Faster Two-Bit Pattern Analysis of Leakage

Ziyuan Meng and Geoffrey Smith

School of Computing and Information Sciences
Florida International University, Miami, FL 33199, USA
zmeng001@cis.fiu.edu, smithg@cis.fiu.edu

Abstract. In the context of quantitative information flow analysis, *two-bit patterns* are a recent approach to computing upper bounds on leakage in deterministic programs. This paper shows that two-bit pattern analysis can be done more efficiently through the use of four new techniques: *implication graphs*, *random execution*, *STP counterexamples*, and *deductive closure*. We find that these techniques reduce the analysis time for a set of case studies by an average of 72%; in close to half the cases, the reduction is greater than 90%.

1 Introduction

Much experience has shown that computer systems are prone to leaks, both inadvertent and malicious, of their confidential data. Moreover, eliminating such leaks completely is often infeasible, for instance due to the existence of side channels based on timing or power consumption. *Quantitative information flow* addresses 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; this area has seen growing interest over the past decade (e.g. [1–6].)

To give some quick intuition, assume (as we will throughout this paper) 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.

In the literature, a variety of entropy-like measures have been proposed for quantifying information leakage. But, pleasantly, if we restrict our attention to *deterministic systems* and to their *capacity* (i.e. their maximum leakage over all prior distributions on the secret input), we have the following theorem [3, 7, 8]:

Theorem 1. *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.*

(Note, by the way, that this theorem is consistent with our intuition about the three examples above.)

Given any theory of quantitative information flow, it is desirable to develop automatic techniques for calculating (or at least bounding) the amount of leakage in a system, to verify whether it conforms to a given quantitative flow policy. In previous work [9], the authors developed the approach of *two-bit patterns* to calculate upper bounds on the capacity of deterministic programs. The key idea (justified by Theorem 1 above) 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,

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

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 $\mathit{Zero}(3)$. 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 $\{\texttt{00}, \texttt{11}, \texttt{01}\}$, which we express as $\mathit{Leq}(4, 0)$. Bits 4 and 2, in contrast, can take all four combinations $\{\texttt{00}, \texttt{10}, \texttt{01}, \texttt{11}\}$, which we express as $\mathit{Free}(4, 2)$. The complete two-bit patterns for this example are shown in Figure 1.[1] Two-bit patterns represent constraints that must be satisfied by the bits of each feasible output. So 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.

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

Fig. 1. Two-bit patterns for $\{\texttt{00010}, \texttt{10001}, \texttt{00001}, \texttt{00110}, \texttt{10101}, \texttt{00101}\}$

In the small (but often intricate) case studies in [9], we found that two-bit patterns could be calculated using the STP solver [10] within a few seconds, and

[1] While Figure 1 suggests that n non-fixed bits lead to n^2 two-bit patterns, in fact there are only $\binom{n}{2} = n(n-1)/2$ interesting patterns to determine—the (i, i) patterns are all *Eq*, and each (i, j) pattern follows trivially from the (j, i) pattern.

that they usually (though not always) gave quite accurate bounds on leakage. As an example, consider the "Mix and duplicate" case study:

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

We translate this program into the following STP assertions, where we use the symbol `Y1` to denote the intermediate value of variable `Y`:[2]

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

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

We can then determine the bit-patterns for `Y` by making STP *queries*, which ask whether a given property (e.g. "`Y[3] = Y[19]`?") is a logical consequence of the `ASSERT` statements. In [9], we used one or two STP queries to determine each one-bit pattern, and then used a decision tree of at most four STP queries to determine the two-bit pattern among each pair of non-fixed bits. Here it turns out that all one-bit patterns are *Non-fixed*, and the only interesting two-bit patterns are that bits i and $i + 16$ are equal, for $0 \leq i \leq 15$; all others are *Free*. Finally, we count the number of solutions to the two-bit patterns by using the `SatisfiabilityCount` function of *Mathematica*. Here it is easy to see that there are 2^{16} solutions, implying that there are at most 2^{16} feasible outputs and hence that the leakage from `X` to `Y` is at most $\log 2^{16} = 16$ bits. (In this case, the leakage bound happens to be exact.)

