raju Rangaswami

---

Geoffrey Smith, Major Professor

Date of Defense: March 27, 2014

The dissertation of Ziyuan Meng is approved.

---

Dean Amir Mirmiran  
College of Engineering and Computing

---

Dean Lakshmi N. Reddi  
University Graduate School

Florida International University, 2014

© Copyright 2014 by Ziyuan Meng

All rights reserved.

DEDICATION

To my parents.

## ACKNOWLEDGMENTS

I would like to dedicate this thesis to my family and friends for their love and support, in particular, to my mother Yuanlan Hu and father Zhongxiang Meng, who have made my accomplishments possible. I would also like to recognize my friends in the Florida International University community. I truly appreciate the inspiration and encouragement you have given me during my time in the doctoral program.

Special thanks to my friend and advisor Geoffrey Smith for his guidance, confidence in my abilities and balanced perspective on life. I wish to thank other members of my dissertation committee: Jinpeng Wei, Raju Rangaswami, Jeffrey Fan and Bogdan Carbunar for generously offering their time, support, guidance and good will throughout the preparation and review of this document. I also want to thank Alexander Tepper for his help in editing this dissertation.

ABSTRACT OF THE DISSERTATION  
TWO-BIT PATTERN ANALYSIS FOR QUANTITATIVE INFORMATION  
FLOW

by

Ziyuan Meng

Florida International University, 2014

Miami, Florida

Professor Geoffrey Smith, Major Professor

Protecting confidential information from improper disclosure is a fundamental security goal. While encryption and access control are important tools for ensuring confidentiality, they cannot prevent an authorized system from leaking confidential information to its publicly observable outputs, whether inadvertently or maliciously. Hence, secure information flow aims to provide end-to-end control of information flow. Unfortunately, the traditionally-adopted policy of noninterference, which forbids all improper leakage, is often too restrictive. Theories of quantitative information flow address this issue by quantifying the amount of confidential information leaked by a system, with the goal of showing that it is intuitively “small” enough to be tolerated. Given such a theory, it is crucial to develop automated techniques for calculating the leakage in a system.

This dissertation is concerned with program analysis for calculating the maximum leakage, or capacity, of confidential information in the context of deterministic systems and under three proposed entropy measures of information leakage: Shannon entropy leakage, min-entropy leakage, and g-leakage. In this context, it turns out that calculating the maximum leakage of a program reduces to counting the number of possible outputs that it can produce.

The new approach introduced in this dissertation is to determine two-bit patterns, the relationships among pairs of bits in the output; for instance we might determine that two bits must be unequal. By counting the number of solutions to the two-bit patterns, we obtain an upper bound on the number of possible outputs. Hence, the maximum leakage can be bounded. We first describe a straightforward computation of the two-bit patterns using an automated prover. We then show a more efficient implementation that uses an implication graph to represent the two-bit patterns. It efficiently constructs the graph through the use of an automated prover, random executions, STP counterexamples, and deductive closure. The effectiveness of our techniques, both in terms of efficiency and accuracy, is shown through a number of case studies found in recent literature.



## TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION . . . . .	1
1.1 Motivation . . . . .	1
1.2 Contributions . . . . .	2
1.3 Organization . . . . .	7
2. PRELIMINARIES . . . . .	8
2.1 Information Channel . . . . .	8
2.2 Measuring Leakage using Mutual Information . . . . .	9
2.3 Measuring Leakage using Min-entropy . . . . .	11
2.4 Measuring Leakage using Gain Functions and $g$ -leakage . . . . .	14
2.5 Channel Capacity . . . . .	17
3. TWO-BIT PATTERNS . . . . .	20
3.1 Deriving Predicates . . . . .	21
3.2 Discovering One-Bit and Two-Bit Patterns . . . . .	23
3.3 Counting the Number of Solutions . . . . .	28
4. OPTIMIZATION . . . . .	31
4.1 Formal Framework . . . . .	31
4.1.1 Implication Graphs . . . . .	32
4.1.2 Semantic Characterization . . . . .	33
4.2 One-bit Patterns, Random Execution, and STP Counterexamples . . . . .	35
4.3 Two-bit Patterns and Deductive Closure . . . . .	36
4.4 Computing Implication Graphs Efficiently . . . . .	40
4.5 Revisiting the Illustrative Example . . . . .	42
5. EXPERIMENTS . . . . .	43
5.1 Sanity Check . . . . .	43
5.2 Implicit Flow . . . . .	45
5.3 Population Count . . . . .	46
5.4 Mix and Duplicate . . . . .	47
5.5 Masked Copy . . . . .	48
5.6 Binary Search . . . . .	49
5.7 Electronic Purse . . . . .	50
5.8 Sum Query . . . . .	51
5.9 Ten Random Outputs . . . . .	52
5.10 Summary . . . . .	53

6. RELATED WORK . . . . .	58
6.1 Calculating Quantitative Information Flow . . . . .	58
6.2 Binary Implication Graph . . . . .	64
6.3 Unit-Two Variable Per Inequality Constraints . . . . .	67
7. CONCLUSION AND FUTURE WORK . . . . .	69
7.1 Conclusion . . . . .	69
7.2 Future Work . . . . .	69
7.2.1 Abstract Interpretation . . . . .	70
7.3 Low Input . . . . .	73
7.4 The Origin of Undue Information Flow . . . . .	73
BIBLIOGRAPHY . . . . .	77
VITA . . . . .	81

## LIST OF FIGURES

FIGURE	PAGE
1.1 Two-bit patterns for $\{00010, 10001, 00001, 00110, 10101, 00101\}$ . . . . .	4
3.1 Illustrative example program that leaks information from $X$ to $Y$ . . . . .	20
3.2 Translation of illustrative example into STP . . . . .	22
3.3 Algorithm to determine two-bit patterns for $Y_f$ . . . . .	27
4.1 The implication graph for $\{00010, 10001, 00001, 00110, 10101, 00101\}$ . . . . .	33
4.2 A partially-known implication graph . . . . .	37
6.1 The symbolic representation for $\{000, 100, 110\}$ . . . . .	62
6.2 A SQIF state exploration trace for sanity check program . . . . .	63
6.3 An implication graph derived from $E$ (reprinted from [HJB11]) . . . . .	65
6.4 Stamp times in $\text{IG}(E)$ (reprinted from [HJB11]) . . . . .	66
7.1 The stack holds the return address, the arguments, and the local variables for <code>foo</code> . . . . .	75
7.2 The stack with overwritten return address . . . . .	76

# CHAPTER 1

## INTRODUCTION

### 1.1 Motivation

With mobile devices and networking technology becoming more widespread, computing systems process more *confidential* information than ever. Users would like to know whether this information has been *leaked* to other undesirable users. Traditional approaches to information security, such as access control and encryption, cannot guarantee control over the way in which this confidential information is distributed/propagated. For instance, an Android download can claim to respect the system's permission management, yet secretly send sensitive information to the network after it is authorized. Similarly, encryption can ensure that only the endpoints of a communication channel have access to the secret information. However, it cannot stop the receiver from improperly distributing the decrypted data.

The concept of *secure information flow* aims to provide an end-to-end mechanism to control the *flow of information*, and therefore, protect the confidential data. Unfortunately, the traditionally-adopted policy of *non-interference*, which forbids all improper leakage, is often unrealistic in many application scenarios. There are two main reasons for this. First, in some applications, leakage is an intrinsic part of their functionalities. For instance, an ATM machine that rejects an incorrect PIN, thereby reveals that the secret PIN differs from the one that was entered. Similarly, publishing the tally of votes in an election manifests some information about the secret ballots that were cast. Another more subtle reason of leakage is the side channel. For instance, the amount of *time* taken by a cryptographic operation may be observable by an adversary and may inadvertently reveal information about the secret key. As a result, the last decade has seen growing interest in *quantitative*

theories of information flow, which address this problem by quantifying the amount of confidential information leaked by a system, with the goal of showing that it is “small” enough to be tolerated [CHM05, CMS05, KB07, Smi09, AAP10, HSP10].

Given such a theory, it is crucial to develop automatic techniques for calculating or estimating the amount of leakage in a system, in order to verify whether it conforms to a given quantitative flow policy. This is an area that is now seeing a great deal of work, both in the context of deterministic imperative programs [BKR09, KR10, NMS09, HM10] and probabilistic systems [APvRS10, CCG10], and utilizing both model checking and statistical sampling techniques.

To give some quick intuition, assume (as we will throughout this dissertation) that  $X$  and  $Y$  are 32-bit unsigned integers, where  $X$  is the secret input and  $Y$  is the observable output. Consider the following three C programs:

1.  $Y = X;$ ,
2.  $Y = 17;$ ,
3.  $Y = X \& 0x1f;$

Intuitively, it seems clear that the leakage of these three programs should be 32, 0, and 5 bits, respectively. Notice that these quantities are the logarithms (to base 2) of the number of *feasible values* for  $Y$ , which is  $2^{32}$ , 1, and  $2^5$ , respectively.

## 1.2 Contributions

The major contribution in this dissertation is to introduce and explore the use of what we call *two-bit patterns* to calculate *upper bounds* on the maximum amount of leakage in deterministic imperative programs under *Shannon entropy*, *min-entropy*, and the recently proposed *g-entropy*. Min-entropy leakage and g-entropy leakage

are alternatives to the more commonly-used measure based on Shannon entropy and mutual information. In Chapter 2, we review these theories and motivations in the context of security. For now, it suffices to know that the *capacity* (i.e. their maximum leakage over all prior distributions on the secret input) of *deterministic systems*, whether measured by Shannon entropy or min-entropy, is the logarithm of the number of feasible outputs. And this quantity is also an upper bound on the  $g$ -leakage, for any gain function  $g$  [Smi09, BCP09, ACPS12].

Thus, the problem can be reduced to calculate the *number of feasible outputs* that a deterministic program can produce. Our approach to bounding this quantity is to determine *two-bit patterns* among the bits of the feasible outputs. The key idea is to bound the number of feasible outputs by determining *one-bit patterns* that constrain each *individual* bit and *two-bit patterns* that constrain each *pair* of bits in the output. For example, suppose that the program has 6 feasible outputs:

$$\{00010, 10001, 00001, 00110, 10101, 00101\}$$

where we index the 5 bit positions from 4 down to 0. Studying these outputs, we notice that bit 3 is *fixed*—it is 0 in every output, which we express as  $Zero(3)$ . In contrast, bits 4, 2, 1, and 0 can each be 0 or 1, and we refer to them as *Non-fixed*. Notice that it is only the non-fixed bits that give rise to multiple outputs; here the fact that there are 4 non-fixed bits tells us immediately that there can be at most  $2^4 = 16$  feasible outputs.

We can tighten this bound by considering the relationship between each pair of non-fixed bits. For instance, if we examine bits 4 and 0, we see that the possible combinations of values that they can take are  $\{00, 11, 01\}$ , which we express as  $Leg(4, 0)$ . Bits 4 and 2, in contrast, can take all four combinations  $\{00, 10, 01, 11\}$ , which we express as  $Free(4, 2)$ . The complete two-bit patterns for this example are

$$Zero(3) \quad \begin{array}{c} 4 \\ 2 \\ 1 \\ 0 \end{array} \left[ \begin{array}{cccc} & 4 & 2 & 1 & 0 \\ & Eq & Free & Nand & Leq \\ & Free & Eq & Free & Free \\ & Nand & Free & Eq & Neq \\ & Geq & Free & Neq & Eq \end{array} \right]$$

Figure 1.1: Two-bit patterns for  $\{00010, 10001, 00001, 00110, 10101, 00101\}$

shown in Figure 1.1. Two-bit patterns represent constraints that must be satisfied by the bits of each feasible output. If we count the number of solutions to the two-bit patterns, we get an *upper bound* on the number of feasible outputs. In this case, it turns out that there are just 6 solutions, meaning that here our upper bound is exact.

For a small program, the two-bit patterns of its output could be calculated by using STP solver, an automated theorem prover. Consider the C-like programs that take as input a secret value  $X$  and produce an output  $Y$ , where we assume that all variables are 32-bit unsigned integers. The idea of two-bit patterns is to determine, for every *pair*  $(i, j)$  of bit positions,<sup>1</sup> which of the four combinations  $(0, 0)$ ,  $(0, 1)$ ,  $(1, 0)$ , and  $(1, 1)$  are possible values for bits  $i$  and  $j$  of  $Y$ . As an example, consider the following program, adapted from [NMS09]:

```
Y = ((X >> 16) ^ X) & 0xffff;
Y = Y | Y << 16;
```

This program can be translated into the following STP assertions, where the symbol  $Y1$  denotes the intermediate value of variable  $Y$ :

```
X : BITVECTOR(32);
Y1, Y : BITVECTOR(32);
```

---

<sup>1</sup>We number the bits from 0 to 31, right to left.

```

ASSERT(Y1 = BVXOR((X >> 16), X) & 0hex0000ffff);
ASSERT(Y = Y1 | (Y1 << 16));

```

Then the bit-patterns for  $Y$  is determined by making STP *queries*, which ask whether a given property (e.g., “ $Y[3] = Y[19]$ ”) is a logical consequence of the `ASSERT` statements. We use one or two STP queries to determine each one-bit pattern, and then use a decision tree of at most four STP queries to determine the two-bit pattern among each pair of non-fixed bits. In this case, there are  $32 \cdot 31/2 = 496$  two-bit patterns on  $Y$ ; the only interesting patterns are that bits  $i$  and  $i + 16$  must be equal, for  $0 \leq i \leq 15$ . That is, bits  $i$  and  $i + 16$  can be  $(0, 0)$  or  $(1, 1)$ , but not  $(0, 1)$  or  $(1, 0)$ . If we count the number of solutions to the two-bit patterns by using the `SatisfiabilityCount` function of *Mathematica*, we get an *upper bound* on the number of possible values of  $Y$ . Here there are  $2^{16}$  solutions to the two-bit patterns, giving a maximum leakage of at most  $\log 2^{16} = 16$  bits, which is exact in this case.

Notice that two-bit patterns subsume *one-bit* patterns, which classify each bit as 0, 1, or *non-fixed*. For example, if we know that bits  $i$  and  $j$  cannot be  $(0, 0)$  or  $(0, 1)$ , then we know that bit  $i$  must be 1. But one-bit patterns are clearly inadequate for estimating leakage, as seen by an example like:

```

if (X % 2 == 0)
    Y = 0;
else
    Y = 0xffffffff;

```

in which all 32 bits of  $Y$  are non-fixed, even though  $Y$  has only two possible values.

Another contribution in this dissertation is the representation of two-bit patterns as a directed *implication graph*. Nodes represent bits or the negations of bits, and edges represent logical implication. Such representation not only provides a coher-



ent mathematical representation of two-bit patterns, but also facilitates many new techniques to improve the efficiency of two-bit pattern computation. We include three optimization techniques in this dissertation:

1. *random execution*: Random execution of the program can fill in many entries in the adjacency matrix of the implication graph without using STP queries.
2. *STP counterexamples*: When a STP query returns *invalid*, STP can give a counterexample showing why the query is invalid. This gives a new, and probably rare, feasible output. This output may well reveal some interesting bit patterns, which we have not seen before. Thus, it cheaply fills in many additional entries in the adjacency matrix.
3. *deductive closure*: Given a partially known adjacency matrix, we can fill in additional entries whose value is a logical consequence of the entries already known.

Portions of this dissertation are based on work previously published on ACM SIGPLAN 2011 Workshop on Programming Languages and Analysis for Security (PLAS) [MS11], which introduced the concept of two-bit pattern analysis and described a straightforward algorithm to compute the two-bit patterns of the output for a given deterministic program, and 2013 International Workshop on Quantitative Aspects in Security Assurance (QASA) [MS13], which introduced a coherent representation of two-bit patterns as implication graphs and described an optimized algorithm to speed up the two-bit pattern computing. My specific contributions include proposing the original idea of using bit-level constraints to characterize the feasible outputs, building the implementation, and conducting the experiments on the collected case studies.

### 1.3 Organization

The rest of the dissertation is structured as follows. In Chapter 2, we begin by reviewing the theories of Shannon entropy, min-entropy and g-entropy measure of leakage that we use in this work. In Chapter 3, we explain the concept of two-bit patterns in detail using an illustrative example. In Chapter 4, we explore several optimized techniques to speed up the two-bit pattern analysis. In Chapter 5, we present the results achieved by our techniques on a number of case studies drawn from the recent literature in quantitative information flow analysis. In Chapter 6, we discuss related work and compare it with our approach. Finally, in Chapter 7, we discuss future directions and conclude.

