

Applied Quantitative Information Flow and Statistical Databases

Jonathan Heusser Pasquale Malacaria

School of Electronic Engineering and Computer Science
Queen Mary University of London
{jonathanh, pm}@dcs.qmul.ac.uk

Abstract We firstly describe an algebraic structure which serves as solid basis to quantitatively reason about information flows. We demonstrate how programs in form of partition of states fit into that theoretical framework.

The paper presents a new method and implementation to automatically calculate such partitions, and compares it to existing approaches. As a novel application, we describe a way to transform database queries into a suitable program form which then can be statically analysed to measure its leakage and to spot database inference threats.

1 Introduction

Quantitative Information Flow (QIF) [5,6] provides a general setting for measuring information leaks in programs and protocols. In QIF programs are interpreted as equivalence relations on input states: two inputs are equivalent if they generate the same observations, e.g. if the program-run on those two inputs terminates with the same output. These equivalence relations form a complete lattice, the Lattice of Information [14] that satisfies nice algebraic properties. Also, once input states are equipped with a probability distribution the equivalence relations correspond to random variables. Information theoretical notions like entropy can be used on these relations to quantify information leaks.

Applied Quantitative Information Flow, i.e. the automatic interpretation of programs in the lattice of information and the related information theoretical computations, is steadily coming of age. Based on a growing number of impressive progress in the field of model checking, SAT solvers, theorem provers and program analysis it is now possible to test quantitative information flow ideas on real code [2,18].

Of course there are still severe limitations of this kind of automatic analysis and it is well possible that most complex code will be out of reach for the foreseeable future. As a comparison a quantitative analysis is intrinsically more complex than a qualitative one and hence we should accordingly moderate our expectations in quantitative tools achieving the same results as qualitative ones anytime soon. There are however important families of programs where automatic analysis is within reach, for example side channel analysis for cryptographic protocols

[13]. Also the integration of quantitative analysis with heuristics and software engineering tools has been successfully demonstrated [18].

In this paper we will introduce an original technique to automatically compute QIF. This technique uses state of the art technology to compute the lattice interpretation of a program. While stressing the general purpose nature of our applied QIF, we investigate in Section 5 the possible application of this tool to a particular field where we believe these techniques have great potential: statistical databases.

1.1 Statistical Databases

Database queries are a major source of information. While data mining tries to maximise the amount of information that can be extracted by a database, security experts work in the opposite direction, i.e. to minimise the confidential information that can be extracted. This paper addresses security issues of statistical databases, i.e. databases where users are allowed to query statistics about confidential data; a typical example would be the average salary in a company: the security threat is that information about an individual salary may be leaked by one or more queries. As a trivial example, knowing that all employees are paid the same amount in conjunction with knowing the average salary will reveal the individual salary of all employees. Ideally, a statistical database security officer should prevent or detect attacks that gain individual information. However, this has been shown as unachievable, for example because of “trackers” [10]. In Section 5 we sketch how applied QIF can be used to measure the amount of confidential information leaked by a set of queries and hence to improve security risk assessment for statistical databases.

In relating QIF to statistical databases, the idea is to interpret a statistical query as a simple program in a programming language and use the interpretation of programs in the Lattice of Information to apply known tools and techniques to measure the amount of confidential information leaked by a set of queries.

1.2 Contributions

The tool presented in Section 4 is original; its relation to DISQUANT is discussed in the same section. Proposition 2 is also original and allows for an automatic and elegant interpretation of databases queries in LoI . Also the ideas in Section 5 about the use of applied QIF in the statistical databases context are original.

2 Lattice of Information

It has been shown by Landauer and Redmond [14] that observations about the behaviour of a deterministic system can be elegantly modelled in terms of a lattice. Let Σ be the set of all states in a system. An observation can be seen as an equivalence class of states defined by $\sigma \sim \sigma'$ if and only if σ and σ' are

indistinguishable given that observation. The join \sqcup and meet \sqcap lattice operations stand for the intersection of relations and the transitive closure union of relations respectively. Thus, higher elements in the lattice can distinguish more while lower elements in the lattice can distinguish less states. It is easy to show that this is a complete lattice of equivalence relations.

