Calculating Bounds on Information Leakage Using Two-Bit Patterns

Ziyuan Meng   Geoffrey Smith
Florida International University
zmeng001@cis.fiu.edu, smithg@cis.fiu.edu

Abstract

Theories of quantitative information flow have seen growing interest recently, in view of the fundamental importance of controlling the leakage of confidential information, together with the pragmatic necessity of tolerating intuitively "small" leaks. Given such a theory, it is crucial to develop automated techniques for calculating the leakage in a system. In this paper, we address this question in the context of deterministic imperative programs and under the recently-proposed min-entropy measure of information leakage, which measures leakage in terms of the confidential information's vulnerability to being guessed in one try by an adversary. In this context, calculating the maximum leakage of a program reduces to counting the number of feasible outputs that it can produce. We approach this task by determining patterns among pairs of bits in the output, for instance by determining 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 feasible outputs and hence on the leakage. We explore the effectiveness of our approach on a number of case studies, in terms of both efficiency and accuracy.

Categories and Subject Descriptors E.4 [*Coding and Information Theory*]; D.2.4 [*Software/Program Verification*]

General Terms Security, Measurement, Languages

Keywords quantitative information flow, min-entropy

1. Introduction

One of the most fundamental challenges for trustworthy computing is to control the *flow of information*, whether to prevent confidential information from being *leaked*, or to prevent trusted information from being *tainted*. But while it is sometimes possible to stop undesirable information flows completely, it is perhaps more typical that some undesirable flows are unavoidable. For instance an ATM machine that rejects an incorrect PIN thereby reveals that the secret PIN differs from the one that was entered. Similarly, revealing the tally of votes in an election reveals some information about the secret ballots that were cast. More subtlely, 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 [CHM05, CMS05, KB07, Smi09, AAP10, HSP10]. Such theories allow us to talk about "how much" information is leaked and (perhaps) allow us to tolerate "small" leaks.

Given such a theory, it is of course 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.

Our main contribution in this paper is to introduce and explore the use of what we call *two-bit patterns* to calculate *upper bounds* on the maximum amount of *min-entropy leakage* in deterministic imperative programs. Min-entropy leakage is an alternative to the more commonly-used measures based on Shannon entropy and mutual information; in Section 2 we review its theory and motivation in a security context. For now, it suffices to know that the maximum min-entropy leakage of a deterministic program is simply the logarithm (base 2) of the *number of feasible outputs* that it can produce. Our approach to bounding this quantity is to determine *two-bit patterns* among the bits of the feasible outputs.

Consider C-like programs that take as input a secret value S and produce an output O, 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,[1] which of the four combinations $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$ are feasible values for bits i and j of O. As an example, consider the following program, adapted from [NMS09]:

[1] We number the bits from 0 to 31, right to left.

PLAS'11 June 5, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0830-4/11/06...$10.00

```
O = ((S >> 16) ^ S) & 0xffff;
O = O | O << 16;
```

Here there are $32 \cdot 31/2 = 496$ two-bit patterns on O to determine; the only interesting ones 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, we get an *upper bound* on the number of feasible values of O. Here there are 2^{16} solutions to the two-bit patterns, giving a maximum min-entropy leakage of at most $\log 2^{16} = 16$ bits, which is exact in this case. In general, two-bit patterns may greatly overestimate the leakage, but we will present experimental evidence that they tend to give reasonably accurate bounds.

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

```
if (S % 2 == 0)
  O = 0;
else
  O = 0xffffffff;
```

in which all 32 bits of O are non-fixed, even though it has only two feasible values.

In this paper, we restrict ourselves to *loop-free* programs. Programs with loops are commonly handled by unrolling them up to some limit [BKR09, NMS09, HM10]. (An example of this can be seen in our completely unrolled binary search program in Section 4.6.) But of course unrolling loops only partially can result in *underestimating* the leakage, so this remains an important area for future research.

The rest of the paper is structured as follows. In Section 2, we begin by reviewing the theory of the min-entropy measure of leakage that we use in this work. In Section 3 we explain our technique in detail on an illustrative example. Then in Section 4 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 Section 5 we discuss related work and compare it with our approach. Finally, in Section 6 we discuss future directions and conclude.

2. Theory of Min-Entropy Leakage

In this section, we briefly review the mathematical foundations of quantitative information flow analysis, recalling important concepts of information theory [Sha48, Fel68, Gal68, Mac03, CT06], such as Shannon entropy and mutual information, along with the *min-entropy* measure of information leakage proposed in [Smi09]. We also present key results about min-entropy leakage that have appeared in the recent literature.