## CHAPTER 2

### PRELIMINARIES

This chapter introduces the mathematical foundations of quantitative information flow analysis, recalling important concepts of information theory [Sha48, Fel68, Gal68, Mac03, CT06] and their measure of information leakage. These concepts include the classical *mutual information* measure of leakage, *min-entropy* measure of leakage proposed in [Smi09], and *g-leakage* proposed in [ACPS12].

#### 2.1 Information Channel

An information theoretic *channel* offers a very general setting for theories of quantitative information flow. Channels do not rely on any explicit notion of “messages”; instead they capture relationships between system *inputs* and *outputs* through a *channel matrix*. A channel matrix gives the conditional probability of each possible output, given each possible input. Here “outputs” can be any subtle aspect of the system’s behavior that is observable to an adversary. For instance, the amount of time taken by a cryptographic operation may be observable by an adversary, and may reveal information about the secret key.

Formally, a *channel* is a triple  $(\mathcal{X}, \mathcal{Y}, C)$ , where  $\mathcal{X}$  is a finite set of secret input values,  $\mathcal{Y}$  is a finite set of observable output values, and  $C$  is an  $|\mathcal{X}| \times |\mathcal{Y}|$  matrix, called the *channel matrix*, such that  $C[x, y] = p(y|x)$ , the conditional probability of obtaining output  $y$  given that the input is  $x$ . Note that each row of  $C$  sums to 1. An important special case is a *deterministic channel*, in which each input produces a unique output. In terms of  $C$ , this means that each entry is either 0 or 1, and each row contains exactly one 1.

Any *a priori* distribution  $\pi$  on  $\mathcal{X}$  determines a random variable  $X$ . By  $\pi$  and  $C$ , the joint probability  $p_{XY}$  on  $\mathcal{X} \times \mathcal{Y}$  can be determined:

$$p(x, y) = \pi[x]C[x, y]$$

It can be shown that  $p_{XY}$  contains all information needed to determine a marginal distribution for  $Y$ :

$$p(y) = \sum_{x \in \mathcal{X}} p(x, y)$$

and to reconstruct the marginal distribution for  $X$ :

$$p(x) = \sum_{y \in \mathcal{Y}} p(x, y) = \pi[x]$$

Hence the conditional probabilities in  $C$  can also be reconstructed from  $p_{XY}$ :

$$p(y|x) = \frac{p(x,y)}{p(x)} = C[x, y]$$

provided that  $p(x)$  is nonzero.

We are interested in quantifying the amount of information that flows from  $X$  to  $Y$  by considering an adversary  $\mathcal{A}$ , who knows both  $C$  and  $\pi$ , wishes to guess the value of  $X$  by observing  $Y$ 's value after the execution of  $C$ . It is, therefore, natural to measure information leakage by comparing  $\mathcal{A}$ 's “uncertainty” about  $X$  before and after seeing the value of  $Y$ , using the equation

$$\text{leakage} = \text{initial uncertainty} - \text{remaining uncertainty.}$$

## 2.2 Measuring Leakage using Mutual Information

Until recently, most works on quantitative information flow (for example, [CPP08] and [Mal07]) have defined “uncertainty” using *Shannon entropy* and *conditional Shannon entropy* [Sha48]:

$$H(\pi) = - \sum_{x \in \mathcal{X}} \pi[x] \log \pi[x]$$

and

$$H(\pi, C) = \sum_{y \in \mathcal{Y}} p(y) H(p_{X|y}),$$

which leads to defining leakage as *mutual information*:

$$\text{leakage} = H(\pi) - H(\pi, C) = I(\pi, C).$$

A critical question about any leakage measure, however, is whether it gives good operational security guarantees. In particular, we would like to know whether the measure of remaining uncertainty accurately reflects the threat to  $X$ , given  $Y$ . For  $H(\pi, C)$ , Massey's *guessing entropy* bound [Mas94] shows that  $G(\pi, C)$ , the expected number of guesses required to guess  $X$  given  $Y$ , grows exponentially with  $H(\pi, C)$ . A weakness of this bound, however, is that  $G(\pi, C)$  can be arbitrarily high, even when  $X$  is highly vulnerable to being guessed in *one* try. A key example from [Smi09] illustrates this. Consider the program:

```
if (X % 8 == 0)
    Y = X;
else
    Y = 1;
```

(2.1)

where  $X$  is a uniformly-distributed 64-bit unsigned integer,  $0 \leq X < 2^{64}$ , so that the initial uncertainty  $H(\pi) = 64$ . Using the symmetric property of the mutual information and the fact that  $H(Y|X) = 0$  in deterministic programs, the mutual

information leakage of this program can be easily calculated

$$\begin{aligned}
I(\pi, \text{Ex2.1}) &= I(Y, X) \\
&= H(Y) - H(Y|X) \\
&= H(Y) \\
&= \sum_{y \in \mathcal{Y}} p(y) \log \frac{1}{p(y)} \\
&= 2^{61} 2^{-64} \log 2^{64} + \frac{7}{8} \log \frac{8}{7} \approx 8.17
\end{aligned}$$

which means that the remaining uncertainty  $H(\pi, C) \approx 55.83$ . Here the adversary  $\mathcal{A}$ 's expected probability of guessing  $X$  in one try exceeds  $1/8$ , since  $X$  is leaked completely whenever  $Y \neq 1$ . Nevertheless, the guessing entropy is high, since nothing is leaked when  $Y = 1$  (except the fact that the last three bits are not all 0):

$$G(\pi, \text{Ex2.1}) = \frac{1}{8} \cdot 1 + \frac{7}{8} \cdot \frac{1}{2} \cdot \left(\frac{7}{8} 2^{64} + 1\right) \approx 2^{62.6}.$$

It is instructive to compare program (2.1) with

$$Y = X \ \& \ 0777; \tag{2.2}$$

which simply copies the 9 bits of  $X$  into  $Y$ . The mutual information leakage of program (2.2) is 9, making it *worse* than program (2.1), even though it gives  $\mathcal{A}$  a probability of guessing  $X$  in one try of only  $2^{-55}$ , since the first 55 bits of  $X$  remain completely unknown.

### 2.3 Measuring Leakage using Min-entropy

In view of the unsatisfactory security guarantees given by mutual information leakage, it was proposed in [Smi09] to define ‘‘uncertainty’’ in terms of the *vulnerability*

of  $X$  to being guessed correctly *in one try* by  $\mathcal{A}$ . Again, we make the assumption that  $\mathcal{A}$  knows  $\pi$  and  $C$ , then the *a priori* vulnerability is

$$V(\pi) = \max_{x \in \mathcal{X}} \pi[x]$$

and the *a posteriori* vulnerability is

$$\begin{aligned} V(\pi, C) &= \sum_{y \in \mathcal{Y}} p(y) \max_{x \in \mathcal{X}} p(x|y) \\ &= \sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} p(x, y) \\ &= \sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} \pi[x] C[x, y]. \end{aligned}$$

We convert from vulnerability to uncertainty by taking the negative logarithm, giving Rényi's *min-entropy* [R61]. Our definitions, then, are

- initial uncertainty:  $H_\infty(\pi) = -\log V(\pi)$
- remaining uncertainty:  $H_\infty(\pi, C) = -\log V(\pi, C)$

Finally, we define the *min-entropy leakage* from  $X$  to  $Y$  via  $C$ , denoted  $\mathcal{L}(\pi, C)$ , to be

$$\begin{aligned} \mathcal{L}(\pi, C) &= H_\infty(\pi) - H_\infty(\pi, C) \\ &= -\log V(\pi) - (-\log V(\pi, C)) \\ &= \log \frac{V(\pi, C)}{V(\pi)}. \end{aligned}$$

Thus, min-entropy leakage is the logarithm of the factor by which knowledge of  $Y$  increases the expected one-guess vulnerability of  $X$ . Using this new definition of leakage to re-evaluate program (2.1), we find that its min-entropy leakage is 61.00, reflecting the fact that  $V(\pi, C) \approx 1/8$ . In contrast, for program (2.2) the min-entropy leakage is 9, reflecting the fact that  $V(\pi, C) = 2^{-55}$ .

Min-entropy leakage is a step toward measuring information leakage with operational significances. However, there still exist scenarios where leakage measured by min-entropy can be misleading. One such situation is where  $\mathcal{A}$  is allowed to make multiple guesses. To illustrate, comparing the program (2.1) with the following example from [Smi09]:

$$Y = X \mid 07; \tag{2.3}$$

Again, we assume that  $X, Y$  are 64-bit unsigned integers and  $X$  is uniformly distributed. Both programs have 61.000 bits min-entropy leakage. However, they present different threats in the 8-guess scenario.  $\mathcal{A}$  can determine  $X$  within 8 guesses after observing the value of  $Y$  produced by program (2.3), while program (2.1) offers  $\mathcal{A}$  almost no clue about  $X$  seven-eighths of the time. Another scenario indicating the limitation of min-entropy leakage is the case where  $\mathcal{A}$  is only interested in guessing the secret *partially* or *approximately*. An example from [ACPS12] illustrates this. Consider a probabilistic channel which takes a secret array  $X$  containing 10-bit, uniformly-distributed passwords for 1000 users and produces one randomly-chosen user’s password along with his/her index:

$$\begin{aligned} u &\stackrel{?}{\leftarrow} \{0\dots999\}; \\ Y &= (u, X[u]); \end{aligned} \tag{2.4}$$

If we take all the users’ passwords as a whole, then the channel’s prior vulnerability is  $2^{-10000}$  and its posterior vulnerability is  $2^{-9990}$ . Thus, the leakage measured by min-entropy is 10 bits. If we focus on the threat to any particular user  $i$ ’s password, then the prior vulnerability becomes  $2^{-10}$  and the posterior vulnerability becomes  $0.001 \cdot 1 + 0.999 \cdot 2^{-10} \approx 0.00198$ , since in one time out of a thousand,  $\mathcal{A}$  learns user  $i$ ’s password but knows nothing about it the rest of the time. Thus, the leakage of this “sub-channel” measured by min-entropy is  $\log 2.023 \approx 1.016$  bits out of 10 bits.



In either circumstance, min-entropy leakage fails to reflect the real danger in this example: *some* user’s password is always publicized.

## 2.4 Measuring Leakage using Gain Functions and $g$ -leakage

To overcome its limitation and to model the threats in a wide variety of scenarios, min-entropy is generalized using the notion of *gain function* [ACPS12]. The key idea is that for each guess  $w$  which  $\mathcal{A}$  could make about the secret, there is a value between 0 and 1 quantifying the benefit which he/she gains when the secret is actually  $x$ . Different operational scenarios can be expressed by carefully designed gain functions. Formally, given a set  $\mathcal{W}$  of guesses and a set  $\mathcal{X}$  of secrets, a gain function  $g$  is a function:  $\mathcal{W} \times \mathcal{X} \rightarrow [0, 1]$ . Then a generalization of the prior vulnerability is defined as:

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

The rationale of the definition is that adversary  $\mathcal{A}$  makes a guess  $w$  that maximizes the expected gain over  $\mathcal{X}$ . The generalization of the posterior vulnerability is defined in a similar way:

$$\begin{aligned} V_g(\pi, C) &= \sum_{y \in \mathcal{Y}} \max_{w \in \mathcal{W}} \sum_{x \in \mathcal{X}} \pi[x]C[x, y]g(w, x) \\ &= \sum_{y \in \mathcal{Y}} \max_{w \in \mathcal{W}} \sum_{x \in \mathcal{X}} p(x, y)g(w, x) \\ &= \sum_{y \in \mathcal{Y}} p(y)V_g(p_{X|y}) \end{aligned}$$

The definitions of  $g$ -entropy and  $g$ -leakage are similar to the definitions of min-entropy and min-leakage:

$$\begin{aligned}
H_g(\pi) &= -\log V_g(\pi) \\
H_g(\pi, C) &= -\log V_g(\pi, C) \\
\mathcal{L}_g(\pi, C) &= H_g(\pi) - H_g(\pi, C) = \log \frac{V_g(\pi, C)}{V_g(\pi)}
\end{aligned}$$

This section is concluded by introducing several gain function examples from [ACPS12], which illustrate how gain functions allow a wide range of scenarios to be expressed. A very simple gain function is the *identity gain function*  $g_{id} : \mathcal{X} \times \mathcal{X} \rightarrow [0, 1]$ . It describes an operational scenario when  $\mathcal{A}$  only benefits from guessing the entire secret correctly. Formally, it is defined as:

$$g_{id}(w, x) = \begin{cases} 1, & \text{if } w = x \\ 0, & \text{if } w \neq x \end{cases}$$

[ACPS12] shows that ordinary vulnerability is a special case of  $g$ -vulnerability.

**Proposition 2.4.1** *Vulnerability under  $g_{id}$  coincides with vulnerability:*

$$V_{g_{id}}(\pi) = V(\pi)$$

*Proof.* For any  $w$ ,  $\sum_x \pi[x]g_{id}(w, x) = \pi[w]$ . So  $V_{g_{id}}(\pi) = \max_w \pi[w] = V(\pi)$ .  $\square$

This implies that  $g_{id}$ -leakage also coincides with min-entropy leakage.

To specify the scenario where  $\mathcal{A}$  can make 3 guesses to find the secret, *3-tries gain function* is crafted where  $W \in 2^{\mathcal{X}}$  and  $|W| = 3$ :

$$g_3(W, x) = \begin{cases} 1, & \text{if } x \in W \\ 0, & \text{otherwise.} \end{cases}$$

The gain function  $g_3$  helps distinguish program (2.1) from program (2.3). Recall that both programs have min-entropy leakage of 61 bits when  $X$  is a uniformly-distributed 64-bit unsigned integer. Their leakages measured by  $g_3$  are quite different. In program (2.3), both the prior vulnerability and the poster vulnerability increase by a factor of 3 due to the 3 tries. Hence,  $\mathcal{L}_{g_3}(\pi, \text{Ex2.3})$  remains the same. Program (2.1)'s posterior vulnerability doesn't increase very much under  $g_3$ :

$$V_{g_3}(\pi, \text{Ex2.1}) = \frac{1}{8} \cdot 1 + \frac{7}{8} \cdot 3 \cdot 2^{-64} \approx \frac{1}{8}$$

Hence,  $\mathcal{L}_{g_3}(\pi, \text{Ex2.1})$  is reduced to about 59.4 bits. However, in the scenario where there is a penalty for wrong guesses (e.g., electrocution via keyboard) and  $\mathcal{A}$  is reluctant to guess, program (2.1) is actually worse. This is because  $\mathcal{A}$  knows the exact value of  $X$  whenever  $Y \neq 1$ .

To reflect the danger of program (2.4), adversary  $\mathcal{A}$  is only interested in some users' password, we can design the following  $\mathcal{W}$  and gain function  $g$ :

$$\mathcal{W} = \{(u, x) | 0 \leq u \leq 999 \text{ and } 0 \leq x \leq 1023\}$$

$$g((u, x), X) = \begin{cases} 1, & \text{if } X[u] = x \\ 0, & \text{otherwise.} \end{cases}$$

Revisiting example (2.4), the prior vulnerability becomes  $2^{-10}$  under  $g$ , since every user's password is uniformly distributed and ranges from 0 to 1023. The posterior vulnerability under  $g$  becomes 1, since for every observable output  $(u, X[u])$ ,  $\mathcal{A}$  can always choose the guess accordingly to guarantee gain 1. Hence, the  $g$ -leakage under  $g$  is 10 bits out of 10 bits. Comparing this result with the min-entropy leakage which is 10 bits out of 10000 bits, leakage under  $g$  describes the threat more accurately.

## 2.5 Channel Capacity

This section will discuss the channel capacity related properties of the entropy definitions introduced in the previous sections. Primarily, we focus on the capacities of deterministic channels which are the objects of leakage analysis in this dissertation. *Channel capacity*, the maximum leakage over all possible *a priori* distributions, is an important notion in information theory. It provides a further abstract way of studying a channel independent of any particular prior distribution and focuses on its “worst-case” leakage. Throughout this dissertation, we will use the name *Shannon capacity* referring to the capacity under mutual information, *min-capacity* with the notation  $\mathcal{ML}(C)$  referring to the capacity under min-entropy leakage, and *g-capacity* with the notation  $\mathcal{ML}_g(C)$  referring to the capacity under *g*-leakage.