The bottom of this lattice is the least informative observation (any two states are equivalent, i.e. all states are equivalent) and the top of the lattice is the most informative observation (each state is only equivalent to itself). Aptly they named this lattice Lattice of Information (**LoI**). The ordering of **LoI** is defined as

$$\approx \sqsubseteq \sim \leftrightarrow \forall \sigma_1, \sigma_2 (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_1 \approx \sigma_2) \quad (1)$$

where $\sigma_1, \sigma_2 \in \Sigma$. An equivalent presentation for the same lattice is in terms of partitions. In fact any equivalence relation can be seen as a partition whose blocks are its equivalence classes. Seen as a lattice of partition we have $\sigma \sqcup \sigma' = \{a \cap b \mid a \in \sigma, b \in \sigma'\}$.

In this paper we will assume this lattice to be finite; this is motivated by considering information storable in programs variables: such information is $\leq 2^k$ where k is the number of bits of the secret variable.

We give a typical example of how these equivalence relations can be used in an information flow setting. Let us assume the set of states Σ consists of a tuple $\langle l, h \rangle$ where l is a low variable and h is a confidential variable. One possible observer can be described by the equivalence relation

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

That is the observer can only distinguish two states whenever they agree on the low variable part. Clearly, a more powerful attacker is the one who can distinguish any two states from one another, or

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

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

A random variable on a finite space can be seen as map $X : D \rightarrow R(X)$, where D is a finite set with a probability distribution and $R(X)$, a measurable set, is the range of X . For each element $d \in D$, the probability of it is denoted $p(d)$. For $x \in R(X)$ $p(x)$ means the probability that X takes on the value x , i.e. $p(x) \stackrel{def}{=} \sum_{d \in X^{-1}(x)} p(d)$. From that perspective, X partitions the space D into sets which are indistinguishable to an observer who sees the value that X takes on. This can be seen as the equivalence relation $\ker(X)$:

$$d \ker(X) d' \text{ iff } X(d) = X(d') \quad (2)$$

The Shannon entropy of a random variable X is denoted $H(X)$, defined as follows

$$H(X) = - \sum_x p(x) \log p(x)$$

As seen from the definition of $p(x)$, the entropy of X only depends on its set of inverse images $X^{-1}(x)$. Thus, if two random variables X and Y have the same inverse images they will necessarily have the same entropy. More formally, we write $X \simeq Y$ whenever the following holds

$$X \simeq Y \text{ iff } \{X^{-1}(x) : x \in R(X)\} = \{Y^{-1}(y) : y \in R(Y)\}$$

and thus if $X \simeq Y$ then $H(X) = H(Y)$.

This shows that each element of the lattice LoI can be seen as a random variable. We can hence identify LoI with a lattice of random variables ordered by (1).

Notice that the \sqcup of two random variables is the classic notion of *joint* random variable, i.e. $X \sqcup Y = (X, Y)$. In general, LoI is not distributive.

2.1 Measures

We can attempt to quantify the amount of information provided by a point in LoI by using lattice theoretic notions, such as semivaluations.

A join semivaluation on LoI is a real valued map $\nu : \text{LoI} \rightarrow \mathbb{R}$, that satisfies the following properties:

$$\nu(X \sqcap Y) + \nu(X \sqcup Y) \leq \nu(X) + \nu(Y) \quad (3)$$

$$X \sqsubseteq Y \text{ implies } \nu(X) \leq \nu(Y) \quad (4)$$

for every element X and Y in a lattice [17]. The property (4) is order-preserving: a higher element in the lattice has a larger valuation than elements below itself. The first property (3) is a weakened inclusion-exclusion principle.