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*, which gives the conditional probability of each possible output, given each possible input. (Here "outputs" should be understood to encompass any aspects of the system's behavior that are observable to an adversary.) Within this framework, it is natural to quantify *leakage* of confidential information based on the extent to which the channel's output helps an adversary to determine the input. (Quantitative *integrity* has a less clear basis, but there is important recent work [NMS09, CS10] that considers the amount of *influence* that untrusted inputs have on trusted outputs.)

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

Any *a priori* distribution P_S on $\mathcal{S}$ determines a random variable S. Moreover, P_S and $C_{\mathcal{SO}}$ determine a joint probability P_{SO} on $\mathcal{S} \times \mathcal{O}$, where $P_{SO}[s,o] = P_S[s]C_{\mathcal{SO}}[s,o]$, and a marginal distribution P_O on $\mathcal{O}$, where $P_O[o] = \sum_{s \in \mathcal{S}} P_{SO}[s,o]$, giving a random variable O.

We *quantify* the amount of information that flows from S to O by considering an adversary $\mathcal{A}$ who wishes to guess the value of S. It is natural to measure information leakage by comparing $\mathcal{A}$'s "uncertainty" about S before and after seeing the value of O, using the equation

leakage = initial uncertainty – remaining uncertainty.

2.1 Measuring leakage using mutual information

Most of the literature on quantitative information flow (for example, [CPP08] and [Mal07]) has defined "uncertainty" using *Shannon entropy* and *conditional Shannon entropy* [Sha48]:

$$H(S) = -\sum_{s \in \mathcal{S}} P_S[s] \log P_S[s]$$

and

$$H(S|O) = \sum_{o \in \mathcal{O}} P_O[o] H(S|o),$$

which leads to defining leakage as *mutual information*:

$$\text{leakage} = H(S) - H(S|O) = I(S;O).$$

In the special case of a deterministic channel, note that we have $H(O|S) = 0$, since the value of O is determined by the value of S. Using the fact that mutual information is symmetric, this gives a simpler formula for leakage in a deterministic channel:

$$I(S;O) = I(O;S) = H(O) - H(O|S) = H(O).$$

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 S, given O. For $H(S|O)$, Massey's *guessing entropy* bound [Mas94] shows that $G(S|O)$, the expected number of guesses required to guess S given O, grows exponentially with $H(S|O)$. A weakness of this bound, however, is that $G(S|O)$ can be arbitrarily high even when S is highly vulnerable to being guessed in *one* try. A key example from [Smi09] illustrates this. Consider the program

```
if ($S$ % 8 == 0)
    $O = S$;
else
    $O = 1$;
```
(1)

where S is a uniformly-distributed 64-bit unsigned integer, $0 \leq S < 2^{64}$, so that the initial uncertainty $H(S) = 64$. For this program, the mutual information leakage is

$$I(S;O) = H(O) = 2^{61}2^{-64}\log 2^{64} + \frac{7}{8}\log\frac{8}{7} \approx 8.17,$$

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

$$G(S|O) = \frac{1}{8}\cdot 1 + \frac{7}{8}\cdot\frac{1}{2}\cdot(\frac{7}{8}2^{64}+1) \approx 2^{62.6}.$$

It is instructive to compare program (1) with

$$O = S \,\&\, 0777; \tag{2}$$

which simply copies the last 9 bits of S into O. The mutual information leakage of program (2) is 9, making it *worse* than program (1), even though it gives $\mathcal{A}$ a probability of guessing S in one try of only 2^{-55}, since the first 55 bits of S remain completely unknown.