In general, calculating the Shannon capacity of a channel matrix is difficult. But as shown in [BCP09, KS10], calculating the min-capacity is straightforward. It is simply the logarithm of the sum of the column maximums of  $C$  :

**Theorem 2.5.1** *For any channel matrix  $C$ ,*

$$\mathcal{ML}(C) = \log \sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} C[x, y]$$

*Proof.*

$$\begin{aligned} \mathcal{L}(\pi, C) &= \log \frac{V(\pi, C)}{V(\pi)} \\ &= \log \frac{\sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} (\pi[x] C[x, y])}{\max_{x \in \mathcal{X}} \pi[x]} \\ &\leq \log \frac{\sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} C[x, y] (\max_{x \in \mathcal{X}} \pi[x])}{\max_{x \in \mathcal{X}} \pi[x]} \\ &= \log \sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} C[x, y] \end{aligned}$$

□

Notice that the min-capacity is always realized by a uniform distribution on  $\mathcal{X}$ . As a corollary, the min-capacity of a deterministic channel is just the logarithm of the number of possible outputs. Interestingly, this is also its Shannon capacity [Smi09]:

**Theorem 2.5.2** *If  $C$  is deterministic, then its min-capacity and Shannon capacity coincide, with both equal to  $\log |\mathcal{Y}|$  (assuming that every element of  $\mathcal{Y}$  is really possible).*

*Proof.* Since  $C$  is a deterministic channel, each of its entry is either 0 or 1. And there is at least one entry of 1 for each column since every element in  $\mathcal{Y}$  is feasible. By Theorem 2.5.1, the min-capacity of  $C$  is the logarithm of the sum of its column maximums. Hence the min-capacity of  $C$  is just  $\log |\mathcal{Y}|$ .

Recall that the Shannon leakage of a deterministic channel is  $H(Y)$ .  $H(Y)$  achieves its maximum value  $\log |\mathcal{Y}|$  when  $\mathcal{Y}$  is uniformly distributed. This can always be realized by some *a priori* distribution on  $\mathcal{X}$ .  $\square$

When it comes to  $g$ -capacity, there is miraculous order between  $g$ -capacity and min-capacity: for every gain function  $g$ , min-capacity is an upper bound on  $g$ -capacity [ACPS12]:

**Theorem 2.5.3** *For every channel  $C$  and gain function  $g$ ,  $\mathcal{ML}_g(C) \leq \mathcal{ML}(C)$ .*

*Proof.*

$$\begin{aligned} V_g(\pi, C) &= \sum_{y \in \mathcal{Y}} \max_{w \in \mathcal{W}} \sum_{x \in \mathcal{X}} C[x, y] \pi[x] g(w, x) \\ &\leq \sum_{y \in \mathcal{Y}} \max_{w \in \mathcal{W}} \sum_{x \in \mathcal{X}} (\max_{x \in \mathcal{X}} C[x, y]) \pi[x] g(w, x) \\ &= \left( \sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} C[x, y] \right) \left( \max_{w \in \mathcal{W}} \sum_{x \in \mathcal{X}} \pi[x] g(w, x) \right), \end{aligned}$$

Applying Theorem 2.5.1, we have

$$V_g(\pi, C) \leq 2^{\mathcal{ML}(C)} V_g(\pi)$$

Hence,

$$\mathcal{L}_g(\pi, C) = \log \frac{V_g(\pi, C)}{V_g(\pi)} \leq \mathcal{ML}(C)$$

This implies that  $\mathcal{ML}_g(C) \leq \mathcal{ML}(C)$ .

□

From the theorems above, we have the following conclusion on the channel capacities of deterministic channels:

**Corollary 2.5.4** *The capacity of a deterministic system, whether measured by Shannon entropy or min-entropy, is the logarithm of the number of feasible outputs. This quantity is also an upper bound on the  $g$ -leakage, for any gain function  $g$ .*

*Proof.* Follows from Theorem 2.5.2 and Theorem 2.5.3. □

## CHAPTER 3

### TWO-BIT PATTERNS

Given any theory of quantitative information flow, it is desirable to develop automatic techniques for calculating/bounding the amount of leakage in a system, to verify whether it conforms to a given quantitative flow policy. As we have seen in Corollary 2.5.4, in the context of deterministic systems, this task is reduced to calculating/bounding the number of the feasible outputs. In this chapter, we introduce the approach of *two-bit patterns* to calculate upper bounds on the channel capacity of a deterministic C-like program. The chapter is based on our work previously published at ACM SIGPLAN 2011 Workshop on Programming Languages and Analysis for Security (PLAS) [MS11].

The approach can be divided into three major steps. The first step is to derive the mathematical relationship between the initial values of the secret input variables and final values of the output variables. The second step is to discover the *one-bit patterns* that constrain each *individual* bit and the *two-bit patterns* that constrain each *pair* of bits in the final values of the outputs. The third step is to use a #SAT algorithm to count the number of instances that satisfy all the bit patterns discovered in the second step; the logarithm of this number is our upper bound on the channel capacity.

Throughout this chapter, we use the analysis of the program shown in Figure 3.1 to illustrate these three steps in detail. Assuming that X is the secret input variable,

```
X = X & 0x77777777;  
if (X <= 64) Y = X; else Y = 0;  
if (Y % 2 == 0) Y++;
```

Figure 3.1: Illustrative example program that leaks information from X to Y

Y is the output variable, and both are 32-bit unsigned integers, the program can produce 17 distinct outputs: 1, 3, 5, 7, 17, 19, 21, 23, 33, 35, 37, 39, 49, 51, 53, 55, 65.

### 3.1 Deriving Predicates

This step can be accomplished by generating a series of predicates which describe the relationship between the value of variables before and after each computation step. As in SSA (single static assignment) form, we represent the successive values of each variable V by a sequence of *symbols* V0, V1, V2, etc. For each computation step, a fresh symbol is introduced for each variable affected by the step; it represents the value of the variable after the computation step. Then a predicate is derived to describe the relationship between the new symbol and the previous symbols. For example, the first assignment in the program in Figure 3.1 gives rise to the predicate

```
X1 = X0 & 0x77777777
```

Here the symbols X0 and X1 represent variable X's value before and after the first assignment. Since variable Y is not affected, no new symbol is introduced for it and Y0 (variable 0's initial value) remains current.

The second and third commands in the example each require a conditional expression:

```
Y1 = if X1 <= 64 then X1 else 0
```

```
Y2 = if Y1 mod 2 = 0 then Y1+1 else 01
```

These three predicates implicitly constitute a symbolic description of Y's final value in terms of X's initial value. Currently, the loops need to be unrolled completely to get their SSA forms.



```

X0, X1 : BITVECTOR(32);
Y1, Y2 : BITVECTOR(32);

ASSERT(X1 = X0 & 0hex77777777);

ASSERT(Y1 = IF (BVLE(X1, 0hex00000040))
              THEN X1
              ELSE 0hex00000000
              ENDIF);

ASSERT(Y2 = IF (BVMOD(32, Y1, 0hex00000002) = 0hex00000000)
              THEN BVPLUS(32, Y1, 0hex00000001)
              ELSE Y1
              ENDIF);

```

Figure 3.2: Translation of illustrative example into STP

Next, we translate the predicates that we have derived into the language of the STP solver [GD07]. STP is an efficient decision procedure for testing validity (or satisfiability) of predicates in quantifier-free first-order logic over bit-vectors and arrays; it has been widely used by many program analysis research groups.

The translation is straightforward—each symbol (representing a 32-bit value) is declared as a bit-vector, the operations in each predicate are replaced with STP equivalents. For instance, the expression `Y1 mod 2` translates into

```
BVMOD(32, Y1, 0hex00000002)
```

where `BVMOD` stands for “Bit-Vector Modulo” and the parameter `32` gives the word size. Finally, each predicate is translated into an STP `ASSERT` statement. The complete translation of the program in Figure 3.1 is shown in Figure 3.2.

So far, we do the translation to STP manually, leaving generalization and automation of the process to future work.

## 3.2 Discovering One-Bit and Two-Bit Patterns

Now we wish to discover the relations (bit patterns) among the bits in  $Y_2$ , which is the final value of the output variable  $Y$ . We achieve this by making STP *queries* with respect to the assertions generated in Step 1. In STP,  $\text{QUERY}(P)$  asks whether predicate  $P$  is a logical consequence of the **ASSERT** statements that have been made. If so, STP responds **VALID**; if not, it responds **INVALID**.

We start by determining the one-bit pattern for each bit of  $Y_2$ . A one-bit pattern describes the set of possible values for a particular bit. Since a bit is either 0 or 1, there are three one-bit patterns: *Zero*, *One*, and *Non-fixed*, which means that it is possible for the bit to be either 0 or 1. The STP query

```
QUERY(Y2[i:i] = 0bin0)
```

tests whether bit  $i$  of  $Y_2$  is necessarily 0, given the **ASSERT** statements that have been made. If STP returns **VALID**, then we can conclude that the bit must be 0; if it returns **INVALID**, then we know that the bit  $i$  can be 1. Similarly, the STP query

```
QUERY(Y2[i:i] = 0bin1)
```

tests whether bit  $i$  of  $Y_2$  is necessarily 1. If both queries return **INVALID**, then the bit can be either 0 or 1.

Using more readable notation, if we denote the final output symbol with  $Y_f$ , then the algorithm to determine the one-bit pattern for bit  $i$  of  $Y_f$  is

```
if ( $Y_f[i] = 0$ ) is valid then  
    Zero  
else if ( $Y_f[i] = 1$ ) is valid then  
    One  
else
```

*Non-fixed*

**end if**

Notice that it requires one or two STP queries per bit.

On the example in Figure 3.1, as translated into Figure 3.2, STP discovers that 26 of the bits in Y2 have pattern *Zero*, namely, bits 31 down to 7, along with bit 3. Also, bit 0 has pattern *One*. The remaining 5 bits (bits 6, 5, 4, 2, and 1) have pattern *Non-fixed*. The one-bit patterns can be displayed compactly in a vector, using \* to represent the bits with pattern *Non-fixed*:

```
00000000000000000000000000000000***0**1
```

Discovering these 32 one-bit patterns required a total of 38 STP queries and 1072 ms.<sup>1</sup> Notice that we can immediately conclude that the number of possible values for Y2 is at most  $2^5 = 32$ , since it has only 5 non-fixed bits.

We can tighten our upper bound by next determining the two-bit pattern for every pair of bits; notice that we need to do this only among the bits with the pattern *Non-fixed*. A two-bit pattern describes the set of possible values that a pair of bits can have. Hence the set of two-bit patterns is the powerset of the set of two-bit values, minus the empty set. There are four possible values for a pair of bits: {00, 01, 10, 11}. Hence the number of two-bit patterns is  $2^4 - 1 = 15$ .

Here is the complete enumeration of the possible two-bit patterns:

1. {00}
2. {01}
3. {10}
4. {11}

---

<sup>1</sup>Throughout this paper, all times are given in milliseconds.

5.  $\{00, 01\}$
6.  $\{00, 10\}$
7.  $\{01, 11\}$
8.  $\{10, 11\}$
9.  $\{00, 11\}$
10.  $\{01, 10\}$
11.  $\{00, 01, 10\}$
12.  $\{00, 01, 11\}$
13.  $\{00, 10, 11\}$
14.  $\{01, 10, 11\}$
15.  $\{00, 01, 10, 11\}$

Notice, however, that the first eight patterns will never occur, since in each of them at least one of the two bits is fixed. Therefore, we only need to consider the last seven patterns (patterns 9 through 15). Interestingly, each of these seven patterns can be interpreted as a binary relation:

- $\{00, 11\}$  is the *equality* relation
- $\{01, 10\}$  is the *inequality* relation
- $\{00, 01, 10\}$  is the logical *nand* relation
- $\{00, 01, 11\}$  is the  $\leq$  relation
- $\{00, 10, 11\}$  is the  $\geq$  relation
- $\{01, 10, 11\}$  is the logical *or* relation
- $\{00, 01, 10, 11\}$  is the *universal* relation, saying that the two bits are independent of each other.

We will refer concisely to these seven patterns as *Eq*, *Neq*, *Nand*, *Leq*, *Geq*, *Or*, and *Free*, respectively.

The two-bit patterns can be computed by a straightforward algorithm. It determines the two-bit patterns via a decision tree of queries. For instance,

QUERY(Y2[i:i] = 0bin0 OR Y2[j:j] = 0bin0)

returns VALID iff bits *i* and *j* cannot both be 1. Similarly,

QUERY(Y2[i:i] = 0bin1 OR Y2[j:j] = 0bin1)

returns VALID iff bits *i* and *j* cannot both be 0. So if both these queries return VALID, then the pattern for bits *i* and *j* must be {01, 10}, or *Neq*. (Notice that both 01 and 10 must be possible, because we find two-bit patterns only among bits that are not fixed.)

Other two-bit patterns can be determined in a similar manner. The complete algorithm is shown in Figure 3.3. Notice that under this algorithm, 2 STP queries are required to determine the *Neq* and *Nand* patterns, 3 STP queries are required to determine the *Eq*, *Geq*, and *Leq* patterns, and 4 STP queries are required to determine the *Or* and *Free* patterns. Hence, if the output  $Y_f$  contains  $m$  non-fixed bits, then at most  $2m(m - 1)$  STP queries suffice to determine all the two-bit patterns.

On the program in Figure 3.1, it turns out that there are four interesting two-bit patterns among the 5 non-fixed bits of Y2, namely *Nand*(6,1), *Nand*(6,2), *Nand*(6,4), and *Nand*(6,5). All other pairs of non-fixed bits are *Free*. Finding these two-bit patterns required a total of 32 STP queries and 2558 ms.

We remark that the average time per STP query for the illustrative example is about 41 ms for the one-bit pattern queries, and 80 ms for the two-bit pattern queries. These times are unusually high, compared with the times for the other case

```

for all non-fixed bits  $i$  and  $j$  such that  $i > j$  do
  if  $(Y_f[i] = 0 \vee Y_f[j] = 0)$  is valid then
    if  $(Y_f[i] = 1 \vee Y_f[j] = 1)$  is valid then
       $Neq(i, j)$ 
    else
       $Nand(i, j)$ 
    end if
  else if  $(Y_f[i] \geq Y_f[j])$  is valid then
    if  $(Y_f[i] \leq Y_f[j])$  is valid then
       $Eq(i, j)$ 
    else
       $Geq(i, j)$ 
    end if
  else if  $(Y_f[i] \leq Y_f[j])$  is valid then
     $Leq(i, j)$ 
  else if  $(Y_f[i] = 1 \vee Y_f[j] = 1)$  is valid then
     $Or(i, j)$ 
  else
     $Free(i, j)$ 
  end if
end for

```

Figure 3.3: Algorithm to determine two-bit patterns for  $Y_f$

studies by the same straightforward approach in Chapter 5. The cause turns out to be the use here of the expensive `BVMOD` operation. If we rewrite the last line of the illustrative example from

```
if (Y % 2 == 0) Y++;
```

to the equivalent

```
if (Y & 0x00000001 == 0) Y++;
```

we find that the cost per STP query drops to under 2 ms.

### 3.3 Counting the Number of Solutions

Finally, we determine an upper bound on the number of possible outputs by counting the number of solutions to the two-bit patterns. We do this using the `SatisfiabilityCount` function provided by *Mathematica*.<sup>2</sup> Given a boolean proposition  $P$  and a list of boolean variables  $b_1, b_2, \dots$ , the Mathematica call

$$\text{SatisfiabilityCount}[P, \{b_1, b_2, \dots\}]$$

returns the number of truth assignments to  $b_1, b_2, \dots$  that make  $P$  true. (Notice that if some  $b_i$  does not occur in  $P$ , then it can be freely set to *true* or *false* without affecting the truth of  $P$ .)

We call `SatisfiabilityCount` with a boolean proposition formed from the two-bit patterns (other than `Free`) discovered in Step 2, together with a list of all the non-fixed bits of the output. In the case of the program in Figure 3.1, we make the call

---

<sup>2</sup><http://www.wolfram.com/mathematica/>

```
In[1] = SatisfiabilityCount[Nand[b6,b1] &&
                             Nand[b6,b2] &&
                             Nand[b6,b4] &&
                             Nand[b6,b5] ,
                             {b6,b5,b4,b2,b1}]
```

which produces the result

```
Out[1] = 17
```

in less than 1 ms. It is straightforward to see that this result is an *upper bound* on the number of outputs that can be produced by the program; in this example, it turns out to be exactly correct. It implies a min-capacity of at most  $\log 17 \approx 4.087$  bits.

From a theoretical perspective, it is interesting to note that the proposition  $P$  that we construct from the two-bit patterns can easily be put into 2CNF (2 conjunctive normal form). For example, if bits  $a$  and  $b$  have pattern **Eq**, then they cannot be 01 or 10, giving

$$\neg(\bar{a}b + a\bar{b}) \equiv (a + \bar{b})(\bar{a} + b).$$

While testing satisfiability of propositions in 2CNF can be done in linear time, it turns out that counting the number of satisfying assignments is still #P-complete [Val79]. Nevertheless, our experiments have been encouraging with respect to the feasibility of this approach—in all cases, we found that `SatisfiabilityCount` took a negligible amount of time compared with the time to find the bit patterns.