Proposition 1. *The map*

$$\nu(X \sqcup Y) = H(X, Y) \quad (5)$$

is a join semivaluation.

Equation 5 is an important result, firstly described by Nakamura [17]. He proved that the *only* probability-based join semivaluation on the lattice of information is Shannon's entropy. It is easy to show that a valuation itself is not definable on this lattice, thus Shannon's entropy is the best approximation to a probability-based valuation on this lattice.

Other measures can be used, which are however less mathematically appealing. We will also consider Min-Entropy which seems like a good complementing measure. While Shannon entropy intuitively results in an "averaging" measure over a probability distribution, the Min-Entropy H_∞ takes on a "worst-case" view: only the maximal value $p(x)$ of a random variable X is considered

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

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

We write $\mathcal{M}(X)$ to indicate Shannon's entropy or a more general Renyi's entropy.

3 Measuring Program Leakage

In previous works, we developed theories to quantify the information leakage of programs [5,15]. The main idea for deterministic programs is to interpret observations on a program as equivalence relations on states [15,16] and therefore as random variables in the lattice of information. The random variable associated to a program P is the equivalence relation on any states σ, σ' from the universe of states Σ defined by

$$\sigma \simeq \sigma' \iff P(\sigma) =_{\text{obs}} P(\sigma') \quad (6)$$

in this paper $=_{\text{obs}}$ represents the relation “to have the same observable output”. We denote the interpretation of a program P in LoI as defined by the equivalence relation (6) by $\Pi(P)$.

Consider the example `if h=0 then access else deny` where the variable h ranges over $\{0, \dots, 3\}$. The output random variable O associated to the program represents the information available to an observer. The equivalence relation (i.e. partition) associated to the above program is hence

$$O = \underbrace{\{0\}}_{\text{access}} \underbrace{\{1, 2, 3\}}_{\text{deny}}$$

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

The quantitative evaluation of the partition O is measuring such knowledge gains of an attacker, solely depending on the partition of states and the probability distribution of the input.

The next proposition says that we can represent algebraic operations in LoI using programs:

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

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

Given programs P_1, P_2 , we define $P_{1 \sqcup 2} = P'_1; P'_2$ where the primed programs P'_1, P'_2 are P_1, P_2 with variables renamed so to have disjoint variable sets. If the two programs are syntactically equivalent, then this results in self-composition [3]. For example, consider the two programs

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

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

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

The corresponding lattice element is the join, i.e. intersection of blocks, of the individual programs $P_{1,2}$

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

The above result can be extended to expressions of the language: we can associate to an expression e the program consisting of the assignment $\mathbf{x} = \mathbf{e}$ and use Proposition 2 to compute the lub in LoI of a set of expressions. This is the basic technique we will later use for computing leakage of database queries.

Notice that $\Pi(P)$ is a general representation that can be used as the basis for several quantitative measures likes Shannon’s entropy, Renyi entropies or guessability measures, as described in Section 2.

The overarching idea for quantifying the leakage of a partition $\Pi(P)$ is to compute the difference between uncertainty about the secret before and after observing the output of the program. For a Shannon-based measure, leakage is defined in [5,15] as $I(\Pi(P); h|l)$, i.e. the conditional mutual information between the program and the secret given the low input.

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

where equality A holds because the program is deterministic and B holds when the program only depends on the high inputs, i.e. all low variables are initialised in the code of the program. Thus, for such programs, the Shannon-based leakage measure is reduced to simply the Shannon entropy of the partition $\Pi(P)$.

We can relate the order in LoI and the amount of leakage by the following result

Proposition 3. *Let P_1, P_2 be two programs depending only on the high inputs. Then $\Pi(P_1) \sqsubseteq \Pi(P_2)$ iff for all probability distributions on states in LoI , $H(\Pi(P_1)) \sqsubseteq H(\Pi(P_2))$.*

4 Automatically Calculating $\Pi(P)$

The computationally intensive task in quantifying information leaks is calculating the partition of input states $\Pi(P)$. Applying a measure $\mathcal{M}(\Pi(P))$ is in comparison cheap and easy to do (if the probability distribution is known). We developed a tool, AQUA (Automated Quantitative Analysis) which calculates $\Pi(P)$ given a program P in the programming language C without user interaction or code annotations.