2.2 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 S to being guessed correctly *in one try* by $\mathcal{A}$. If we make the worst-case assumption that $\mathcal{A}$ knows P_S and C_{SO}, then the *a priori* vulnerability is

$$V(S) = \max_{s\in\mathcal{S}} P_S[s]$$

and the *a posteriori* vulnerability is

$$\begin{aligned} V(S|O) &= \sum_{o\in\mathcal{O}} P_O[o]\max_{s\in\mathcal{S}} P[s|o] \\ &= \sum_{o\in\mathcal{O}} \max_{s\in\mathcal{S}} P_{SO}[s,o] \\ &= \sum_{o\in\mathcal{O}} \max_{s\in\mathcal{S}} (P_S[s]C_{SO}[s,o]). \end{aligned}$$

(Notice that this is the complement of *Bayes risk*.) We convert from vulnerability to uncertainty by taking the negative logarithm, giving Rényi's *min-entropy* [Rén61]. Our definitions, then, are

- initial uncertainty: $H_\infty(S) = -\log V(S)$
- remaining uncertainty: $H_\infty(S|O) = -\log V(S|O)$

(It should be noted however that there is no universally agreed-upon definition of conditional min-entropy $H_\infty(S|O)$ in the literature [Cac97].) Finally, we define the *min-entropy leakage from S to O*, denoted $\mathcal{L}_{SO}$, to be

$$\begin{aligned} \mathcal{L}_{SO} &= H_\infty(S) - H_\infty(S|O) \\ &= -\log V(S) - (-\log V(S|O)) \\ &= \log\frac{V(S|O)}{V(S)}. \end{aligned}$$

Thus min-entropy leakage is the logarithm of the factor by which knowledge of O increases the expected one-guess vulnerability of S.

Revisiting program (1), we find that its min-entropy leakage is 61.00, reflecting the fact that $V(S|O) \approx 1/8$. In contrast, for program (2) the min-entropy leakage is 9, reflecting the fact that $V(S|O) = 2^{-55}$.

2.3 Min-capacity

An important notion in information theory is *channel capacity*, which is the maximum leakage over all possible *a priori* distributions. When we measure leakage using mutual information, we will use the name *Shannon capacity*, and when we measure leakage using min-entropy, we will use the name *min-capacity* and the notation $\mathcal{ML}(C_{SO})$. While calculating the Shannon capacity of a channel matrix is in general difficult, calculating the min-capacity is easy, as it is just the logarithm of the sum of the column maximums of C_{SO} [BCP09, KS10]:

$$\mathcal{ML}(C_{SO}) = \log\sum_{o\in\mathcal{O}}\max_{s\in\mathcal{S}} C_{SO}[s,o]$$

Moreover, min-capacity is always realized by a uniform distribution on $\mathcal{S}$ (and possibly by other distributions as well).

As a corollary, the min-capacity of a *deterministic* channel is just the logarithm of the number of feasible outputs. Interestingly, this is also the Shannon capacity [Smi09]:

THEOREM 2.1. *If C_{SO} is deterministic, then its min-capacity and Shannon capacity coincide, with both equal to $\log|\mathcal{O}|$ (assuming that every element of $\mathcal{O}$ is feasible).*

However, this coincidence does not carry over to the general case of probabilistic channels.

3. Using Two-Bit Patterns to Calculate Leakage

Our approach to bounding the min-capacity of a deterministic C-like program can be broken down 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 and two-bit patterns among the 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 min-capacity.

Throughout this section, we use the analysis of the program shown in Figure 1 to illustrate these three steps in detail. In all our examples, `S` is the secret input variable, `O` is the output variable, and both are 32-bit unsigned integers.

3.1 Step 1: Derive predicates from the program

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 1 gives rise to the predicate

```
S1 = S0 & 0x77777777
```

Here the symbols `S0` and `S1` represent variable `S`'s value before and after the first assignment. Since variable `O` is not affected, no new symbol is introduced for it and `O0` (variable `O`'s initial value) remains current.

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

```
O1 = if S1 <= 64 then S1 else 0

O2 = if O1 mod 2 = 0 then O1+1 else O1
```

These three predicates implicitly constitute a symbolic description of `O`'s final value in terms of `S`'s initial value.

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, and the operations in each predicate are replaced with STP equivalents. For instance, the expression `O1 mod 2` translates into

```
BVMOD(32, O1, 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 1 is shown in Figure 2.

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

3.2 Step 2: Discover one-bit and two-bit patterns

Now we wish to discover the relations (bit patterns) among the bits in `O2`, which is the final value of the output variable `O`. We achieve this by making STP *queries* with respect to the assertions generated in Step 1. In STP, `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 `O2`. A one-bit pattern describes the set of feasible 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(O2[i:i] = 0bin0)
```

tests whether bit `i` of `O2` 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(O2[i:i] = 0bin1)
```

tests whether bit `i` of `O2` 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 O_f, then the algorithm to determine the one-bit pattern for bit i of O_f is