We have performed the two-bit pattern computing for the aforementioned illustrative example on a Lenovo B570 computer with a 2.3 GHz Intel Core i3-2310M processor and 3GB of DDR3 RAM. The machine runs Ubuntu 12.04 Linux operating system. We have implemented the two-bit pattern computing algorithm using



OpenJDK for Java 6 and Java binding for the STP decision procedure. We used the `SatisfiabilityCount` function in *Mathematica 9* to count the number of solutions for the two-bit patterns. Under this environment, our approach takes less than 3 seconds to find an upper bound (17) on the number of possible outputs, and here it turns out to be exact.

One might wonder how these results compare with a more brute-force approach to counting the number of possible outputs. Specifically, we can test whether any 32-bit value  $v$  is a possible output using the STP query

```
QUERY(NOT(Y2[0:31] = v))
```

which returns `INVALID` iff  $v$  is a possible output. If we try this query on all  $2^{32}$  values of  $v$ , from `0x00000000` to `0xffffffff`, then we will know exactly how many outputs are possible. However, experiments under the environment described above show that on average each of these queries takes 20 ms, which implies that it would take 2.7 years to complete the  $2^{32}$  queries.

Another approach to counting the number of possible outputs is exhaustive testing. We can execute the program on each 32-bit value of  $X$  and count how many distinct values  $Y$  may obtain. Experiments (again under the environment described above) show that on average each execution takes 0.1 ms, which implies that it would take about 5 days to complete all  $2^{32}$  executions. <sup>3</sup>

---

<sup>3</sup>Additional time would be required to count the number of distinct values produced. We have not implemented such a procedure.

## CHAPTER 4

### OPTIMIZATION

In this chapter, we introduce an efficient approach to compute two-bit patterns. The chapter is based on our work previously published at 2013 International Workshop on Quantitative Aspects in Security Assurance (QASA) [MS13]. The approach is based on four techniques: *implication graph*, *random execution*, *STP counterexamples* and *deductive closure*. We first show that the two-bit patterns can be represented as a directed implication graph, as used in the study of the 2SAT problem. Nodes represent bits or the negations of bits, and edges represent logical implication. Then, we show that random execution of the program can cheaply produce feasible outputs, which allow us to fill in many entries of the adjacency matrix representation of the implication graph without using STP queries. Moreover, STP counterexamples is a feature of STP solver. It gives a counter example to explain why a query returns *invalid*, and thus allows us to fill in many additional entries of the adjacency matrix. Finally, given a partially known adjacency matrix, we can perform deductive closure to fill in additional entries whose value is a logical consequence of the entries already known.

We combine these techniques into a single algorithm. As a re-evaluation on the illustrative example in Figure 3.1 shows, this optimized approach enables us to significantly reduce the time required for two-bit pattern analysis. More case studies on the effectiveness of this approach are presented in the next chapter.

#### 4.1 Formal Framework

This section explores the mathematical aspects of the two-bit patterns: implication graph representations, and their relationship with concrete states. The set of feasible outputs of a program can be modeled as a set  $R$  of *states*  $\rho$ . The bits in a state

are indexed by a set  $I$  of *indices*. (For example, for the 5-bit states modeled in Figure 1.1, we would have  $I = \{0, 1, 2, 3, 4\}$ .) Formally, a state  $\rho$  is a mapping:  $\rho : I \rightarrow \mathbb{B}$ , where  $\mathbb{B} = \{0, 1\}$ .

### 4.1.1 Implication Graphs

Recall that the seven possible two-bit patterns among a pair of non-fixed bits,  $Eq$ ,  $Neq$ ,  $Nand$ ,  $Leq$ ,  $Geq$ ,  $Or$ , and  $Free$ , can be encoded as 2CNFs. Since implication and negation are logically complete, each CNF can be expressed in terms of implications over *literals*, which are indices or negated indices:

Two-Bit Pattern	Implications
$Eq(i, j)$	$i \rightarrow j, \bar{j} \rightarrow \bar{i}, j \rightarrow i, \bar{i} \rightarrow \bar{j}$
$Neq(i, j)$	$i \rightarrow \bar{j}, j \rightarrow \bar{i}, \bar{i} \rightarrow j, \bar{j} \rightarrow i$
$Nand(i, j)$	$i \rightarrow \bar{j}, j \rightarrow \bar{i}$
$Leq(i, j)$	$i \rightarrow j, \bar{j} \rightarrow \bar{i}$
$Geq(i, j)$	$j \rightarrow i, \bar{i} \rightarrow \bar{j}$
$Or(i, j)$	$\bar{i} \rightarrow j, \bar{j} \rightarrow i$

(Notice that  $Free(i, j)$  does not result in any implications.)

This translation enables us to represent a set of two-bit patterns as a directed graph whose nodes are literals and whose edges represent implication; such graphs are known as *implication graphs* in the study of the 2SAT problem [Kro67, APT79]. As an example, Figure 4.1 shows the implication graph, in both graphical and adjacency matrix representations, corresponding to the two-bit patterns in Figure 1.1. (To avoid clutter, we omit self-loops in the graphical representation.) Implication

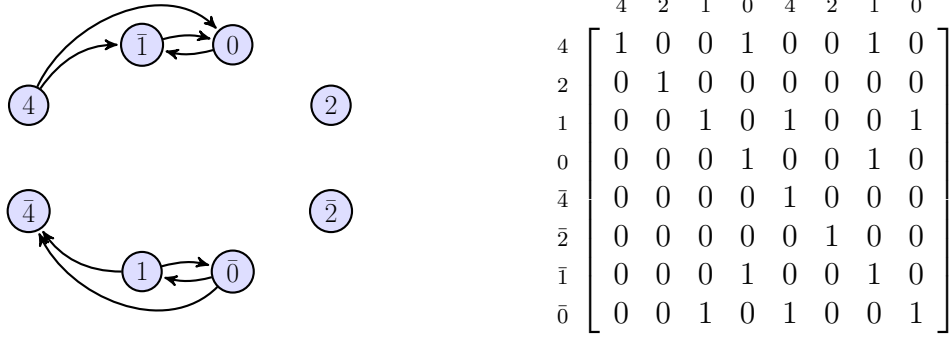


Figure 4.1: The implication graph for  $\{00010, 10001, 00001, 00110, 10101, 00101\}$

graphs have the property of being *skew-symmetric* [GK96], since there is an edge from  $i$  to  $j$  iff there is an edge from  $\bar{j}$  to  $\bar{i}$ .

The implication graph in Figure 4.1 omits mention of the fixed bit 3, since fixed bits do not contribute to multiple outputs. But the one-bit patterns can be incorporated into the implication graph. For example, the implication  $3 \rightarrow \bar{3}$  expresses that bit 3 must be 0.

### 4.1.2 Semantic Characterization

To establish the correctness of our two-bit pattern analysis, we need a semantic characterization of implication graphs (represented as adjacency matrices). To use the language of *abstract interpretation* [CC04], we want to see implication graphs as an *abstract domain* for the *concrete domain* of sets of states.

To facilitate this connection, we define *literals*  $\hat{I} = I \cup \{\bar{i} \mid i \in I\}$ . We, moreover, extend states to  $\hat{I}$  by specifying that  $\rho(\bar{i}) = \overline{\rho(i)}$ . Notice then that an implication  $i \rightarrow j$  holds in state  $\rho$  iff  $\rho(i) \leq \rho(j)$ .

Now, we define our *abstraction function*  $\alpha$  that maps a set  $R$  of states to an implication graph  $M$ :

**Definition 4.1.1** Abstraction function  $\alpha : \mathcal{P}(\hat{I} \rightarrow \mathbb{B}) \rightarrow (\hat{I} \times \hat{I} \rightarrow \mathbb{B})$  is given by

$$\alpha(R)_{ij} = \begin{cases} 1, & \text{if for all } \rho \in R, \rho(i) \leq \rho(j) \\ 0, & \text{otherwise.} \end{cases}$$

Next, we define the *concretization function*  $\gamma$  that maps an implication graph  $M$  to a set  $R$  of states:

**Definition 4.1.2** Concretization function  $\gamma : (\hat{I} \times \hat{I} \rightarrow \mathbb{B}) \rightarrow \mathcal{P}(\hat{I} \rightarrow \mathbb{B})$  is given by

$$\gamma(M) = \{\rho \mid \text{for all } i, j, \text{ if } M_{ij} = 1 \text{ then } \rho(i) \leq \rho(j)\}$$

(Note that 0s in  $M$  do not constrain the states.)

The key correctness property of the implication graph domain is given by the following theorem, which ensures that when we calculate implication graph  $M = \alpha(R)$ , where  $R$  is the set of feasible states, then we know that  $\gamma(M)$  is a *superset* of  $R$ , implying that we *over-approximate* the set of feasible states.

**Theorem 4.1.3** Given any set  $R$  of states,  $R \subseteq \gamma(\alpha(R))$ .

*Proof.* Let  $M = \alpha(R)$ . We need to prove that  $\rho \in R \Rightarrow \rho \in \gamma(M)$ . By the definition of  $\alpha$ ,  $\forall i, j$ , if  $M_{ij} = 1$  then  $\rho(i) \leq \rho(j)$  since  $\rho \in R$ . By the definition of  $\gamma$ , this means that  $\rho \in \gamma(M)$ .  $\square$

Note, however, that the relationships specified in an arbitrary implication graph  $M$  may be incoherent; for instance, we might have  $M_{ij} = 1$  and  $M_{jk} = 1$ , but  $M_{ik} = 0$ . Hence, we have the following definition.

**Definition 4.1.4** Implication graph  $M$  is coherent if there exists a set  $R$  such that  $M = \alpha(R)$ .

Coherent implication graphs behave well with respect to  $\gamma$  and  $\alpha$ :

**Theorem 4.1.5** *If  $M$  is coherent, then  $\alpha(\gamma(M)) = M$ .*

*Proof.* Let  $M = \alpha(R)$  and  $\alpha(\gamma(M)) = N$  where  $R$  is a nonempty set of states.  $\forall i, j$ , if  $M_{ij} = 1$  then  $\forall \rho \in \gamma(M), \rho(i) \leq \rho(j)$  by the definition of  $\gamma$ . Hence,  $N_{ij} = 1$  by the definition of  $\alpha$ .  $\forall i, j$ , if  $M_{ij} = 0$  then  $\exists \rho \in R, \rho(i) > \rho(j)$  by the definition of  $\alpha$ . By Theorem 4.1.3,  $\rho \in \gamma(M)$ . Hence,  $N_{ij} = 0$  by the definition of  $\alpha$ . Therefore,  $M$  is identical to  $N$ .

□

## 4.2 One-bit Patterns, Random Execution, and STP Counterexamples

As was shown in Figure 4.1, we include only the *non-fixed* bits in the implication graph. This means that computing the implication graph for the set  $R$  of feasible outputs of a given program still has to begin by determining the one-bit patterns. For each bit  $i$  of output  $Y$ , we use the same one-bit pattern queries as in the straightforward approach in Chapter 3 to determine whether it is *Zero*, *One* or *Non-fixed*. Also, only the *Non-fixed* bits in  $\hat{I}$  are included in the implication graph  $M$ .

In the hope of avoiding the need for so many STP queries, however, we first execute the program on a set of *randomly-chosen* inputs  $X$ . For each bit  $i$  of  $Y$ , these random executions reveal at least one possible value, allowing us to determine its one-bit pattern using just *one* STP query. If we are lucky, the random executions may reveal that bit  $i$  can be both 0 and 1, allowing us to conclude that it is *Non-fixed* without making *any* STP queries. (Of course, the likelihood of this will depend on the probability distribution on  $Y$ , as many programs produce certain values of  $Y$  with very low probability.) We found in our case studies that doing 40 random executions

typically gets most of the possible benefit without costing very much, so that is the number of random executions that we use in our implementation.

We can further improve efficiency by making use of *STP counterexamples*. If we query “ $Y[i] = 0?$ ” and STP returns *invalid*, then STP can give us, essentially for free, a counterexample showing why the query is invalid. This gives us a new, and probably rare, feasible output that we have not seen before. (Here it would be an output where  $Y[i] = 1$ .) This output may well reveal that some *other* bits of  $Y$  are *Non-fixed*, freeing us from the need to make queries about them.

### 4.3 Two-bit Patterns and Deductive Closure

Our goal is to determine the implication graph  $M$  using as few STP queries as possible. To this end, it is useful to extend to *partially-known* implication graphs, where we use  $\perp$  to denote unknown entries. Recall that we limit  $M$  to the non-fixed bits of  $\hat{I}$ , which is denoted by  $\hat{I}_N$ . Hence, formally,  $M : \hat{I}_N \times \hat{I}_N \rightarrow \mathbb{B}_\perp$ , where  $\mathbb{B}_\perp$  is the flat domain  $\{0, 1, \perp\}$  with partial order  $\perp \preceq 0$  and  $\perp \preceq 1$ . We also extend  $\preceq$  to implication graphs  $M$  pointwise.

When we are calculating the implication graph  $M$  of a set  $R$  of feasible outputs, our strategy will be to populate  $M$  with  $\perp$ s initially, and to fill in entries so as to preserve the key invariant  $M \preceq \alpha(R)$ , which says that every 0 and 1 entry in  $M$  accurately describes  $R$ .

The first entries that we can make in  $M$  are the trivial ones saying that, for all literals  $i \in \hat{I}_N$ ,  $M_{ii} = 1$  and  $M_{\bar{i}\bar{i}} = 0$ . ( $M_{\bar{i}\bar{i}} = 1$  would imply that  $i$  is *Zero*.)

We can also fill in a large number of entries based on the random executions and STP counterexamples described above. Suppose that we have found a feasible output where bits  $i$  and  $j$  are 0 and 1, respectively. Then we can conclude that



Figure 4.2: A partially-known implication graph

$M_{ji} = 0$  and  $M_{\bar{i}\bar{j}} = 0$ . Indeed, every combination of values for bits  $i$  and  $j$  allows us to deduce two 0s in  $M$ . Hence, a single feasible output lets us fill in *one fourth* of the nontrivial entries of  $M$ . (If there are  $n$  non-fixed bits, then  $M$  has  $4n^2 - 4n$  nontrivial entries, and a feasible output lets us fill in  $n^2 - n$  of them.) Additional feasible outputs let us fill in a variable number of additional entries, depending on the particular patterns of bits that they exhibit.

Each remaining entry of  $M$  could of course be filled in by an STP query, but we can do better by taking advantage of the *dependencies* among the entries. Consider the partially-known implication graph in Figure 4.2, where the dashed edges denote edges whose existence/non-existence is unknown. If we consider the unknown edge from  $a$  to  $c$ , we can easily deduce that it *must* exist, by transitivity. More interestingly, we can also deduce that the unknown edge from  $c$  to  $a$  must *not* exist. For an edge from  $c$  to  $a$  would by transitivity imply also an edge from  $c$  to  $d$ , contradicting the known fact that there is no such edge.

---

**Algorithm 1:** DeductiveClosure1 algorithm

---

**Input** : implication graph  $M$  over non-fixed bits  $\hat{I}_N$   
**Output**:  $M$  with additional 1s implied by transitivity

```

for  $k \in \hat{I}_N$  do
  for  $i \in \hat{I}_N$  do
    for  $j \in \hat{I}_N$  do
      if  $M_{ik} = 1 \wedge M_{kj} = 1$  then
         $M_{ij} \leftarrow 1$ ;

```

---



---

**Algorithm 2:** DeductiveClosure2 algorithm

---

**Input** : transitive implication graph  $M$  over non-fixed bits  $\hat{I}_N$

**Output:**  $M$  with additional 0s implied by transitivity

```
for  $k \in \hat{I}_N$  do
  for  $i \in \hat{I}_N$  do
    for  $j \in \hat{I}_N$  do
      if  $(M_{ik} = 0 \wedge M_{jk} = 1) \vee (M_{ki} = 1 \wedge M_{kj} = 0)$  then
         $M_{ij} \leftarrow 0$ ;
```

---

These insights lead to two algorithms for *deductive closure*. The first, shown in Algorithm 1, is Warshall's classic transitive closure algorithm. The second, shown in Algorithm 2, takes as input a transitive implication graph  $M$  and deduces additional 0 entries. To get some intuition, notice that when the first disjunct of the **if** holds, then we have  $i \not\rightarrow k$  and  $j \rightarrow k$ . Hence,  $i \rightarrow j$  is impossible, as this would yield  $i \rightarrow k$  by transitivity.