The idea is best explained using a similar example from before with 4 bit variable width, and the secret input variable `pwd`:

$$P \equiv \text{if}(\text{pwd} == 4) \{ \text{return } 1; \} \text{ else } \{ \text{return } 0; \}$$

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

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

The next section will look at these two steps in more detail.

4.1 Method

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

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

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

$$\begin{aligned} P_{\neq}(i) &\equiv h = i; P; P'; \mathbf{assert}(1! = 1') \\ P_{=}(i) &\equiv h = i; P; P'; \mathbf{assert}(1 = 1') \end{aligned}$$

The program P is self-composed [3,19] and is either asserting low-equality or low-inequality on the output variable and its copy. Their argument is the initialisation value for the input variable. This method works on any number of input variables, but we simplify it to a single variable.

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

P_{\neq} is solved using a number of SAT solver calls using a standard reachability algorithm (SAT-based fixed point calculation) [12].

Algorithm 1 describes this input discovery. In each iteration it discovers a new input h' which does not lead to the same output as previous the input h . The new input h' is added to the set S_{input} . The observable output l is added to the formula as blocking clause, to avoid finding the same solution again in a different iteration. This process is repeated until P_{\neq} is unsatisfiable which signifies that the search for S_{input} elements is exhausted.

Given S_{input} (or a subset of it) as result of Algorithm 1, we can use $P_{=}$ to count the sizes of the equivalence classes represented by S_{input} using model counting. This process is displayed in Algorithm 2 and is straightforward to understand.

The algorithm calculates the size of the equivalence class $[h]_{P_{=}}$ for every h in S_{input} by counting the satisfying models of $P_{=}(h)$. The output M of Algorithm 2 is the partition $\Pi(P)$ of the original program P .

```

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

```

Algorithm 1: Calculation of S_{input} using P_{\neq}

```

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

```

Algorithm 2: Model counting of equivalence classes in S_{input}

Proposition 4 (Correctness). *The set S_{input} of Algorithm 1 contains a representative element for each possible equivalence class of $\Pi(P)$. Algorithm 2 calculates $\{[s_1]_{P_{=}}, \dots, [s_n]_{P_{=}}\}$ which, according to (6), is $\Pi(P)$.*

4.2 Implementation

The implementation builds up on a toolchain of existing tools, together with some interfacing, language translations, and optimisations. See Figure 1 for an overview.

AQUA has the following main features:

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

AQUA works on the equational intermediate representation of the CBMC bounded model checker [7]. C code is translated by CBMC into a program of constraints which in turn gets optimised through standard program analysis techniques into cleaned up constraints¹. This program then gets self-composed and user-provided source and sink variables get automatically annotated.

¹ CBMC adds some constraints which distorts the model counting.

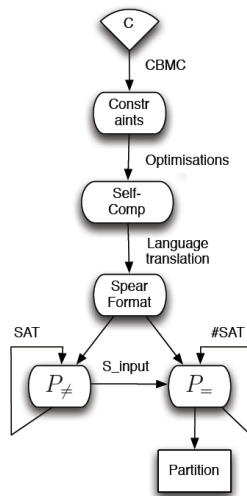


Figure 1. Translation steps

In a next step, the program gets translated into the bit-vector arithmetic SPEAR format of the SPEAR theorem prover [1]. At this point, AQUA will spawn the two instances, $P_=_$ and $P_≠$, from the input program P .

Algorithms 1 and 2 get executed sequentially on those two program versions. However, depending on the application and cost of the SAT queries, one could also choose to execute them interleaved, by first calculating one input to the program $P_=_$ and then model counting that equivalence class.