```
**if** $(O_f[i] = 0)$ is valid **then**
    *Zero*
**else if** $(O_f[i] = 1)$ 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 1, as translated into Figure 2, STP discovers that 26 of the bits in `O2` 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*:

```
0000000000000000000000000***0**1
```

```
S = S & 0x77777777;
if (S <= 64) O = S; else O = 0;
if (O % 2 == 0) O++;
```

Figure 1. Illustrative example program that leaks information from `S` to `O`

```
S0, S1 : BITVECTOR(32);
O1, O2 : BITVECTOR(32);

ASSERT(S1 = S0 & 0hex77777777);

ASSERT(O1 = IF (BVLE(S1, 0hex00000040))
            THEN S1
            ELSE 0hex00000000
            ENDIF);

ASSERT(O2 = IF (BVMOD(32, O1, 0hex00000002) = 0hex00000000)
            THEN BVPLUS(32, O1, 0hex00000001)
            ELSE O1
            ENDIF);
```

Figure 2. Translation of illustrative example into STP

Discovering these 32 one-bit patterns required a total of 38 STP queries and 1562 ms.[2] Notice that we can immediately conclude that the number of feasible values for `O2` 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 pattern *Non-fixed*. A two-bit pattern describes the set of feasible 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 complete enumeration of the possible two-bit patterns:

1. $\{00\}$
2. $\{01\}$
3. $\{10\}$
4. $\{11\}$
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. So 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 pattern testing procedure is similar to one-bit pattern testing. It determines the two-bit patterns via a decision tree of queries. For instance,

```
QUERY(O2[i:i] = 0bin0 OR O2[j:j] = 0bin0)
```

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

```
QUERY(O2[i:i] = 0bin1 OR O2[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

[2] Throughout this paper, all times are given in milliseconds.

be feasible, 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. 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 O_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 1, it turns out that there are four interesting two-bit patterns among the 5 non-fixed bits of `O2`, 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 studies in Section 4. 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 (O % 2 == 0) O++;
```

to the equivalent

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

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

3.3 Step 3: Count the number of solutions

Finally, we determine an upper bound on the number of feasible outputs by counting the number of solutions to the two-bit patterns. We do this using the `SatisfiabilityCount` function provided by *Mathematica*.[3] Given a boolean proposition P and a list of boolean variables $b_1, b_2, \ldots$, the Mathematica call

$$\texttt{SatisfiabilityCount}[P, \{b_1, b_2, \ldots\}]$$

returns the number of truth assignments to b_1, b_2, ... 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 1, we make the call

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

[3] `http://www.wolfram.com/mathematica/`

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 practicality 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.

To conclude the discussion of our illustrative example, note that our two-bit pattern approach takes just over 4 seconds to find an upper bound (17) on the number of feasible outputs, and here it turns out to be exact. One might wonder how these results compare with a more straightforward approach to counting the number of feasible outputs. Specifically, we can test whether any 32-bit value `v` is a feasible output using the STP query

```
  QUERY(NOT(O2[0:31] = v))
```

which returns `INVALID` iff `v` is a feasible 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 feasible. However, experiments show that on average each of these queries takes 28 ms, which implies that it would take 3.8 years to complete the 2^{32} queries.

4. Case Studies

In this section, we assess the accuracy and efficiency of our two-bit pattern approach by trying it on a number of case studies, many of which come from the recent literature in quantitative information flow analysis.

4.1 Sanity Check

Consider the "sanity check" program from [NMS09], where `O` is influenced by `S` only when `S` is found to be within an acceptable range:

```
  if (S < 16)
    O = base + S;
  else
    O = base;
```

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

Figure 3. Algorithm to determine two-bit patterns for O_f

The feasible outputs here range from `base` to `base+15`, giving a min-capacity of $\log 16 = 4$ bits.

An interesting property of this program is that the bit patterns of `O`'s final value depend on the initial value of `base`. For instance, when `base` is `0x00001000`, then 28 bits in `O` are fixed: bits 31 through 13 and bits 11 through 4 are *Zero*, and bit 12 is *One*. The last 4 bits are *Non-fixed*, so the one-bit patterns are

```
0000000000000000000100000000****
```

The two-bit patterns among the 4 non-fixed bits are all *Free*. `SatisfiabilityCount` computes that there are 16 instances which satisfy all these bit patterns, giving a (precisely correct) min-capacity of 4 bits. In terms of efficiency, the one-bit patterns required 37 STP queries and 57 ms, the two-bit patterns required 24 STP queries and 35 ms, and `SatisfiabilityCount` required less than 1 ms.