In spite of the successful case studies in [9], the *scalability* of the two-bit pattern approach is a clear concern. Calculating the 32 one-bit and $\binom{32}{2} = 496$ two-bit patterns for the example above required a total of 2032 STP queries, each taking around half a millisecond. A bit of thought, however, makes clear that there is inefficiency in treating the two-bit patterns as being independent of one another. For instance, if we know the patterns $\mathit{Leq}(i, j)$, and $\mathit{Leq}(j, k)$, then we can immediately conclude $\mathit{Leq}(i, k)$ by transitivity, without needing to do additional STP queries. But how can we exploit such dependencies uniformly, over all the various two-bit patterns (*Neq*, *Nand*, *Geq*, *Or*, ...)? More generally, are there effective ways to reduce the number of STP queries needed?

In this paper, we make four main contributions to the efficient calculation of two-bit patterns:

1. We show how to represent the two-bit patterns 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.
2. We show that *random execution* of the program allows us to fill in many entries of the adjacency matrix representation of the implication graph without using STP queries.

[2] A fundamental limitation of this approach, of course, is that it requires us to unroll any program loops completely.

3. We show that using *STP counterexamples* allows us to cheaply fill in many additional entries of the adjacency matrix.
4. We show that, 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.

As we will see, these contributions enable us to reduce the time required for two-bit pattern analysis in each of the case studies in [9]; the reduction averages 72%, and varies from 33% to 99%.

The rest of the paper is structured as follows: Section 2 presents implication graphs and their semantic characterization; Section 3 presents the techniques of random execution, STP counterexamples, and deductive closure; Section 4 gives case studies; Section 5 describes related work; and Section 6 concludes.

2 Our Formal Framework

We model the set of feasible outputs of a program as a set R of *states* ρ. The bits in a state are indexed by a set I of *indices*. (For example, for the 5-bit states modeled in Figure 1, we would have $I = \{0, 1, 2, 3, 4\}$.) Formally, a state ρ is a mapping: $\rho : I \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$.

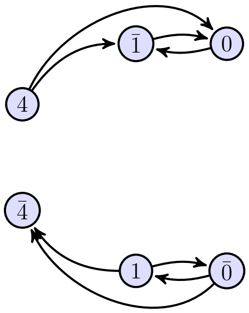
2.1 Implication Graphs

It turns out that there are seven possible two-bit patterns among a pair of non-fixed bits: *Eq*, *Neq*, *Nand*, *Leq*, *Geq*, *Or*, and *Free*. This diversity would seem to complicate a simple representation. However, if we recall that implication and negation are logically complete, we see that each two-bit pattern can be expressed as a set of implications over *literals*, which are indices or negated indices:

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

(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 [11, 12]. As an example, Figure 2 shows the implication graph, in both graphical and adjacency matrix representations, corresponding to the two-bit patterns in Figure 1. (To avoid clutter, we omit self-loops in the graphical representation.)



$$\begin{array}{c|cccccccc}
 & 4 & 2 & 1 & 0 & \bar{4} & \bar{2} & \bar{1} & \bar{0} \\
\hline
4 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
\bar{4} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\bar{2} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\bar{1} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
\bar{0} & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1
\end{array}$$

Fig. 2. The implication graph for $\{\texttt{00010}, \texttt{10001}, \texttt{00001}, \texttt{00110}, \texttt{10101}, \texttt{00101}\}$

Implication graphs have the property of being *skew-symmetric* [13], 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 2 omits mention of the fixed bit 3, since fixed bits do not contribute to multiple outputs. But we could incorporate the one-bit patterns into the implication graph if we wished to; for example, the implication $3 \rightarrow \bar{3}$ expresses that bit 3 must be 0.

2.2 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* [14], 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 ρ iff $\rho(i) \leq \rho(j)$.

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

Definition 1. *Abstraction function* $\alpha : \mathcal{P}(\hat{I} \rightarrow \mathbb{B}) \rightarrow (\hat{I} \times \hat{I} \rightarrow \mathbb{B})$ *is given by*

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

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

Definition 2. *Concretization function* $\gamma : (\hat{I} \times \hat{I} \rightarrow \mathbb{B}) \rightarrow \mathcal{P}(\hat{I} \rightarrow \mathbb{B})$ *is given by*

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

(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 thus *over-approximate* the set of feasible states.