*DeductiveClosure1* has one interesting property: after any number of iterations of its outer loop, if  $M_{ij} = 1$  then there must already be a path from  $i$  to  $j$  in the initial  $M$ .

**Lemma 4.3.1** *Let  $M^n$  be the adjacency matrix after  $n$  iterations of the outer loop of *DeductiveClosure1*, if  $M_{ij}^n = 1$  then there is a path from  $i$  to  $j$  in  $M^0$ .*

*Proof.* We prove the lemma by induction on  $n$ .

The base case  $n = 0$  is true since  $M^0$  is just the initial adjacency matrix.

Let's assume that the lemma is true for the case  $n = l - 1$  and let  $k$  be the literal selected by the outer loop in the  $l^{th}$  iteration. Then,  $M_{ij}^l$  is set to 1 by the  $l^{th}$  iteration only if  $M_{ik}^{l-1} = 1$  and  $M_{kj}^{l-1} = 1$ . By the induction assumption, there are a path from  $i$  to  $k$  and a path from  $k$  to  $j$  in  $M^0$ . By the transitivity, there must be a path from  $i$  to  $j$  in  $M^0$ . Hence, if  $M_{ij}^l = 1$  then there must be a path from  $i$

to  $j$  in  $M^0$ . Thus, the lemma holds for the case  $n = l$ . From the base step and the induction step, by the principle of mathematical induction, we can conclude that the lemma holds for all  $n$ .

□

It is trivial to show that the lemma holds after the completion of *DeductiveClosure1*. *DeductiveClosure2* also has a similar property for the 0 entries. After any number of iterations of its outer loop, if  $M_{ij} = 0$  there exists  $u, v$  in the initial  $M$  such that there is path from  $u$  to  $i$ , and a path from  $j$  to  $v$ , but no path from  $u$  to  $v$ .

**Lemma 4.3.2** *Let  $M^n$  be the adjacency matrix after  $n$  iterations of the outer loop of *DeductiveClosure2*, if  $M_{ij}^n = 0$  then there exists  $u, v$  in  $M^0$  such that  $M_{ui}^0 = 1$ ,  $M_{jv}^0 = 1$ , and  $M_{uv}^0 = 0$ .*

*Proof.* The base case  $k = 0$  is true since  $M^0$  is just the initial adjacency matrix and we can have  $u = i$  and  $v = j$  for each  $M_{ij}^0 = 0$ .

Let's assume that the lemma is true for the case  $n = l - 1$  and let  $k$  be the literal selected by the outer loop in the  $l^{th}$  iteration. Then,  $M_{ij}^l$  is set to 0 by the  $l^{th}$  iteration only if either (a)  $M_{kj}^{l-1} = 0 \wedge M_{ki}^{l-1} = 1$  or (b)  $M_{ik}^{l-1} = 0 \wedge M_{jk}^{l-1} = 1$ . In the case (a), by the induction assumption, we know that there must exist two literals  $u, v$  where  $M_{uv}^0 = 0$ ,  $M_{uk}^0 = 1$ ,  $M_{jv}^0 = 1$ . This means that  $M_{ui}^0 = 1$ . In the case (b), by the induction assumption, we know that there must exist two literals  $u, v$  where  $M_{uv}^0 = 0$ ,  $M_{ui}^0 = 1$ ,  $M_{kv}^0 = 1$ . This means that  $M_{jv}^0 = 1$ .

Hence, if  $M_{ij}^l = 0$  then there exists  $u, v$  in  $M^0$  such that  $M_{ui}^0 = 1$ ,  $M_{jv}^0 = 1$ , and  $M_{uv}^0 = 0$ . Thus, the lemma holds for the case  $n = l$ . From the base step and the induction step, by the principle of mathematical induction, we can conclude that the lemma holds for all  $n$ . □

This lemma also holds after the completion of *DeductiveClosure2*( $M$ ). Let  $DC(M)$  denote the result of calling the procedure of *DeductiveClosure1*( $M$ ) followed by *DeductiveClosure2*( $M$ ). The *soundness* of  $DC$  is given by the following theorem, which says that if implication graph  $M$  is correct for some set  $R$  of feasible outputs, then so is  $DC(M)$ ; this implies that both the 0 and 1 entries filled in by  $DC$  are correct for  $R$ .

**Theorem 4.3.3** *For all  $M$  and  $R$ , if  $M \preceq \alpha(R)$ , then  $DC(M) \preceq \alpha(R)$ .*

*Proof.* Suppose  $DC(M)_{ij} = 1$ . By the Lemma 4.3.1, there is a path in  $M$ :  $i \rightarrow i_1 \rightarrow \dots \rightarrow i_m \rightarrow j$  ( $m \geq 0$ ). Therefore,  $\forall \rho \in R$ ,  $\rho(i) \leq \rho(i_1) \leq \dots \leq \rho(i_m) \leq \rho(j)$  since  $M$  is correct for  $R$ . Hence,  $DC(M)_{ij} = \alpha(R)_{ij}$ .

Now suppose  $DC(M)_{ij} = 0$ . By the Lemma 4.3.2, there exists  $u, v$  in  $M$  such that  $M_{uv} = 0$ ,  $M_{ui} = 1$ , and  $M_{jv} = 1$ . Hence,  $\exists \rho \in R$  such that  $\rho(i) \geq \rho(u) > \rho(v) \geq \rho(j)$  since  $M$  is correct for  $R$ . Hence,  $DC(M)_{ij} = \alpha(R)_{ij}$ .  $\square$

We moreover conjecture that  $DC$  satisfies a *completeness* property saying that it fills in *as many* 1 and 0 entries as can be done without violating soundness. Nevertheless, we have yet not proved this.

## 4.4 Computing Implication Graphs Efficiently

Our algorithm for building the implication graph  $M$  is shown as Algorithm 3. Notice that the selection of the  $\perp$  entry to fill in next is unspecified—our current implementation does this randomly. Also,  $DC$  is invoked each time a new entry of  $M$  is found, to see whether any additional entries can be deduced. However, *DeductiveClosure1* is invoked only if a new 1 entry was found, since otherwise it cannot possibly deduce anything new. Note also that the **else** branch corresponds to an invalid STP query, which gives us a new counterexample to exploit.

---

**Algorithm 3:** Compute the implication graph

---

**input** : non-fixed bits  $\hat{I}_N$  for a set  $R$  of feasible outputs  
**output**: implication graph  $M$  representing the two-bit patterns

for all  $i, j \in \hat{I}_N$ ,  $M_{ij} \leftarrow \perp$ ;  
for all  $i \in \hat{I}_N$ ,  $M_{ii} \leftarrow 1$ ,  $M_{i\bar{i}} \leftarrow 0$ ;  
fill in the 0 entries in  $M$  determined by random executions and counterexamples;

**while**  $M$  has an entry with  $\perp$  **do**

- select  $p, q \in \hat{I}_N$  with  $M_{pq} = \perp$ ;
- if** STP query reveals that bit  $p \leq$  bit  $q$  in  $R$  **then**
  - $M_{pq} \leftarrow 1$ ;
  - $M_{q\bar{p}} \leftarrow 1$ ;
  - DeductiveClosure1 ( $M$ );
  - DeductiveClosure2 ( $M$ );
- else**
  - $M_{pq} \leftarrow 0$ ;
  - $M_{q\bar{p}} \leftarrow 0$ ;
  - get counterexample and use it to fill in more 0 entries of  $M$ ;
  - DeductiveClosure2 ( $M$ );

---

To show the correctness of Algorithm 3, note that the initialization of  $M$  establishes  $M \preceq \alpha(R)$ . Assuming that the STP queries are answered correctly, the assignments to  $M_{pq}$  and  $M_{q\bar{p}}$  preserve this invariant, as do the calls to  $DC$ , by Theorem 4.3.3. Hence, the algorithm terminates with  $M = \alpha(R)$ , as desired.

Having calculated the implication graph  $M$ , we next compute the *size* of  $\gamma(M)$  by extracting the inequalities in  $M$  and counting the number of solutions using *Mathematica*'s `SatisfiabilityCount` function. (Here we first compact the inequalities by collapsing the strongly connected components of  $M$  and taking the transitive reduction.) Finally, we compute the maximum leakage as  $\log |\gamma(M)|$ , since (by Theorem 4.1.3) the size of  $\gamma(M)$  is an upper bound on the number of feasible outputs.

## 4.5 Revisiting the Illustrative Example

We revisit the illustrative example in Figure 3.1 using the optimized approach. The optimized approach achieves a notable improvement on performance—57% time reduction. This is achieved in the same computing environment described in Chapter 3. However, the effectiveness of different techniques varied. On this program, random execution can only have very limited benefit, since a randomly-chosen 32-bit value for  $\mathbf{x}$  is highly unlikely to be less than or equal to 64. In all the experiments, at the second line, only the **else** branch are taken. Therefore, the only output that has been produced is 1. This is why it filled just 25% of the non-trivial entries in the implication graph. STP counterexamples, on the other hand, contribute significantly—57% of the non-trivial entries. The two-bit patterns of the outputs in this case are so simple that there is no transitive structure in them. This prevents deductive closure from making any contribution.

## CHAPTER 5

### EXPERIMENTS

In this chapter, we assess the accuracy and efficiency of our two-bit pattern approach by trying it on eleven case studies, most of which come from the recent literature in quantitative information flow analysis. In all the case studies, we assume that  $X$  is the secret input variable,  $Y$  is the output variable, and all the variables are 32-bit unsigned integers. For each case study, both the straightforward technique which determined each two-bit pattern individually through STP queries and the optimized technique described in Chapter 4 are used to compute the two-bit patterns of the feasible outputs.<sup>1</sup> Recall that both techniques are doing the same two-bit pattern analysis, this means that upper bounds on leakage derived from them are exactly the same. Hence, the differences between them are only on efficiency. Also, for all the case studies, their channel capacities, measured by Shannon entropy, min-entropy, or g-leakage coincide since they are deterministic programs. Throughout this chapter, we use the term *capacity* referring to all these channel capacities.

#### 5.1 Sanity Check

Consider the “sanity check” program from [NMS09], where  $Y$  is influenced by  $X$  only when  $X$  is found to be within an acceptable range:

```
if (X < 16)
    Y = base + X;
else
    Y = base;
```

---

<sup>1</sup>Because of the randomness in our new techniques, the new timings using the optimized technique are averages over 10 executions.



interesting two-bit patterns among the  $32 \cdot 31/2 = 496$  pairs of bits. Namely, bits 30 through 4 are all equal to one another, and different from bit 31. Moreover, bits 30 through 4 are all less than or equal to bit 3. Finally, we have  $Or(31,3)$ . In total, we find that 90 pairs have pattern *Free*, and the remaining 406 pairs have pattern *Eq*, *Neq*, *Leq*, or *Or*. Determining these two-bit patterns requires 1552 STP queries and 2040 ms. Finally, `SatisfiabilityCount` requires 1 ms to determine that there are 24 solutions to the bit patterns, implying a capacity of at most  $\log 24 \approx 4.58$  bits, which is close to the actual capacity of 4 bits. Because a large number of literals are in the equivalence relation, deductive closure is very helpful here, since it found 36% of the entries. Overall, the optimized approach gains a time reduction of 90%. In fact, when the random testing and STP counterexamples are disabled, the contribution of deductive closure becomes the dominating force, finding 93% of the entries.

## 5.2 Implicit Flow

Here is a program from [NMS09] that indirectly copies  $X$  to  $Y$  if  $X \leq 6$ ; otherwise, it sets  $Y$  to 0:

```

Y = 0;
if (X == 0) then Y = 0;
else if (X == 1) then Y = 1;
else if (X == 2) then Y = 2;
...
else if (X == 6) then Y = 6;
else Y = 0;

```

Since there are 7 possible outputs, the capacity is  $\log 7 \approx 2.81$  bits.





```
X = (X & 0x00ff00ff) + ((X>>8) & 0x00ff00ff);
Y = (X + (X>>16)) & 0xffff;
```

It has 33 possible outputs. Thus, its capacity is  $\log 33 \approx 5.044$  bits.

The straightforward approach uses 38 STP queries and 184 ms to find that the one-bit patterns are

```
00000000000000000000000000000000*****
```

Among the 6 non-fixed bits, we find (using 50 STP queries and 721 ms) that there are 5 interesting two-bit patterns:  $Nand(5,4)$ ,  $Nand(5,3)$ ,  $Nand(5,2)$ ,  $Nand(5,1)$  and  $Nand(5,0)$ . These patterns have exactly 33 instances; therefore, our bound is exact.

The optimized approach achieves 80% time reduction. Random execution is highly effective in this case. It shows that the last 5 bits of Y are non-fixed, and therefore reduce the one-bit pattern queries from 32 to 27. This turns out to be a big win in time reduction, since the one-bit pattern queries on the last 5 bits are more expensive. Random execution also contributes 80% of the non-trivial entries in the implication graph. The rest of the entries are filled by only a few implication queries and deductive closure.

## 5.4 Mix and Duplicate

Next, we revisit the example (also from [NMS09]) discussed in the Introduction. It combines the two halves of X using XOR, and then duplicates these 16 bits in both the upper and lower halves of Y:

```
Y = ((X >> 16) ^ X) & 0xffff;
Y = Y | Y << 16;
```

Hence, it has  $2^{16} = 65536$  possible outputs, giving a capacity of 16 bits.

In 64 STP queries and 23 ms, the straightforward approach finds that all 32 bits are non-fixed. Then, in 1968 STP queries and 863 ms, it finds that there are 16 *Eq* patterns

$$Eq(31, 15), Eq(30, 14), Eq(29, 13), \dots, Eq(16, 0)$$

and 480 *Free* patterns.

For some reason, `SatisfiabilityCount` took much longer here than in any other case—it took 42 ms to determine that there are 65536 solutions to the bit patterns. We get a capacity of at most  $\log 65536 = 16$  bits, which is again exact.

The optimized approach, on the other hand, only spends 82 ms—reduced time by 91%. The major win came from random execution. The random execution is so effective that it shows that all the bits are non-fixed. Therefore, no one-bit pattern query is needed. It also fills most of the 0 entries in the implication graph.

## 5.5 Masked Copy

This simple program copies the first 16 bits of X into Y, masking out the last 16 bits:

```
Y = X & 0xffff0000;
```

As in the previous example, it has  $2^{16} = 65536$  possible outputs, giving a capacity of 16 bits.

In 48 STP queries and 9 ms, the straightforward approach finds that the one-bit patterns are

```
*****0000000000000000
```

In 480 STP queries and 114 ms, it finds that all two-bit patterns are *Free*. Thus, we get a min-capacity of 16 bits, which is again exact.

In the optimized approach, random execution turns out to be highly effective. It produces enough feasible outputs which manifest all the non-fixed bits and *Free* relations among them. Hence, all of the 0 entries in the implication graph are filled. Only the rightmost 16 bits are left for one-bit pattern queries to find out that they have fixed value 0. Thus, it leaves no room for STP counterexample and deductive closure to make any contribution.

## 5.6 Binary Search

Now we consider a program that uses binary search to leak the first  $b$  bits of  $X$  to  $Y$ :

```
Y = 0;
for (i = 0; i < b; i++) {
    m = 2^(31-i);
    if (Y + m <= X) Y += m;
}
```

We handle the loop by unrolling it completely, precomputing the value of  $m$  at each iteration. When  $b = 16$ , we get the program

```
Y = 0;
if (Y + 2147483648 <= X) Y += 2147483648;
if (Y + 1073741824 <= X) Y += 1073741824;
if (Y + 536870912 <= X) Y += 536870912;
if (Y + 268435456 <= X) Y += 268435456;
if (Y + 134217728 <= X) Y += 134217728;
if (Y + 67108864 <= X) Y += 67108864;
if (Y + 33554432 <= X) Y += 33554432;
```

```

if (Y + 16777216 <= X) Y += 16777216;
if (Y + 8388608 <= X) Y += 8388608;
if (Y + 4194304 <= X) Y += 4194304;
if (Y + 2097152 <= X) Y += 2097152;
if (Y + 1048576 <= X) Y += 1048576;
if (Y + 524288 <= X) Y += 524288;
if (Y + 262144 <= X) Y += 262144;
if (Y + 131072 <= X) Y += 131072;
if (Y + 65536 <= X) Y += 65536;

```

As in the previous example, it has  $2^{16} = 65536$  possible outputs, giving a capacity of 16 bits.

In 48 STP queries and 246 ms, the straightforward approach finds that the one-bit patterns are

```

*****0000000000000000

```

In 480 STP queries and 4220 ms, it finds that all two-bit patterns are *Free*. Thus, we get a capacity of 16 bits, which is again exact.