In contrast, when `base` is `0x7ffffffa`, the bit patterns in `O` are more complex, since now the feasible outputs range from `0x7ffffffa` to `0x80000009`. Here 64 STP queries (taking 88 ms) show that all 32 bits are *Non-fixed*. But there are many 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 required 1552 STP queries and 3095 ms. Finally, `SatisfiabilityCount` required 1 ms to determine that there are 24 solutions to the bit patterns, implying a min-capacity of at most $\log 24 \approx 4.58$ bits, which is close to the actual capacity of 4 bits.

4.2 Implicit Flow

Here is a program from [NMS09] that indirectly copies `S` to `O` if $S \leq 6$; otherwise it sets `O` to 0:

```
O = 0;
if (S == 0) then O = 0;
else if (S == 1) then O = 1;
else if (S == 2) then O = 2;
else if (S == 3) then O = 3;
else if (S == 4) then O = 4;
else if (S == 5) then O = 5;
else if (S == 6) then O = 6;
else O = 0;
```

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

Our analysis finds that the one-bit patterns here are

```
00000000000000000000000000000***
```

and the two-bit patterns on the 3 *Non-fixed* bits are all *Free*. Hence there are 8 solutions to the bit patterns, implying a min-capacity of at most 3 bits.

Notice that two-bit patterns do not capture that fact that 7 is not a feasible output here. The reason is that while the last three bits of `O` cannot all be 1, any *two* of them can be 1.

4.3 Population Count

This program from [NMS09] uses clever bit operations to count the number of bits in S that are 1, and leaks this count to O:

```
S = (S & 0x55555555) + ((S>>1) & 0x55555555);
S = (S & 0x33333333) + ((S>>2) & 0x33333333);
S = (S & 0x0f0f0f0f) + ((S>>4) & 0x0f0f0f0f);
S = (S & 0x00ff00ff) + ((S>>8) & 0x00ff00ff);
O = (S + (S>>16)) & 0xffff;
```

It has 33 feasible outputs, so its min-capacity is $\log 33 \approx 5.044$ bits.

Here our analysis (using 38 STP queries and 292 ms) finds that the one-bit patterns are

```
  00000000000000000000000000******
```

Among the 6 non-fixed bits we find (using 50 STP queries and 1138 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, so our bound is exact.

4.4 Mix and Duplicate

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

```
  O = ((S >> 16) ^ S) & 0xffff;
  O = O | O << 16;
```

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

In 64 STP queries and 33 ms, our analysis finds that all 32 bits are non-fixed. Then, in 1968 STP queries and 1267 ms, we find that there are 16 *Eq* patterns

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

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 min-capacity of at most $\log 65536 = 16$ bits, which is again exact.

4.5 Masked Copy

This simple program copies the first 16 bits of S into O, masking out the last 16 bits:

```
  O = S & 0xffff0000;
```

Like the previous example, it has $2^{16} = 65536$ feasible outputs, giving a min-capacity of 16 bits.

In 48 STP queries and 14 ms, we find that the one-bit patterns are

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

In 480 STP queries and 183 ms, we find that all two-bit patterns are *Free*. So we get a min-capacity of 16 bits, which is again exact.

4.6 Binary Search

Now we consider a program that uses binary search to leak the first b bits of S to O:

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

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

```
  O = 0;
  if (O + 2147483648 <= S) O += 2147483648;
  if (O + 1073741824 <= S) O += 1073741824;
  if (O + 536870912 <= S) O += 536870912;
  if (O + 268435456 <= S) O += 268435456;
  if (O + 134217728 <= S) O += 134217728;
  if (O + 67108864 <= S) O += 67108864;
  if (O + 33554432 <= S) O += 33554432;
  if (O + 16777216 <= S) O += 16777216;
  if (O + 8388608 <= S) O += 8388608;
  if (O + 4194304 <= S) O += 4194304;
  if (O + 2097152 <= S) O += 2097152;
  if (O + 1048576 <= S) O += 1048576;
  if (O + 524288 <= S) O += 524288;
  if (O + 262144 <= S) O += 262144;
  if (O + 131072 <= S) O += 131072;
  if (O + 65536 <= S) O += 65536;
```

Like the previous example, it has $2^{16} = 65536$ feasible outputs, giving a min-capacity of 16 bits.