Theorem 2. *Given any set R of states, $R \subseteq \gamma(\alpha(R))$.*

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

Coherent implication graphs behave well with respect to γ and α:

Theorem 3. *If M is coherent, then $\alpha(\gamma(M)) = M$.*

3 Computing Implication Graphs Efficiently

We now consider the question of how we can efficiently compute the implication graph for the set R of feasible outputs of a given program represented (as shown in the "Mix and Duplicate" example in Section 1) as a set of STP assertions.

3.1 One-bit patterns, random execution, and STP counterexamples

As was shown in Figure 2, we include only the *non-fixed* bits in the implication graph. This requires that we begin by determining the one-bit patterns for R. For each bit i of output `Y`, we can first make STP query "`Y`$[i] = 0$?". If this yields *valid*, then bit i is *Zero*. If it yields *invalid*, then it is possible for bit i to be 1, and we can make a second STP query "`Y`$[i] = 1$?" to determine whether bit i is *One* or *Non-fixed*.

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. And, 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.

3.2 Two-bit patterns and deductive closure

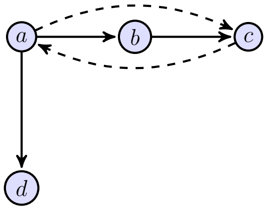
Our goal is to determine the implication graph M using as few STP queries as possible. To this end, it is useful to extend to *partially-known* implication graphs, where we use $\bot$ to denote unknown entries. As in Figure 2, we limit M to the non-fixed bits of $\hat{I}$, which we denote by $\hat{I}_N$. So, formally, $M : \hat{I}_N \times \hat{I}_N \to \mathbb{B}_\bot$, where $\mathbb{B}_\bot$ is the flat domain $\{0, 1, \bot\}$ with partial order $\bot \preceq 0$ and $\bot \preceq 1$. 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 $\bot$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 $M_{i\bar{i}} = 0$. ($M_{i\bar{i}} = 1$ 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 $M_{ji} = 0$ 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 3, 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.



$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & \bot & 1 \\ b & 0 & 1 & 1 & 0 \\ c & \bot & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 1 \end{array}$$

Fig. 3. A partially-known implication graph

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

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
                $M_{ij} \leftarrow 0$;
```

deduces additional 0 entries. To get some intuition, notice that when the first disjunct of the **if** holds, then we have $i \not\to k$ and $j \to k$. Hence $i \to j$ is impossible, as this would yield $i \to k$ by transitivity.

Let $DC(M)$ denote the result of calling *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. *For all M and R, if $M \preceq \alpha(R)$, then $DC(M) \preceq \alpha(R)$.*

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. But we have yet not proved this.

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.

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,

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
        $M_{pq} \leftarrow 1$;
        $M_{\bar{q}\bar{p}} \leftarrow 1$;
        DeductiveClosure1 ($M$);
        DeductiveClosure2 ($M$);
    else
        $M_{pq} \leftarrow 0$;
        $M_{\bar{q}\bar{p}} \leftarrow 0$;
        get counterexample and use it to fill in more 0 entries of $M$;
        DeductiveClosure2 ($M$);
```

the assignments to M_{pq} and $M_{\bar{q}\bar{p}}$ preserve this invariant, as do the calls to DC, by Theorem 4. 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 2) the size of $\gamma(M)$ is an upper bound on the number of feasible outputs.

4 Case Studies

In [9], the authors presented 11 case studies testing both the accuracy and the efficiency of two-bit pattern leakage analysis.[3] In that work, each two-bit pattern was determined individually through STP queries. Here we assess the performance benefits that result from using implication graphs, random execution, STP counterexamples, and deductive closure.

The first point to make is that we are still doing the same two-bit pattern analysis as before, which means that our upper bounds on leakage are exactly the same as before. As discussed in [9], the bounds turn out to be accurate to within one bit in all cases except for "Ten random outputs", where the error exceeds 15 bits.

[3] The code for each case study, along with discussion, can be found in [9], available at `http://users.cis.fiu.edu/~smithg/papers/plas11.pdf`.