As in the case of Masked Copy, random execution is highly effective—it usually finds enough feasible outputs to fill in *all* of the entries of  $M$ . Because STP queries on this program are very expensive (after all, it has  $2^{16}$  possible execution paths), avoiding STP queries is very beneficial, since it reduces the analysis time by more than 99%. The only significant time spending comes from 16 one-bit pattern queries needed to confirm that the rightmost 16 bits are fixed with the value 0.

## 5.7 Electronic Purse

Next, we consider the electronic purse program from [BKR09]:

```

Y = 0;
while(X >= 5) {
    X = X - 5;
    Y = Y + 1;
}

```

Here we add the assumption that  $X < 20$ , which means that  $Y$  can range from 0 to 3, giving a capacity of 2 bits.

Again we unrolled the loop. The straightforward approach finds (in a total of 272 ms) that the first 30 bits of  $Y$  must be 0, and the last 2 bits are *Free*, giving a capacity of 2 bits. As in the two previous cases, random execution produces enough feasible outputs which reveal the non-fixed bits and their *Free* relations. However, since one-bit patterns queries dominate the time in both techniques, the overall improvement is modest—42% time reduction.

## 5.8 Sum Query

Here is the sum query from [BKR09]:

```

Y = X1;
Y = Y + X2;
Y = Y + X3;

```

Here we assume that  $X1$ ,  $X2$ , and  $X3$  are each less than 10. This means that there are 28 possible outputs (from 0 to 27) and a capacity of  $\log 28 \approx 4.807$  bits.

In a total of 235 ms, the straightforward approach finds that the first 27 bits of  $Y$  must be 0, and the last 5 bits are *Free*, giving a capacity of  $\log 32 = 5$  bits. As in the case of Electronic Purse, the optimized approach mainly saves the time on

computing two-bit patterns. However, since one-bit pattern queries dominate the time in both techniques, the overall gain in time reduction is only 50%.

## 5.9 Ten Random Outputs

While bit patterns performed quite well in all our previous case studies, we did identify a scenario where they perform very poorly. Consider a family of programs that each have exactly ten possible outputs:

```
if (X == r1) Y = r1;
else if (X == r2) Y = r2;
else if (X == r3) Y = r3;
...
else if (X = r9) Y = r9;
else Y = r10;
```

Suppose we create such a program by generating distinct 32-bit values `r1` through `r10`, uniformly and independently. Intuitively, we would expect that the one-bit patterns for `Y` will all be *Non-fixed*, and the two-bit patterns will overwhelmingly be *Free*, leading us to greatly overestimate the capacity.

We confirmed this intuition experimentally by creating 20 such programs and finding the average result of our bit-pattern analysis. On average, the bit patterns had over 400,000 solutions, giving a capacity of 18.645 bits, which far exceeds the actual capacity of  $\log 10 \approx 3.322$  bits. While the inaccuracy here is striking, practical programs would seem unlikely to produce such completely unrelated outputs; therefore, it is not clear whether this example represents a significant limitation of two-bit patterns.

Program	Min-capacity	Upper bound
Illustrative example	4.087	4.087
Sanity check, base=0x00001000	4.	4.
Sanity check, base=0x7fffffffa	4.	4.585
Implicit flow	2.807	3.
Population count	5.044	5.044
Mix and duplicate	16.	16.
Masked copy	16.	16.
Binary search, b=16	16.	16.
Electronic purse	2.	2.
Sum query	4.807	5.
Ten random outputs (average)	3.322	18.645

Table 5.1: Accuracy of our upper bounds

It is worthwhile to note that STP counterexamples are highly effective in this case. This is due to the fact that there are only ten feasible outputs. Even one or two counter examples can fill many entries in the implication graph, and thus, save STP implication queries. In fact, the optimized approach only needs 67 implication queries in average to compute the implication graph, while the straightforward approach needs about 1900 queries on average. The overall time reduction is 94%.

## 5.10 Summary

We present our results on the case studies in six tables. Table 5.1 compares the channel capacities with our upper bounds. Table 5.2 shows our times (in milliseconds) spent by the straightforward technique to compute one-bit patterns, two-bit patterns, and to count the number of solutions to the bit patterns.<sup>2</sup> Table 5.3 shows the corresponding results obtained by the optimized technique. Table 5.4 presents details of bit-pattern analyses: the number of STP queries required by

---

<sup>2</sup>The times reported here for our analysis on the straightforward technique are faster than those reported in [MS11], because we have redone our old experiments on a faster computer: a 2.3 GHz Intel Core i3-2310M.



Program	One-bit patterns	Two-bit patterns	#SAT
Illustrative example	1072	1747	<1
Sanity check, base=0x00001000	33	21	<1
Sanity check, base=0x7ffffffa	66	2040	1
Implicit flow	12	16	<1
Population count	184	721	<1
Mix and duplicate	23	863	42
Masked copy	9	114	<1
Binary search, b=16	246	4220	<1
Electronic purse	210	62	<1
Sum query	135	100	<1
Ten random outputs (average)	91	3460	14

Table 5.2: Times in ms to calculate our bounds using the straightforward technique: #SAT = times for SatisfiabilityCount

Program	Random execution	One-bit patterns	Two-bit patterns	#SAT
Illustrative example	1	805	398	<1
Sanity check, base=0x00001000	1	28	7	<1
Sanity check, base=0x7ffffffa	2	4	197	<1
Implicit flow	1	7	3	<1
Population count	4	79	96	<1
Mix and duplicate	2	0	80	<1
Masked copy	2	2	6	<1
Binary search, b=16	1	21	2	<1
Electronic purse	4	153	0	<1
Sum query	1	113	4	<1
Ten random outputs (average)	1	7	216	<1

Table 5.3: Times in ms to calculate our bounds using the optimized technique: #SAT = times for SatisfiabilityCount

Program	1-bit	2-bit	1-bit*	=>	# of <i>Non-fixed</i> bits	# of <i>Free</i> pairs
Illustrative example	38	32	30	13	5	6
Sanity check, <code>base=0x00001000</code>	37	24	32	6	4	6
Sanity check, <code>base=0x7ffffffa</code>	64	1552	3	75	32	90
Implicit flow	35	12	32	4	3	3
Population count	38	50	27	6	6	10
Mix and duplicate	64	1968	0	32	32	480
Masked copy	48	480	16	0	16	120
Binary search, <code>b=16</code>	48	480	16	0	16	120
Electronic purse	34	4	30	0	2	1
Sum query	37	40	27	1	5	10
Ten random outputs (average)	64	1843	4	67	32	384

Table 5.4: Details of our bit pattern analyses: `1-bit`=the number of one-bit queries by the straightforward technique, `2-bit`=the number of two-bit queries by the straightforward technique, `1-bit*`=the number of one-bit queries by the optimized technique, `=>`=the number of implication queries by the optimized technique

both techniques, the number of *Non-fixed* bits, and the number of *Free* pairs. Table 5.5 compares the times to do two-bit pattern analysis using both techniques. As can be seen, two-bit patterns usually allow quite accurate bounds to be calculated. Even the straightforward technique only takes a few seconds to compute two-bit patterns. The optimized technique based on implication graphs, random execution, STP counterexamples, and deductive closure allows two-bit pattern analysis to be done more efficiently. The times are reduced in all 11 case studies, by an average of 72%; in five cases, the reduction exceeds 90%. However, the reductions are quite variable, ranging from 33% to over 99%.

One way to understand the varied effectiveness of the different techniques that we are using is to consider what percentage of the non-trivial entries of the implication graph  $M$  are found by random execution, by STP counterexamples, by STP queries, and by deductive closure. Table 5.6 gives this information. It shows that the percentage of entries found by random execution varies greatly, from as little as 25% to as much as 100%. STP counterexamples contribute between 0% and 70%.

Program	OTime	NTime	Reduction
Illustrative example	2819	1204	57%
Sanity check, base=0x00001000	54	36	33%
Sanity check, base=0x7ffffffa	2106	203	90%
Implicit flow	28	11	61%
Population count	905	179	80%
Mix and duplicate	886	82	91%
Masked copy	123	10	92%
Binary search, b=16	4466	24	99%
Electronic purse	272	157	42%
Sum query	235	118	50%
Ten random outputs (average)	3551	224	94%

Table 5.5: Old and new times in ms to do two-bit pattern analysis: OTime=old time for two-bit pattern analysis using the straightforward technique, NTime=new time for two-bit pattern analysis using the optimized technique, Reduction= $1 - \text{NTime}/\text{OTime}$

As for deductive closure, it contributes in only two of the case studies, and this is mostly a function of the fact that random execution and STP counterexamples often fill in almost all of  $M$ . To see what contribution deductive closure *could* have made, we repeated the experiments with random execution and STP counterexamples disabled. As seen in P4\*, deductive closure could have made a significant contribution in five of the case studies.

To further test the optimized approach’s capability of speeding up the performance, we contrived the following program:

```

Y = X;
if ((Y & 0xffff) != 0)
    Y = Y | 0xffff0000;

```

An interesting property of this program is that each of the leftmost 16 bits of  $Y$  is greater than or equal to each of the rightmost 16 bits of  $Y$ , and there are no other constraints. This poses a challenge to the optimized approach due to a great amount of “arrow” among the bits, 256 in total. However, the optimized approach

Program	P1	P2	P3	P4	P3*	P4*
Illustrative example	25	57	18	0	67	33
Sanity check, base=0x00001000	25	50	25	0	100	0
Sanity check, base=0x7ffffffa	25	35	4	36	7	93
Implicit flow	25	58	17	0	100	0
Population count	80	10	10	0	88	12
Mix and duplicate	98	2	0	0	57	43
Masked copy	100	0	0	0	100	0
Binary search, b=16	100	0	0	0	100	0
Electronic purse	100	0	0	0	100	0
Sum query	97	3	0	0	100	0
Ten random outputs (average)	25	70	3	2	53	47

Table 5.6: Average percentage of  $M$  found by different techniques: P1=entries found by random execution, P2=entries found by STP counterexamples, P3=entries found by STP implication queries, P4=entries found by deductive closure. P3\* and P4\* are the same as P3 and P4, but with random execution and STP counterexamples disabled.

still achieved a modest time reduction. It took the straightforward approach 64 STP queries and 23 ms to compute the one-bit patterns, 1728 STP queries and 809 ms to compute the two-bit patterns, With the help of random executions and STP counterexamples, the optimized approach only spent one STP and 1 ms to compute the one-bit patterns, 272 STP queries and 543 ms to build the implication graph. In fact, random executions and STP counterexamples are so effective in this case that they largely eliminate the need to determine the *Free* patterns within both the leftmost 16 bits and the rightmost 16 bits. As in many previous case studies, the deductive closure was prevented from contributing anything.

## CHAPTER 6

### RELATED WORK

This chapter discusses the relevant literature in fields of contemporary quantitative information flow, constraint solving, and abstract interpretation. These works are compared with two-bit patterns from both theoretical and implementational perspectives.

#### 6.1 Calculating Quantitative Information Flow

Calculating quantitative information flow is a challenging problem, as shown for example by the negative computational complexity results given in [YT10]. The paper shows that the problem of *comparing* the min-entropy leakage of two loop-free boolean programs is  $\#P$ -hard; they give a reduction showing that one can count the number of satisfying assignments of a boolean proposition (which is  $\#P$ -complete) via a polynomial number of such comparison queries. Nevertheless, this is an area that is now seeing a great deal of work, both in the context of imperative programs [BKR09, KR10, NMS09, PMTP12, HM10, APvRS10, CCG10] and real world softwares [HM10, KMO12]. Our focus here will be on techniques for calculating channel capacity under min-entropy and Shannon entropy of deterministic programs.

One work, which is similar to our two-bit patterns, is the paper by Newsome, McCamant, and Song [NMS09], which estimates Shannon capacity of deterministic x86 binaries. Interestingly, their motivation is quantitative *integrity*, looking at the amount of *influence* the untrusted input can have on the trusted output. While they actually use *Shannon capacity*, by Theorem 2.5.2 above this coincides with min-capacity, and amounts simply to counting the number of possible output values. They estimate this through various heuristics, using STP to check whether a particular output is possible or not, and whether an *interval* contains any possible

outputs. Using binary search, they try to find which intervals in the range of  $Y$  contain possible outputs and which do not. When they find that an interval contains at least one possible output, they sometimes use random sampling to estimate the *density* of possible outputs within it.

While these techniques often work well, they do poorly on programs like `Mix` and `Duplicate`, whose outputs are sparse and scattered. In that program, interval analysis gives no useful information, and sampling cannot give accurate estimates, since it has only  $2^{16}$  possible outputs (out of  $2^{32}$  32-bit integers). For cases like this, they rely complementarily on a probabilistic `#SAT` algorithm to estimate directly the number of possible output values. However, this is expensive, taking up to 30 seconds in some cases.

We believe that our case studies show that two-bit patterns offer a useful intermediary for leakage calculation for two reasons. First, two-bit patterns can be calculated rather quickly and they usually provide quite accurate upper bounds on the min-capacity. Second, counting the number of solutions to the bit patterns, using `SatisfiabilityCount`, seems to be much faster than trying to count the number of solutions to the whole program model.

A quite different approach to approximating leakage is given in the recent work of Köpf and Rybalchenko [KR10], which uses statistical sampling to estimate the *mutual-information leakage* of a deterministic imperative program from input  $X$  to output  $Y$ , under a uniform *a priori* distribution. While they present the technique in terms of estimating  $H(X|Y)$ , it is easier to remember that the mutual-information leakage is just  $H(Y)$ . They assume that for each possible output value  $y$ , we can estimate its probability (by estimating the number of values of  $X$  that lead to  $y$ ). Then they observe that  $H(Y)$  is the expected value of  $\log \frac{1}{P(y)}$ , where  $y$  is a sampled

output value:

$$\text{mutual information leakage} = H(Y) = E \left( \log \frac{1}{P(y)} \right).$$

With  $n$  samples,  $y_1, y_2, \dots, y_n$ , we find that  $H(Y)$  is also the expected value of  $\frac{1}{n} \sum_{i=1}^n \log \frac{1}{P(y_i)}$ . Crucially, the *variance* of this last random variable is small relative to the number of possible inputs, which means that the Chebyshev inequality can be used to give good bounds on the accuracy of the estimate for not-too-large values of  $n$ . However, it is not clear whether a similar technique can be used to calculate min-entropy leakage.

Another model-checking based work in this area is the recent paper by Heusser and Malacaria [HM10]. While they actually focus on the *Shannon capacity* of deterministic programs, again this essentially amounts to counting the number of possible output values. Rather than attempting to *calculate* the capacity, they instead focus on solving the problem of *testing* whether the capacity is at least some threshold. Their approach is to test whether a program  $P$  can produce at least  $b$  different outputs by forming a new program  $P'$  which runs  $P$  independently  $b$  times on nondeterministically-chosen inputs. They then check (using the bounded model checker CBMC) whether there is a path to a state where all  $b$  outputs are distinct. In this way, they determine whether  $P$ 's capacity is at least  $\log b$  bits. They model both *high* inputs, which are confidential, and *low* inputs, which are controlled by users. In the particular case of low input, all runs of  $P$  in  $P'$  have varying high input, but the same low input. Therefore, the model checker essentially determines whether there exists a value of  $l$  under which  $P$ 's capacity is at least  $\log b$  bits.

While the technique yields interesting results on leakage in real Linux kernel vulnerabilities, the time taken by this method grows very quickly with  $b$ . Based on

their experimental timings, it seems that one cannot go very much above  $b = 128$ ; checking with  $b = 2^{20}$  (corresponding to a 20-bit capacity) would appear infeasible.

Abstract interpretation has also been considered as an approach to measure information leakage in large programs. Köpf, Mauborgne, and Ochoa [KMO12] develop an abstract interpretation for bounding capacity, and use it to show bounds on cache leaks in implementations of the AES cryptosystem.

One closely related work on quantitative information flow analysis is the symbolic quantitative information flow (SQIF) by Phan, Malacaria, Tkachuk, and Păsăreanu [PMT12]. They also tackle the problem of counting the feasible outputs of a deterministic program, but with a different strategy. Unlike two-bit patterns which provide an approximation on the set of feasible states of the output, SQIF essentially describes precisely all of the feasible program outputs using a tree-based symbolic representation. For a program  $P$  with a  $k$ -bit output  $Y$ ,  $Y$  is viewed as a bit vector:  $y_{k-1}y_{k-2}\dots y_0$ . Its feasible states can be represented by a binary tree. The tree has a root node  $y_0$ . For every node  $y_i$ , the left path represents  $\neg y_i$ , and the right path represents  $y_i$ . The succession of the nodes in the tree follows the strict order of  $y_0 > y_2 > \dots > y_{k-1}$ . For example, suppose that the program has 3 feasible outputs,  $\{000, 100, 110\}$ , where we index the 3 bit positions from 2 down to 0. These outputs can be represented by the binary tree in Figure 6.1. One obvious benefit of this representation is that counting the number of feasible outputs becomes straightforward. It is reduced to counting the number of leaf nodes.