For Algorithm 1, SPEAR will SAT solve $P_≠$ directly and report the satisfying model to the tool. The newly found inputs are stored until $P_≠$ is reported to be unsat.

For Algorithm 2, SPEAR will bit-blast $P_=_$ down to CNF which in turn gets model counted by either RELSAT [4] or C2D. C2D is only used in case the user specifies fast model counting through command line options. While the counting is much faster on difficult problems than RELSAT, the CNF instances have to be transformed into a d -DNNF tree [8] which is very costly in memory. This is a trade-off between time and space. In most instances, RELSAT is fast enough, except in cases with multiple constraints on more than two secret input variables.

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

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

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

Table 1. Performance examples. * 30 loop unrollings; † from [2]; ★ counted with *C2D* Machine: Linux, Intel Core 2 Duo 2GHz

Algorithm 1 detects at one point an input which contains all inputs beyond the iteration bound. Using the principle of maximum entropy, this “sink state” can be used to always safely over-approximate entropy.

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

Partition:

{241}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}{1}: 256

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

Proposition 5 (Sound loop leakage). *Let us assume partition $\Pi(P)_n$ is the result of n unwindings of P , and $\Pi(P)_m$ is m unwindings of P , where $m \geq n$. If every element of the “sink state” block $b \in \Pi(P)_n$ is distributed in individual blocks, the partition denoted as $\hat{\Pi}(P)_n$, then $\Pi(P)_m \sqsubseteq \hat{\Pi}(P)_n$. From Proposition 3 follows that $H(\Pi(P)_m) \sqsubseteq H(\hat{\Pi}(P)_n)$.*

Experiences. Table 1 provides a small benchmark to give an idea on what programs AQUA has been tested on. The running times have been split between Algorithm 1 to calculate P_{\neq} and the total run time; also it provides the lines of code (LOC) the program has in SPEAR format.

The biggest example is a full CRC8 checksum implementation where the input is two `char` variables (16 bit) which has over 700 LOC.

The run time depends on the number of secrets and their ranges and as a result on the cardinality of the partition. The programs are available from the first author’s website.

4.3 Comparison to DISQUANT

Recently, Backes, Köpf, and Rybalchenko published an elegant, and inspiring method to calculate and quantify an equivalence relation given a C-like program [2].

Their tool DISQUANT turns information flow checking into a reachability problem by self-composing a program and then applying a model checker to find pairs of inputs which violate the secure information flow safety property. This will result in a logical formula of secret relations. Out of the relation, equivalence classes and their sizes can be calculated. The relation is built by a guided abstraction-refinement technique. This means that the tool starts with a blank canvas where every high input is related to each other. Then it successively refines an equivalence relation R by learning that some input pairs (h_i, h_j) lead to different outputs, in which case a new equivalence class is added to the relation.

However, before DISQUANT can say anything about the (size of the) partition it has to complete the refinement process; intermediate results in the CEGAR refinement process might not be useable for quantification.

In comparison, our method is in a way the opposite: it calculates a whole equivalence class for one input, independent of the remaining equivalence classes. This has multiple advantages: it is easy to distribute the computation for different inputs and model counting over multiple computers; for some problems, not all equivalence classes need to be calculate and the computation can stop after finding certain properties, e.g. finding a too large/small equivalence class for a given policy; we can calculate a partition for a subset of inputs; we can provide incremental lower bounds on the leakage.

Two additional differences between the two approaches are worth mentioning separately: Due to the bit precise modelling of arithmetic operators and overflows our tool can handle non-linear constraints on secret inputs, while DISQUANT is limited by its underlying techniques to linear arithmetic.

Also, Algorithm 2 in our tool is not only able to count the number of elements in an equivalence class but also enumerate the models. While it is prohibitive to completely enumerate large equivalence classes, it is still possible to extract example models which could be used for some purposes.