Program	O1	O2	OTotal	N0	N1	N2	NTotal	Reduction
Illustrative example	1072	1747	2819	1	805	398	1204	57%
Sanity check, `base`=`0x00001000`	33	21	54	1	28	7	36	33%
Sanity check, `base`=`0x7ffffffa`	66	2040	2106	2	4	197	203	90%
Implicit flow	12	16	28	1	7	3	11	61%
Population count	184	721	905	4	79	96	179	80%
Mix and duplicate	23	863	886	2	0	80	82	91%
Masked copy	9	114	123	2	2	6	10	92%
Binary search, `b`=16	246	4220	4466	1	21	2	24	99%
Electronic purse	210	62	272	4	153	0	157	42%
Sum query	135	100	235	1	113	4	118	50%
Ten random outputs (average)	91	3460	3551	1	7	216	224	94%

Table 1. Old and new times in ms to do two-bit pattern analysis: O1=old time for one-bit patterns, O2=old time for two-bit patterns, OTotal=O1+O2, N0=new time for random executions, N1=new time for one-bit patterns, N2=new time for two-bit patterns, NTotal=N0+N1+N2, Reduction=1 − NTotal/OTotal

Table 1 compares the times (in milliseconds) to do two-bit pattern analysis using our old and new techniques.[4] As can be seen, the times are reduced in all 11 case studies, by an average of 72%; in five cases, the reduction exceeds 90%. But 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 2 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%. As for deductive closure, it contributes in only two of the case studies, but 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 better understand these results, consider the "Sanity Check" program:

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

where `base` is a constant. On this program, random execution will have little benefit, since a randomly-chosen 32-bit value for `X` is highly unlikely to be less

[4] The times reported here for our old analysis (O1 and O2) are faster than those reported in [9], because we have redone our old experiments on a faster computer: a 2.3 GHz Intel Core i3-2310M. Also, because of the randomness in our new techniques, the new timings (N0, N1, and N2) are averages over 10 executions.

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 2. 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 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.

than 16—this is why P1 is just 25% here. STP counterexamples, on the other hand, contribute significantly. Finally, the contribution of deductive closure depends on the bit patterns, which in turn depend on the value of `base`. When `base` is `0x00001000`, only the rightmost four bits of `Y` are non-fixed, and their two-bit patterns are all *Free*, preventing deductive closure from deducing anything; hence we see only a 33% time reduction with this `base`. When `base` is `0x7ffffffa`, on the other hand, the situation is more interesting—all bits are non-fixed and the two-bit patterns are complex (involving *Eq*, *Neq*, *Leq*, *Or*, and *Free* patterns) since the possibility of a string of carries leads to many dependencies among the bits of `Y`. As a result, deductive closure is very helpful here, finding 36% of the entries, and resulting in a time reduction of 90%.

To see the opposite extreme, consider the “Binary Search, `b=16`” program. This program has 16 `if` statements that carry out 16 iterations of binary search, copying the leftmost 16 bits of `X` to `Y`:

```
Y = 0;
if (Y + 2147483648 <= X) Y += 2147483648;   // 2^31
if (Y + 1073741824 <= X) Y += 1073741824;   // 2^30
if (Y + 536870912 <= X) Y += 536870912;     // 2^29
...
if (Y + 65536 <= X) Y += 65536;             // 2^16
```

Here we find that the rightmost 16 bits of `Y` are *Zero*, while the leftmost 16 bits are non-fixed, with only *Free* patterns between them. As a result, deductive closure cannot deduce anything. But random execution, in contrast, 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 them is very beneficial, reducing the analysis time by more than 99%.

5 Related Work

While Yasuoka and Terauchi [15] show that (as one would expect) leakage analysis is computationally difficult, the area is now seeing considerable progress, both in the context of deterministic imperative programs [16–21] and probabilistic systems [22–24]. Techniques currently being explored include model checking, statistical sampling, symbolic execution, and abstract interpretation.

We briefly mention some recent works that focus, as we do, on the calculation of the capacity of deterministic programs. Newsome, McCamant, and Song [17] use a variety of heuristic techniques to estimate the capacity of deterministic x86 binaries; their motivation is actually quantitative *integrity.* Heusser and Malacaria [19] use model-checking techniques to test whether the capacity of a deterministic program exceeds a specified threshold, which needs to be small for the analysis to be feasible. Köpf, Mauborgne, and Ochoa [20] develop an abstract interpretation for bounding capacity, and use it to show bounds on cache leaks in implementations of the AES cryptosystem. Finally, Phan, Malacaria, Tkachuk, and Păsăreanu [21] count the number of feasible program outputs through a symbolic execution technique that is more precise than two-bit patterns, though often more expensive.