Given a deterministic program with a *high* input and a *low* output, its symbolic representation of the feasible output states can be computed by a recursive method based on model checking tool like Java Pathfinder (JPF) [JPF]. To illustrate the technique, let's revisit the case study of sanity checker in Chapter 5.

```
base = 8;
```



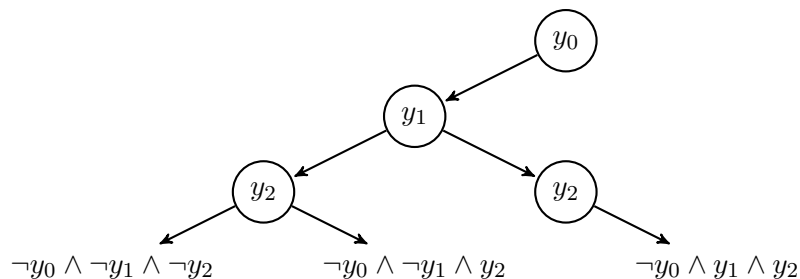


Figure 6.1: The symbolic representation for  $\{000, 100, 110\}$

```

if (X < 16)
    Y = base + X;
else
    Y = base;

```

Assuming that all the variables are 32-bit integers, and the variable `base` has an initial value 8,  $Y$  can only have integer values from 8 to 23. The procedure starts with the lowest bit  $y_0$ . To test whether  $y_0$  can be true, Java Pathfinder checks the validity of  $y_0$  by using the following assertion:

```
assert!y0;
```

In this example, the model checker would return *False* since all odd values from 9 to 23 are the outputs where  $y_0$  is *True*. The procedure then proceeds recursively, first on the path of  $y_0$  and to the next level with a path condition  $pc = y_0$  as the passing argument. From there, it repeats the same check on  $pc \wedge y_1$ . For each node  $y_i$ , the recursion checks the satisfiability of the right path ( $y_i$ ) first and then the left path ( $\neg y_i$ ). The procedure continues until it hits the deepest level of the tree (the 32th level in this case) or the checking fails. A trace of the state space exploration is described in Figure 6.2. The procedure takes its progression on the path of successive right paths, until the 5th level where  $pc = y_0 \wedge y_1 \wedge y_2 \wedge y_3$  and the testing on the

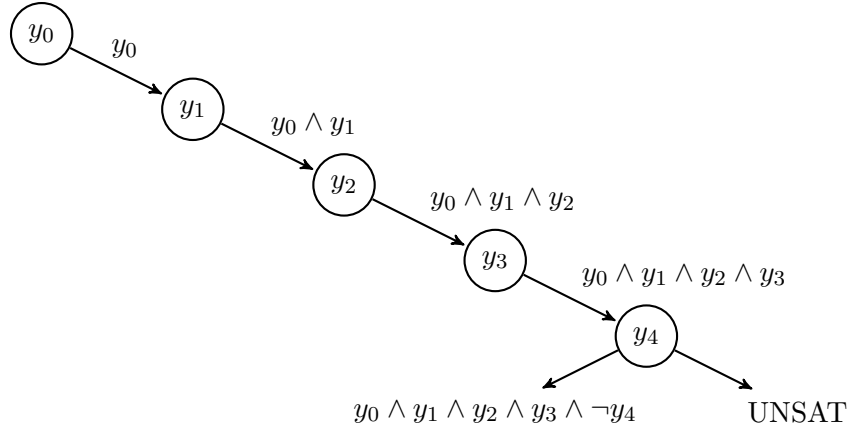


Figure 6.2: A SQIF state exploration trace for sanity check program

satisfiability of  $pc \wedge y_4$  fails. Hence, it takes on the path of  $\neg y_4$ ; there  $pc \wedge \neg y_4$  is found to be satisfiable. From the 6th level, only the path of  $\neg y_i$  is satisfiable until  $i = 32$ . A full path to the feasible output 00000000000000000000000000001111 (15) is found. At the end of the procedure, we have all the feasible outputs from 8 to 23; which means that the program has a Shannon/Min capacity of 4 bits.

In [PMTP12], they also makes a comparison between SQIF and two-bit patterns on a few case studies. In the case of *ten random outputs*, the SQIF triumphs over our approach:

```

if (X == r1) Y = r1;
else if (X == r2) Y = r2;
else if (X == r3) Y = r3;
...
else if (X = r9) Y = r9;
else Y = r10;
  
```

According to their experimentation report, when r1 to r9 are uniformly distributed, SQIF always finds exactly 10 outputs in about 1 second. Recall our experiment in Chapter 5; two-bit pattern analysis takes about 224 milliseconds with a result of

18.645 bits in average. However, in the case studies which have a large number of feasible outputs, it is doubtful that SQIF can compute them efficiently.

## 6.2 Binary Implication Graph

The binary implication graph can be used to facilitate conjunctive normal form (CNF) simplification. One example of this effort is Marijn J. H. Heule, Matti Järvisalo and Armin Biere’s work on efficient CNF simplification based on binary implication graphs [HJB11]. They introduce an efficient method to “unhide” various redundancies in a CNF formula using the time stamping information in the binary implication graphs, derived from the size 2 clauses in the CNF. In particular, they are interested in hidden literals and hidden tautologies. Hidden literals are the kind of literals which can be removed from a clause without affecting the set of satisfying assignments. Hidden tautologies are the kind of clauses which can be removed from the conjunction without affecting the set of satisfying assignments. Knowing the implications among the literals can facilitate the detection of these redundancies. For instance, a literal  $l$  in a clause  $C$  is a hidden literal, if the implication  $l \rightarrow l'$  holds and  $l' \in C$ . In this context,  $l$  is redundant and can be removed from  $C$ . A clause  $C$  is a hidden tautology, if the implication  $\bar{l} \rightarrow l'$  holds and both  $l$  and  $l'$  are in  $C$ .  $C$  is redundant, since either  $l$  or  $l'$  must be true. Consider the following formula from [HJB11]:

$$E = (\bar{a} \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{b} \vee d) \wedge (\bar{b} \vee e) \wedge (\bar{c} \vee f) \wedge (\bar{d} \vee f) \wedge (\bar{f} \vee h) \wedge (\bar{g} \vee f) \wedge (\bar{g} \vee h) \wedge (\bar{a} \vee \bar{e} \vee h) \wedge (\bar{b} \vee \bar{c} \vee h) \wedge (a \vee b \vee c \vee d \vee e \vee f \vee g \vee h).$$

The implicaton graph derived from the size 2 clauses in  $E$ , denoted by  $\text{IG}(E)$ , is presented in Figure 6.3. It has five root nodes (no incoming arcs):  $a, b, \bar{e}, g$ , and  $\bar{h}$ .

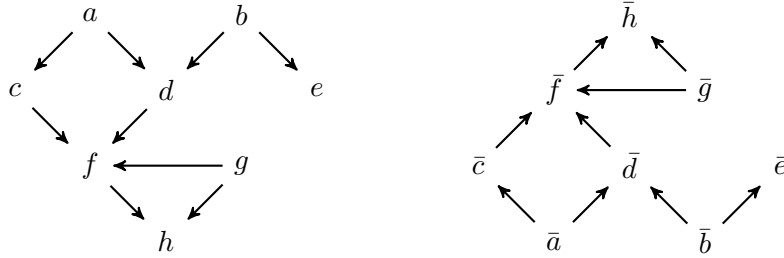


Figure 6.3: An implication graph derived from  $E$  (reprinted from [HJB11])

The formula has redundant clauses and literals. The clauses  $(\bar{a} \vee \bar{e} \vee h)$ ,  $(\bar{g} \vee h)$ , and  $(\bar{b} \vee \bar{c} \vee h)$  are hidden tautologies. In the last clause, all literals except  $e$  and  $h$  are hidden.

The authors use a depth first search (DFS) procedure to associate “time stamps” to literals (nodes) in  $\text{IG}(E)$ . A “time stamping” for a node  $v$  has a *discovered time* and a *finished time*, denoted by  $\text{dsc}(v)$  and  $\text{fin}(v)$ , respectively. The discovered time for a literal is the first time it is encountered during search, and the finished time is the last time it is encountered during search. This “time stamping” procedure is applied first to each root in  $\text{IG}(E)$ , then to the literals which have not yet been visited. The result is a forest of DFS trees with discovered-finished intervals,  $[\text{dsc}(v), \text{fin}(v)]$ , attached to each literal as presented in Figure 6.4. Dashed lines here represent implications in  $\text{IG}(E)$  which are not used to set the time stamps. According to the “parenthesis theorem”, for two nodes  $u$  and  $v$ ,  $v$  is a descendant of  $u$  in the DFS tree if and only if the time stamp interval of  $u$  contains the time stamp interval of  $v$ . In other words,  $\text{dsc}(v) > \text{dsc}(u)$  and  $\text{fin}(u) > \text{fin}(v)$ . Given the time stamps, these conditions can be verified in constant time.

With these time stamps, both hidden literals and hidden tautologies can be detected by simple comparison procedures. For instance, a simple comparison among the literals in the last clause of  $E$  can quickly discover that  $\text{dsc}(c) > \text{dsc}(a)$  and  $\text{fin}(c) < \text{fin}(a)$ . This means that  $a \rightarrow c$ . Hence,  $a$  can be removed from the clause.  $b$  can

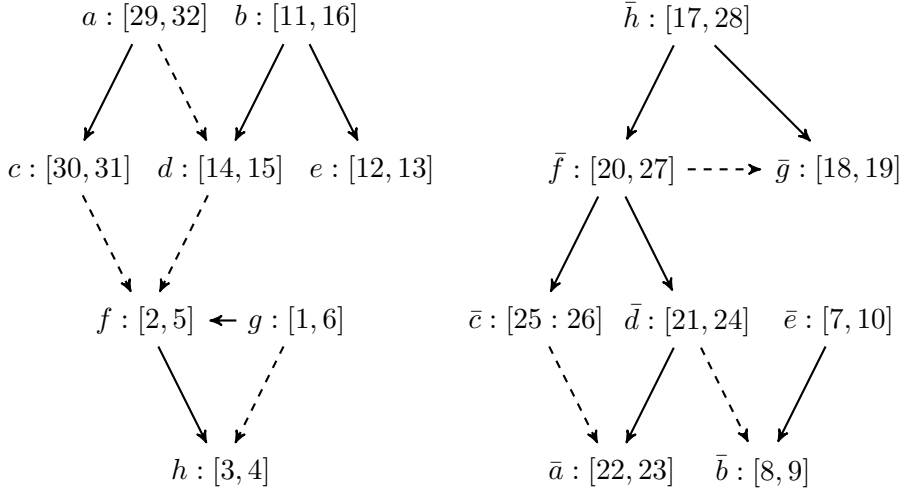


Figure 6.4: Stamp times in  $\text{IG}(E)$  (reprinted from [HJB11])

also be removed from the clause, since  $\text{dsc}(e) > \text{dsc}(b)$  and  $\text{fin}(e) < \text{fin}(b)$ . The time stamps also facilitate the checking of hidden tautologies.  $(\bar{a} \vee \bar{e} \vee h)$  in  $E$ , for instance, can be quickly discovered as a tautology, since the time stamp of  $\bar{h}$  contains the time stamp of  $\bar{a}$ , which means that  $\bar{h} \rightarrow \bar{a}$ .

They also propose an advanced version of DFS time stamping procedure, which can perform additional simplifications, such as transitive reduction and discovering equivalence relations among literals on-the-fly during the stamping procedure. Experiments on real-world SAT competition benchmarks show that *unhiding* based on time stamping notably improves the SAT solving.

Conceivably, the techniques introduced in the above work could improve the efficiency of two-bit pattern computing. Instead of fully relying on STP solver, we can convert the target program into a CNF formula. Then we can use the time stamping information in the implication graph derived from the size 2 clauses to cheaply determine many bit patterns. Hence, we may reduce the number of necessary STP queries.

### 6.3 Unit-Two Variable Per Inequality Constraints

Two-bit patterns is strongly similar to Unit-Two Variables Per Inequality (UTVPI) constraints. A UTVPI constraint takes the form  $a.x+b.y \leq d$  where  $a, b \in \{-1, 0, 1\}$ . The theory of UTVPI has been studied by both decision procedure community and abstract interpretation community. One abstract domain, which closely resembles our works on two-bit patterns and the implication graphs, is Miné’s octagon abstract domain[Min01]. The octagon domain is a practical representation of invariants of the form  $\pm x \pm y \leq d$ , where  $x$  and  $y$  are numerical variables and  $d$  is a numerical constant. As our work on two-bit patterns, the set of numerical variable  $\mathcal{V} = \{V_1, \dots, V_n\}$  is also extended into  $\mathcal{V}' = \{V'_1, \dots, V'_{2n}\}$  with each variable  $V_i$  having both a positive variable  $V'_{2i-1}$  representing  $V_i$  and a negative variable  $V'_{2i}$  representing  $-V_i$ . Every constraint over  $\mathcal{V}$  can be encoded as an equivalent constraint over  $\mathcal{V}'$ :

the constraint	is represented as
$V_i - V_j \leq c \ (i \neq j)$	$V'_{2i-1} - V'_{2j-1} \leq c, V'_{2j} - V'_{2i} \leq c$
$V_i + V_j \leq c \ (i \neq j)$	$V'_{2i-1} - V'_{2j} \leq c, V'_{2j-1} - V'_{2i} \leq c$
$-V_i - V_j \leq c \ (i \neq j)$	$V'_{2i} - V'_{2j-1} \leq c, V'_{2j} - V'_{2i-1} \leq c$
$V_i \leq c$	$V'_{2i-1} - V'_{2i} \leq 2c$
$V_i \geq c$	$V'_{2i} - V'_{2i-1} \leq -2c$

A set of octagonal constraints  $C$  over  $\mathcal{V}'$  can be represented as a *potential graph* which is very similar to the implication graph. A potential graph has nodes in  $\mathcal{V}'$ . For each pair of variables  $(V_i, V_j) \in \mathcal{V}'$ , there is an arc from  $V_i$  to  $V_j$  with weight  $c$ , if the constraint  $V_i - V_j \leq c$  is in  $C$ . The potential graphs can be described by *difference-bound matrices(DBMs)*. Given a set of constraints  $C$  over  $\mathcal{V}'$ , its corresponding DBM  $\mathbf{m}$  is  $2N \times 2N$  matrix ( $N = |\mathcal{V}'|$ ) with the following definition:

$$\mathbf{m}_{ij} = \begin{cases} c, & \text{if } (v_j - v_i \leq c) \in C \\ \infty, & \text{otherwise.} \end{cases}$$

Miné proposes a normalization algorithm inspired by Floyd-Warshall shortest-path closure algorithm to compute the *strong closure* of a given potential graph and build abstract transfer functions with a cubic time cost. Incorporated into the famous Astrée static analyzer, the octagon domain has achieved remarkable success on proving the safety properties of large critical softwares.

The decision procedure community is also interested in UTVPI constraints from the satisfiability perspective: whether a given set of UTVPI constraints is satisfiable. Jaffar, Maher, Stuckey and Yap proposed the first algorithm to decide whether a conjunction of UTVPI constraints is satisfiable in both real and integer domains [JMSY94]. The algorithm is based upon the following inference rules:

$$\frac{a.x + b.y \leq c \quad -a.x + b'.z \leq d}{b.y + b'.z \leq c + d} \quad (\text{TRANSITIVE})$$

$$\frac{a.x + b.y \leq c \quad a.x - b.y \leq d \quad a \in \{-1, 1\}}{a.x \leq \lfloor (c + d)/2 \rfloor} \quad (\text{TIGHTENING})$$

Given a set of UTVPI constraints  $C$ , the decision procedure incrementally computes its closure with respect to the inference rules.  $C$  is unsatisfiable if and only if its closure contains a constraint  $0 \leq d$ , and  $d < 0$ . The algorithm takes  $\mathcal{O}(n^2m)$  time, where  $n$  is the number of variables and  $m$  is the number of constraints.

The structural similarity between UTVPI and our graph representation of two-bit patterns demonstrates the potential utility of the techniques introduced above. For instance, we may conceivably define the abstract interpretation in terms of implication graphs and their transformations.