5 Database Queries as Programs

We will now describe how we can model statistical database queries as programs. Once a database query has been modelled as a program, we can apply our program analysis tools to calculate the partition of states and in turn quantify the leakage of the queries. This section is not about showcasing AQUA's performance but to illustrate the width of applications of applied QIF.

We will use concepts used by Dobkin et al. [11] to describe databases

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

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

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

An arbitrary query is translated by the following transformation

$$Q_1 = q(d_i, \dots, d_j) \quad \Rightarrow \quad \mathbf{l}_1 = \mathbf{e}(\mathbf{h}_i, \dots, \mathbf{h}_j)$$

where the function q applied to (d_i, \dots, d_j) is rewritten to some C expression e^2 on the secret variables $\mathbf{h}_i, \dots, \mathbf{h}_j$, where \mathbf{h}_n is equal to d_n for all $i \leq n \leq j$; the output is stored in the observable variable \mathbf{l}_1 . A sequence of queries Q_1, \dots, Q_n results in tuples of observable variables $(\mathbf{l}_1, \dots, \mathbf{l}_n)$. We denote the partition of states for a query Q_i , after the transformation above, as $\Pi(Q_i)$.

5.1 Database Inference by Examples

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

Definition 2 (SDB Leakage Ratio). *Given an SDB, let Q_1, \dots, Q_n be queries, and h_1, \dots, h_m be the involved secret elements in the database. The percentage of leakage revealed by the sequence of queries is given by*

$$\frac{\mathcal{M}(\bigsqcup_{1 \leq i \leq n} \Pi(Q_i))}{\mathcal{M}(h_1, \dots, h_m)} \quad (7)$$

In the definition we can use Proposition 2 to compute $\bigsqcup_{1 \leq i \leq n} \Pi(Q_i)$

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

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

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

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

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

Assuming 2 bit variables, we get the following calculations:

$$\begin{aligned} H(\Pi(Q_1)) &= 1.7490 & H(\Pi(Q_2)) &= 2.6556 & H(\Pi(Q_1) \sqcup \Pi(Q_2)) &= 4.4046 \\ H(\Pi(Q'_1)) &= 1.7490 & H(\Pi(Q'_2)) &= 2.6556 & H(\Pi(Q'_1) \sqcup \Pi(Q'_2)) &= 3.25 \end{aligned}$$

² Expressions usually used in statistical database are **sum, count, average, mean, median** etc.. our context is however general so any C expression can be used

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

Table 2. Contributors

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

Table 3. Summary Table for Contributors

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

$$\frac{H(\Pi(Q_1) \sqcup \Pi(Q_2))}{H(h_1, h_2, h_3, h_4)} = \frac{4.4046}{8.0} \approx 55\% \quad \frac{H(\Pi(Q'_1) \sqcup \Pi(Q'_2))}{H(h_1, h_2)} = \frac{3.25}{4.0} \approx 81\%$$

where we have used H , the Shannon entropy as the leakage measure³. The 3.25 bits, or 81% of the secret, is the maximal possible leakage for the query, as we still don't know which of the two secrets secret was the bigger one of the two, however "everything" is leaked in a sense, while the first query only reveals 55% of the secret space.

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

Sum Queries Inference. Consider a database storing donations of contributors to a political party from the steel and sugar industry, contributors coming from several geographical areas. Given Tables 2 and 3, a user is allowed to make sum queries on all contributors which share a common attribute (Industry or Geographic Area)⁴. Table 3 summarises all possible queries, where the amount donated by each contributor C_i is represented by the value h_i .

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

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

User 1 is making two queries

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

In other words, User 1 is asking for the sum of the contributors from the steel and sugar industry. For simplicity, we assume only 2 bit variables for each contributor

³ Taking a different measure like min entropy we would get 40% and 75% respectively.