6 Conclusion

We have shown that implication graphs, random execution, STP counterexamples, and deductive closure allow two-bit pattern analysis to be done more efficiently. Experiments on a set of 11 case studies show a substantial benefit from the new techniques, with time reductions averaging 72%, and often exceeding 90%. We expect that these improvements will be increasingly important as we scale up to larger and more complex programs.

In future work, we would like to show the completeness of deductive closure, and to improve its efficiency by computing it *incrementally.* Still, 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.

Acknowledgments: This work was partially supported by the National Science Foundation under grant CNS-1116318.

References

1. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation **18**(2) (2005) 181–199
2. Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: Proc. 14th ACM Conference on Computer and Communications Security (CCS '07). (2007) 286–296

3. Smith, G.: On the foundations of quantitative information flow. In de Alfaro, L., ed.: Proc. 12th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS '09). Volume 5504 of Lecture Notes in Computer Science. (2009) 288–302
4. Alvim, M., Andrés, M., Palamidessi, C.: Probabilistic information flow. In: Proc. 25th IEEE Symposium on Logic in Computer Science (LICS 2010). (2010) 314–321
5. McIver, A., Meinicke, L., Morgan, C.: Compositional closure for Bayes risk in probabilistic noninterference. In: Proc. ICALP'10. (2010) 223–235
6. Clarkson, M.R., Schneider, F.B.: Quantification of integrity. In: Proc. 23nd IEEE Computer Security Foundations Symposium (CSF '10). (2010) 28–43
7. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Quantitative notions of leakage for one-try attacks. In: Proc. 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009). Volume 249 of ENTCS. (2009) 75–91
8. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. In: Proc. 25th IEEE Computer Security Foundations Symposium (CSF 2012). (June 2012) 265–279
9. Meng, Z., Smith, G.: Calculating bounds on information leakage using two-bit patterns. In: Proc. Sixth Workshop on Programming Languages and Analysis for Security (PLAS '11). (2011) 1:1–1:12
10. Ganesh, V., Dill, D.L.: 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. (2007) 524–536
11. Krom, M.R.: 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** (1967) 15–20
12. Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters **8**(3) (1979) 121–123
13. Goldberg, A.V., Karzanov, A.V.: Path problems in skew-symmetric graphs. Combinatorica **16**(3) (1996) 353–382
14. Cousot, P., Cousot, R.: Basic concepts of abstract interpretation. In: Building the Information Society. Kluwer Academic Publishers (2004) 359–366
15. Yasuoka, H., Terauchi, T.: Quantitative information flow — verification hardness and possibilities. In: Proc. 23nd IEEE Computer Security Foundations Symposium (CSF '10). (2010) 15–27
16. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proc. 30th IEEE Symposium on Security and Privacy. (2009) 141–153
17. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: Proc. Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09). (2009) 73–85
18. Köpf, B., Rybalchenko, A.: Approximation and randomization for quantitative information-flow analysis. In: Proc. 23nd IEEE Computer Security Foundations Symposium (CSF '10). (2010) 3–14
19. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: Proc. ACSAC '10. (2010) 261–269
20. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: Proc. 24th International Conference on Computer-Aided Verification (CAV '12). (2012) 564–580

21. Phan, Q.S., Malacaria, P., Tkachuk, O., Corina S. Păsăreanu, C.S.: Symbolic quantitative information flow. SIGSOFT Software Engineering Notes **37**(6) (November 2012) 1–5
22. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical measurement of information leakage. In Esparza, J., Majumdar, R., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10). Volume 6015 of Lecture Notes in Computer Science. (2010) 390–404
23. Andrés, M., Palamidessi, C., van Rossum, P., Smith, G.: Computing the leakage of information-hiding systems. In Esparza, J., Majumdar, R., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10). Volume 6015 of Lecture Notes in Computer Science. (2010) 373–389
24. Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies. In: Proceedings of the Computer Security Foundations Symposium (CSF '11). (June 2011) 114–128