**CONCLUSION AND FUTURE WORK****7.1 Conclusion**

In this dissertation, we introduced the technique of calculating upper bounds on channel capacity of deterministic imperative programs through the use of two-bit patterns. We then introduced the implication graph as a coherent representation of two-bit patterns and laid its mathematical foundation using the language of abstract interpretation. Furthermore, a hybrid optimized approach based on implication graphs, random execution, STP counterexamples, and deductive closure was proposed to allow two-bit pattern analysis to be done more efficiently.

In our case studies of small (but tricky) programs, we found that two-bit patterns usually allow quite accurate bounds to be calculated in a few seconds even using the non-optimized approach. The hybrid optimized approach allows two-bit pattern analysis to be done more efficiently. Experiments show a substantial benefit from the new techniques, with time reductions averaging 72%, and often exceeding 90%.

**7.2 Future Work**

In future work, there are several directions to explore. First, we need to show the completeness of deductive closure. Second, as we scale to complex programs, it is clear that STP queries (and STP counterexamples) about the entire program will ultimately become infeasible. For this reason, we are also interested in exploring the possibility of doing an approximate two-bit pattern analysis as a compositional *abstract interpretation* over the domain of implication graphs. Third, in many real applications, the high input is not the only influence on the observable output.



The *low* input, a non-confidential input, is often an intrinsic part of the system’s functionality. The low input is either provided by users or determined by other external factors. It, too, influences the output. We would like to determine the maximum leakage from the high input to the observable output for all the possible low inputs. Finally, a philosophical study on the origin of undue information flow is crucial for the future of cyber security. We are interested in understanding why there is information flow between the variables with conflicting labels (high/low).

### 7.2.1 Abstract Interpretation

As many senior researchers in the field of formal verification have pointed out, the two-bit patterns already constitute a bit-level relational abstract domain. It is relational because it is about the relationships among the bits. It is an abstract domain because two-bit patterns, represented as implication graphs, form a complete lattice. There is already a partial order, denoted by  $\sqsubseteq^{\text{AM}}$  on the set  $\text{AM}$  of adjacency matrix representations of implication graphs. Intuitively,  $M \sqsubseteq^{\text{AM}} N$  means that for each pair of bits, their constraint in  $M$  is *tighter* than the corresponding constraint in  $N$ .  $\sqsubseteq^{\text{AM}}$  corresponds to the subset inclusion of concrete states:  $M \sqsubseteq^{\text{AM}} N \implies \gamma(M) \subseteq \gamma(N)$ . The set  $\text{AM}$  has the greatest element  $\top^{\text{AM}}$  for  $\sqsubseteq^{\text{AM}}$ . It is defined as  $\forall i, j, \top_{ij}^{\text{AM}} = 0$ .  $\top^{\text{AM}}$  corresponds to the situation where every pair of bits are in *Free* relationship. Therefore,  $\top^{\text{AM}}$  represents the whole space of concrete states. If we extend  $\text{AM}$  by introducing the smallest element, denoted by  $\perp^{\text{AM}}$ , which represents an empty set of the concrete state, we obtain a complete lattice  $(\text{AM}, \sqsubseteq^{\text{AM}}, \sqcup^{\text{AM}}, \sqcap^{\text{AM}}, \top^{\text{AM}}, \perp^{\text{AM}})$  with the following definitions:

$$\begin{aligned} \forall M, N, \quad M \sqsubseteq^{\text{AM}} N &\stackrel{\text{def}}{\iff} \forall i, j, M_{ij} \geq N_{ij} \\ \forall M, N, \quad (M \sqcup^{\text{AM}} N)_{ij} &\stackrel{\text{def}}{=} M_{ij} \wedge N_{ij} \\ \forall M, N, \quad (M \sqcap^{\text{AM}} N)_{ij} &\stackrel{\text{def}}{=} M_{ij} \vee N_{ij} \end{aligned}$$

where  $M$  and  $N$  are adjacency matrices in **AM**. The lattice also has the following definitions with respect to  $\perp^{\text{AM}}$ :

$$\begin{aligned} \forall K^\#, \quad \perp^{\text{AM}} \sqsubseteq^{\text{AM}} K^\# \\ \forall K^\#, \quad \perp^{\text{AM}} \sqcup^{\text{AM}} K^\# &\stackrel{\text{def}}{=} K^\# \sqcup^{\text{AM}} \perp^{\text{AM}} \stackrel{\text{def}}{=} K^\# \\ \forall K^\#, \quad \perp^{\text{AM}} \sqcap^{\text{AM}} K^\# &\stackrel{\text{def}}{=} K^\# \sqcap^{\text{AM}} \perp^{\text{AM}} \stackrel{\text{def}}{=} \perp^{\text{AM}} \end{aligned}$$

where  $K^\#$  stands for any element in **AM** or  $\perp^{\text{AM}}$ .

The challenge is how to define the interpretation on the domain. One solution is to conduct interpretation by using STP solver to find bit patterns for a vector consisting of the concatenation of all the program variables after each instruction in a sequential instruction composition. To illustrate this incremental approach, we use the analysis of the following program to show the transformation of the two-bit patterns after each instruction step in detail:

```
X = X & 0x3;
Y = 16*X + X*X;
```

Assuming that  $X$  is the secret input variable,  $Y$  is the output variable, and both are 6-bit unsigned integers, the program can produce 4 distinct outputs: 0, 17, 36, 57. Therefore, the program has a Shannon/min capacity of 2.

After the first instruction, the analysis finds that the first 4 bits of  $X$  became *Zero*, and all other bits in  $X$  and  $Y$  are non-fixed. It also finds that all of the two-bit

patterns are *Free*. Based on this configuration, the analysis of the second instruction finds more complex patterns among the bits in  $Y$ . Namely, bit 1 is *Zero*. Bit 2 and bit 3 are all less than or equal to bit 5. Moreover, bit 3 is less than or equal to bit 0, which is equal to bit 4. Finally, we have  $Nand(4,2)$ ,  $Nand(0,2)$ , and  $Nand(3,2)$ . There are six solutions corresponding to these bit patterns. Hence, the estimated channel capacity ( under Shannon entropy or min-entropy) is 2.58 bits.

Nevertheless, the two-bit pattern characterization of the feasible intermediate program states can be overly imprecise. Consider the following program:

```
Y = X*X;
```

```
Y = X*X - Y;
```

Assuming that  $X$  is a 2-bit unsigned integer variable and  $Y$  is a 4-bit unsigned integer variable, then the program results with  $Y = 0$ . Hence, there is no leakage in this program. If we analyze bit patterns in concatenation  $XY$  after the first instruction, they are exactly the same as the patterns in the final value of  $Y$  in the previous program. Further analysis of the second instruction would fail to achieve the correct result,  $Y = 0$ .

One way to improve the precision is to extend the current mathematical frame for a more powerful abstract domain. Instead of having only one conjunction of two-bit patterns, we can use multiple disjointed conjunctions of two-bit patterns as the abstract domain. Such extension will allow more complex relationships among bits to be expressed. This, of course, will add considerably to the cost of the analysis.

Due to its bit-wise nature, this kind of abstract interpretation is suitable for static or dynamic analysis for the binary code. One potential application is to detect the security flaws in the binary implementation by analyzing various information flows.

### 7.3 Low Input

The current approach does not distinguish the high input from the low input, either under control of the users or caused by some external factors. In the presence of low inputs, counting the number of feasible outputs can badly overestimate the leakage. As an example, consider the following program:

```
X = X & 0x0000000f;  
L = L & 0xffffffff0;  
Y = X | L;
```

Assuming that  $X$  is the secret input variable,  $L$  is the low input variable,  $Y$  is the output variable, and all are 32-bit unsigned integers, the program can produce  $2^{32}$  distinct outputs. However, only the last 4 bits of  $Y$  are influenced by  $X$ .

Therefore, it is desirable to determine the maximum leakage from the high input to the observable output for all the possible user inputs. One trivial approach is to try all of the values for the low input and estimate the leakage for each of them. This, of course, is computationally inefficient. Heusser and Malacaria’s work [HM10] provides some insights on how to model the leakage with the presence of both high and low inputs. However, more work is still needed to find the “worst” value of the low input which corresponds to the maximum leakage.

### 7.4 The Origin of Undue Information Flow

In addition to the technical improvement on the current bit pattern based code analysis, it is crucial to have an understanding of the origin of undue information flow from historical and philosophical perspectives. Contemporary researchers primarily view the information security issue as a logical problem, and are unaware of

the Cartesian philosophical tradition which we have inherited since the beginning of computer science. Since Descartes, humans have been viewed as self-enclosing rational subjects, who relate to the external world as objects through mental representations. This rationalism has been pervasive in all modern sciences including computer science. The majority of scientists believe that reality can be represented in term of facts and rules.

However, since the 20th century, this Cartesian tradition has been contested by many continental philosophers. The most profound challenge came from the German philosopher, Martin Heidegger. As he pointed out, there are *ontological* differences between the mode of being in the life world and the Cartesian representational mode of being. First, human beings do not usually analyze things theoretically, but rather use them and take them for granted. In everyday life, things are encountered by human beings as mutually referred *equipments* in a unified whole. A house refers to bad weather and to our need to stay dry; the need to stay dry refers to our medical knowledge; this knowledge refers in turn to our fear of illness. We are woven together with this referential totality, the world. Second, the concepts of time are different in these two modes of being. Unlike the chronological time, an infinite series of “nows” counted by a clock, the time in the human everyday life is continuous. According to Heidegger, temporality is a unity against which past, present and future stand out, while remaining essentially interlocked.

When a programmer creates a software, he/she always projects a particular social reality in the *life world* onto a discrete and procedural *universe* of bits. In this virtual universe, things are uniformized as binaries. The organic relationships among them are also mechanized in Turing Machines. This immediately put things in a “vulnerable” state. For any software system which strives to imitate this organic human social existence, it is difficult to keep all of the elements in the system

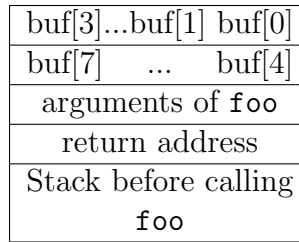


Figure 7.1: The stack holds the return address, the arguments, and the local variables for `foo`

(variables or other resources) within the same statically defined categories (types). At some points of the computation, there will be some variables or handlers which are simultaneously members of different categories. While many of the overlaps are harmless, some can cause serious security problems, especially when the overlap occurs at distinct abstraction layers in the system. As an example, consider the following function `foo`:

```
void foo(const char* input) {
    char buf[8];
    strcpy(buf, input);
}
```

When this function is called from another function, a series of actions occur at the system level. First, the calling function pushes the return address, that is the address of the return statement onto the stack. Then, the called function pushes zeroes on the stack to store its local variable. Since `foo` has an character array `buf[8]`, there will be space for 8 characters allocated. The configuration of the stack is depicted in Figure 7.1. The call to `strcpy` is dangerous here. The function `strcpy` simply copies characters until it encounters a “0” character in the source string. Since the argument which is given to the call can be much longer, it can overwrite the return address in the stack. This vulnerability can be maliciously

exploits
exploits
exploits
starting address of the exploits
Stack before calling foo

Figure 7.2: The stack with overwritten return address

exploited by an adversary to gain the control over the system. The stack with an an overwritten return address is depicted in Figure 7.2. When the buffer overflow happens, the original memory location for the return address becomes a member for two conflicting categories now. It is simultaneously a system parameter and a user input!

In the future, information flow research needs to move beyond the pure logical view of computing and be more aware of this ontological difference. The operational semantics, both in information theory and the enforcement technique, should reflect the the dynamics of human existence and its historical character.

## BIBLIOGRAPHY

- [AAP10] Mário Alvim, Miguel Andrés, and Catuscia Palamidessi. Probabilistic information flow. In *Proc. 25th IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 314–321, 2010.
- [ACPS12] Mário S. Alvim, Kostas Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring information leakage using generalized gain functions. In *Proc. 25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 265–279, June 2012.
- [APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [APvRS10] Miguel Andrés, Catuscia Palamidessi, Peter van Rossum, and Geoffrey Smith. Computing the leakage of information-hiding systems. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 373–389, 2010.
- [BCP09] Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative notions of leakage for one-try attacks. In *Proc. 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009)*, volume 249 of *ENTCS*, pages 75–91, 2009.
- [BKR09] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proc. 30th IEEE Symposium on Security and Privacy*, pages 141–153, 2009.
- [CC04] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In *Building the Information Society*, pages 359–366. Kluwer Academic Publishers, 2004.
- [CCG10] Konstantinos Chatzikokolakis, Tom Chothia, and Apratim Guha. Statistical measurement of information leakage. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 390–404, 2010.



- [CHM05] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 18(2):181–199, 2005.
- [CMS05] Michael Clarkson, Andrew Myers, and Fred Schneider. Belief in information flow. In *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW '05)*, pages 31–45, 2005.
- [CPP08] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206:378–401, 2008.
- [CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., second edition, 2006.
- [Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley & Sons, Inc., third edition, 1968.
- [Gal68] Robert G. Gallager. *Information Theory and Reliable Communication*. John Wiley & Sons, Inc., 1968.
- [GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 524–536, 2007.
- [GK96] Andrew V. Goldberg and Alexander V. Karzanov. Path problems in skew-symmetric graphs. *Combinatorica*, 16(3):353–382, 1996.
- [HJB11] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient cnf simplification based on binary implication graphs. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing, SAT'11*, pages 201–215, Berlin, Heidelberg, 2011. Springer-Verlag.
- [HM10] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proc. ACSAC '10*, pages 261–269, 2010.
- [HSP10] Sardaouna Hamadou, Vladimiro Sassone, and Catuscia Palamidessi. Reconciling belief and vulnerability in information flow. In *Proc. 31st IEEE Symposium on Security and Privacy*, pages 79–92, 2010.

- [JMSY94] Joxan Jaffar, Michael J Maher, Peter J Stuckey, and Roland HC Yap. Beyond finite domains. In *Principles and Practice of Constraint Programming*, pages 86–94. Springer, 1994.
- [JPF] <http://babelfish.arc.nasa.gov/trac/jpf/>.
- [KB07] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 286–296, 2007.
- [KMO12] Boris Köpf, Laurant Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proc. 24th International Conference on Computer-Aided Verification (CAV '12)*, pages 564–580, 2012.
- [KR10] Boris Köpf and Andrey Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 3–14, 2010.
- [Kro67] Melven R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 13:15–20, 1967.
- [KS10] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 44–56, 2010.
- [Mac03] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.
- [Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *Proc. 34th Symposium on Principles of Programming Languages (POPL '07)*, pages 225–235, 2007.
- [Mas94] James L. Massey. Guessing and entropy. In *Proc. 1994 IEEE International Symposium on Information Theory*, page 204, 1994.
- [Min01] Antoine Min. The octagon abstract domain. In *Eighth Working Conference on Reverse Engineering*, pages 310–319. IEEE Computer Society, 2001.

- [MS11] Ziyuan Meng and Geoffrey Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security, PLAS '11*, pages 1:1–1:12, New York, NY, USA, 2011. ACM.
- [MS13] Ziyuan Meng and Geoffrey Smith. Faster two-bit pattern analysis of leakage. In *Proc. 2nd International Workshop on Quantitative Aspects in Security Assurance (QASA '13)*, pages 2:1–2:14, 2013.
- [NMS09] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proc. Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pages 73–85, 2009.
- [PMTP12] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.
- [R61] Alfréd Rényi. On measures of entropy and information. In *Proc. 4th Berkeley Symposium on Mathematics, Statistics and Probability 1960*, pages 547–561, 1961.
- [Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.
- [Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In Luca de Alfaro, editor, *Proc. 12th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS '09)*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302, 2009.
- [Val79] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. on Computing*, 8:410–421, 1979.
- [YT10] Hirotohi Yasuoka and Tachio Terauchi. Quantitative information flow — verification hardness and possibilities. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 15–27, 2010.

## VITA

### ZIYUAN MENG

October 22, 1979	Born, Shijiazhuang, Hebei, China
2002	B.S., Computer Science Tongji University Shanghai, China
2005	M.S., Computer Science Florida International University (Beijing Campus) Beijing, China
2005–2006	VLSI verification engineer VIA Technologies Inc Beijing, China
2007–2009	Teaching assistant Florida International University Miami, Florida
2009–present	Research assistant Florida International University Miami, Florida

### PUBLICATIONS

Ziyuan Meng and Geoffrey Smith, “Calculating Bounds on Information Leakage Using Two-Bit Patterns,” Proceeding of ACM SIGPLAN Sixth Workshop on Programming Languages and Analysis for Security (PLAS), pages 1:1-1:12, 2011.

Ziyuan Meng and Geoffrey Smith, “Faster Two-Bit Pattern Analysis of Leakage,” Proceeding of 2nd International Workshop on Quantitative Aspects of Security Assurance (QASA), pages 2:1-2:14, 2013.