In 48 STP queries and 358 ms, we find that the one-bit patterns are

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

In 480 STP queries and 6207 ms, we find that all two-bit patterns are *Free*. So we get a min-capacity of 16 bits, which is again exact.

We have also conducted experiments to see how the time required to calculate the bit patterns for the binary search program grows with b. Notice that, because the program makes b independent branches, the number of program paths is exactly 2^b. This might make us fear that the analysis time will grow as explosively as 2^b. Fortunately, we have observed far more moderate growth as b is successively doubled:

b	1	2	4	8	16	32
time	20	39	158	879	6426	55524
growth factor	—	2.0	4.1	5.6	7.3	8.6

The steadily increasing growth factor does suggest that the running time is exponential in `b`, however.

4.7 Electronic Purse

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

```
O = 0;
while(S >= 5) {
  S = S - 5;
  O = O + 1;
}
```

Here we add the assumption that `S` < 20, which means that `O` can range from 0 to 3, giving a min-capacity of 2 bits.

Unrolling the loop, we find (in a total of 417 ms) that the first 30 bits of `O` must be 0, and the last 2 bits are *Free*, giving a min-capacity of 2 bits.

4.8 Sum Query

Here is the sum query from [BKR09]:

```
O = S1;
O = O + S2;
O = O + S3;
```

Here we assume that `S1`, `S2`, and `S3` are each less than 10. This means that there are 28 feasible outputs (from 0 to 27) and a min-capacity of $\log 28 \approx 4.807$ bits.

In a total of 366 ms, we find that the first 27 bits of `O` must be 0, and find that the last 5 bits are *Free*, giving a min-capacity of $\log 32 = 5$ bits.

4.9 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 feasible outputs:

```
if (S == r1) O = r1;
else if (S == r2) O = r2;
else if (S == r3) O = r3;
...
else if (S = r9) O = r9;
else O = 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 `O` will all be *Non-fixed*, and the two-bit patterns will overwhelmingly be *Free*, leading us to greatly overestimate the min-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 min-capacity of 18.645 bits, far exceeding the actual min-capacity of $\log 10 \approx 3.322$ bits. While the inaccuracy here is striking, the significance seems unclear, as it is not clear whether this sort of output behavior is common in practice.

4.10 Summary of Case Studies

We summarize our results on the case studies in three tables. Table 1 compares the actual min-capacities with our upper bounds. Table 2 shows our times (in milliseconds) to compute one-bit patterns, two-bit patterns, and to count the number of solutions to the bit patterns. Finally, Table 3 presents details of our bit-pattern analyses: the number of STP queries to determine the one-bit and two-bit patterns, the number of *Non-fixed* bits, and the number of *Free* pairs.

5. Related Work

Calculating quantitative information flow is a challenging problem, as shown for example by the negative computational complexity results given in [YT10]. That 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 deterministic imperative programs [BKR09, KR10, NMS09, HM10] and probabilistic systems [APvRS10, CCG10]. Our focus here will be on techniques for calculating min-entropy leakage and min-capacity of deterministic imperative programs.

Perhaps most similar to our work 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 2.1 above this coincides with min-capacity, and amounts simply to counting the number of feasible output values. They estimate this through various heuristics, using STP to check whether a particular output is feasible or not, and whether an *interval* contains any feasible outputs. Using binary search, they try to find which intervals in the range of `O` contain feasible outputs and which do not. When they find that an interval contains at least one feasible output, they sometimes use random sampling to estimate the *density* of feasible 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} feasible 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 feasible output values. But this is expensive, taking (they say) 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 reasonably quickly and they usually provide quite accurate upper

Program	Min-capacity	Upper bound
Illustrative example	4.087	4.087
Sanity check, `base=0x00001000`	4.	4.
Sanity check, `base=0x7ffffffa`	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 1. Accuracy of our upper bounds

Program	One-bit patterns	Two-bit patterns	`SatisfiabilityCount`
Illustrative example	1562	2558	<1
Sanity check, `base=0x00001000`	57	35	<1
Sanity check, `base=0x7ffffffa`	91	3095	1
Implicit flow	18	27	<1
Population count	292	1138	<1
Mix and duplicate	33	1267	42
Masked copy	14	183	<1
Binary search, `b=16`	358	6207	<1
Electronic purse	324	93	<1
Sum query	215	151	<1
Ten random outputs (average)	132	4603	14

Table 2. Times in ms to calculate our bounds

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

Table 3. Details of our bit pattern analyses

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 S to output O, under a uniform *a priori* distribution. While they present the technique in terms of estimating $H(S|O)$, it is clearer to remember that the mutual-information leakage is just $H(O)$. They assume that for each feasible output value o, we can estimate its probability (by estimating the number of values of S that lead to o). Then they observe that $H(O)$ is the expected value of $log\frac{1}{P(o)}$, where o is a sampled output value:

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

With n samples, $o_1, o_2, \ldots o_n$, we find that $H(O)$ is also the expected value of $\frac{1}{n}\sum_{i=1}^{n}\log\frac{1}{P(o_i)}$. 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 *Shannon capacity* of deterministic programs, again this amounts to counting the number of feasible output values. Rather than trying to *calculate* the capacity, they instead try to solve the simpler 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' that runs P independently b times on nondeterministically-chosen inputs, and then checking (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. While the technique yields interesting results on leakage in real Linux kernel vulnerabilities, it is important to note that 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.

6. Conclusion and Future Work

In this paper, we have introduced the technique of calculating upper bounds on min-capacity of deterministic imperative programs through the use of one-bit and two-bit patterns. On 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.

In future work, there are several directions to explore in trying to calculate the bit patterns more efficiently. First, there seem to be good opportunities for optimizations based on *transitivity*. Recall the Sanity Check program from Section 4.1 when `base` is `0x7ffffffa`. In this case, it turned out that 27 bits (bits 30 through 4) are equal to one another, giving $27 \cdot 26/2 = 351$ *Eq* pairs. Rather than using $3 \cdot 351 = 1053$ STP queries to find each these *Eq* pairs using the algorithm in Figure 3, it would be faster to exploit the transitivity of the *Eq* relation. We could do this by maintaining the complete *Eq* relation as a 32×32 boolean matrix, and computing its *transitive closure* whenever a new *Eq* pair is found. A second approach to optimizing the calculation of the bit patterns is to first find a number of feasible outputs, either by running the program on randomly-chosen inputs, or by taking advantage of STP's *counterexample* feature. Such feasible outputs give us partial information about the bit patterns, potentially allowing us to find the complete bit patterns using a greatly reduced number of subsequent STP queries. It remains to be seen how generally beneficial these sorts of optimizations will be.

To deal with programs (like case study 4.9) where bit patterns are insufficient, we could consider a hybrid approach that augments the bit patterns with an additional predicate P (for example giving range information about `O`) in the `SatisfiabilityCount` query.

To scale up to the analysis of realistic programs, it will also be important to explore *compositional analyses*, in contrast to the whole-program analysis we use here. One possibility in analyzing a sequential composition $c_1; c_2$ would be to abstractly represent the program state at the end of c_1 using bit patterns among *all* the program variables, rather than just on `O`. For greater accuracy, it might here be useful to consider *three-bit* patterns, but it is unclear whether these would be worth the extra cost that they would entail. Also, the translation of programs into STP assertions needs to be automated and generalized, especially to deal with loops, and perhaps to deal with binaries rather than C-like code.

Finally, on programs with a *low* input `I` supplied by the adversary, we would like to find efficient ways to calculate bounds on the maximum leakage over *all* values of `I`.

Acknowledgments

We are grateful to the anonymous referees for their helpful comments. This work was partially supported by the National Science Foundation under grant CNS-0831114.

References

[AAP10] Mario 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.

[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.

[Cac97] Christian Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, Swiss Federal Institute of Technology, 1997.

[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.

[CS10] Michael R. Clarkson and Fred B. Schneider. Quantification of integrity. In *Proc. 23nd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 28–43, 2010.

[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 and 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.

[HM10] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proc. ACSAC '10*, 2010.

[HSP10] Sardaouna Hamadou, Vladimiro Sassone, and Catuscia Palamidessi. Reconciling belief and vulnerability in information flow. In *Proc. 31th IEEE Symposium on Security and Privacy*, pages 79–92, 2010.

[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.

[KR10] Boris Köpf and Andrey Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *Proc. 23nd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 3–14, 2010.

[KS10] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *Proc. 23nd 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.

[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.

[Rén61] 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. Valient. The complexity of enumeration and reliability problems. *SIAM J. on Computing*, 8:410–421, 1979.

[YT10] Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow — verification hardness and possibilities. In *Proc. 23nd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 15–27, 2010.