⁴ Example adapted from [11].

h_i . AQUA calculates a partition with 100 equivalence classes, and a Shannon entropy of 5.9685 of total 12 bits.

This results in a ratio of

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

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

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

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

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

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

where in the second equation term h_1 in the denominator disappear because contributor C_1 knows h_1 (similarly $Q'_4 = \text{sum}(h_4, h_5)$)⁵. If our tool was evaluating the information leakage of these queries before the result was reported back to the user, then Q_4 could be denied for User 2.

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

6 Related Work

The closest work to ours is the one reviewed in Section 4.3. Another impressive method to quantify information flows in large programs is described by McCamant in multiple works [18]. The lattice of information has been described by Landauger and Redmond [14]. There is a large and inspiring literature of Information theoretical notions in statistical databases e.g. [9,20]. No work however has so far used applied QIF for queries analysis.

⁵ To understand the numbers 4.6556 comes by the fact that the queries reveal h_6 i.e. 2 bits, plus $\text{sum}(h_4, h_5)$ which is 2.6556 bits

References

1. Domagoj Babić and Frank Hutter: Spear Theorem Prover. Proc. of the SAT 2008 Race, 2008
2. Michael Backes and Boris Köpf and Andrey Rybalchenko: Automatic Discovery and Quantification of Information Leaks. Proc. 30th IEEE Symposium on Security and Privacy (S&P '09), to appear.
3. Barthe, Gilles and D'Argenio, Pedro R. and Rezk, Tamara: Secure Information Flow by Self-Composition. CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations.
4. R. Bayardo, R. Schrag: Using CSP look-back techniques to solve real-world SAT instances. In Proc. of AAAI-97. pp. 203-208. AAAI Press/The MIT Press, 1997.
5. David Clark, Sebastian Hunt, Pasquale Malacaria: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security, Volume 15, Number 3 / 2007.
6. David Clark, Sebastian Hunt, and Pasquale Malacaria: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language, 18(2):181-199, 2005.
7. Clarke, Edmund, and Kroening, Daniel, and Lerda, Flavio: A Tool for Checking ANSI-C Programs. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Springer, 168-176, Volume 2988
8. Adnan Darwiche and Pierre Marquis: A Knowledge Compilation Map. Journal of Artificial Intelligence Research, 2002, 229-264 Volume 17.
9. Dorothy E. Denning, Cryptography and Data Security, Addison-Wesley
10. Dorothy E. Denning, Jan Schlrer, A fast procedure for finding a tracker in a statistical database, ACM Transactions on Database Systems, Volume 5, Issue 1 (March 1980) . Pages: 88 - 102
11. David Dobkin and Anita K. Jones and Richard J. Lipton: Secure databases: Protection against user influence. ACM Transactions on Database Systems, 1979, Volume 4, 97-106.
12. Pankaj Chauhan and Edmund M. Clarke and Daniel Kroening: Using SAT based Image Computation for Reachability. Carnegie Mellon University, Technical Report CMU-CS-03-151, 2003.
13. Boris Köpf and David Basin: An information-theoretic model for adaptive side-channel attacks. CCS '07: Proceedings of the 14th ACM conference on Computer and communications security, 2007, 286-296
14. Landauer, J., and Redmond, T.: A Lattice of Information. In Proc. of the IEEE Computer Security Foundations Workshop. IEEE Computer Society Press, 1993.
15. Pasquale Malacaria: Assessing security threats of looping constructs. Proc. ACM Symposium on Principles of Programming Language, 2007.
16. Pasquale Malacaria: Risk Assessment of Security Threats for Looping Constructs, to appear in the Journal Of Computer Security, 2009.
17. Y. Nakamura. Entropy and Semivaluations on Semilattices. Kodai Math. Sem. Rep 22 (1970), 443-468
18. Stephen Andrew McCamant: Quantitative Information-Flow Tracking for Real Systems. MIT Department of Electrical Engineering and Computer Science, Ph.D., Cambridge, MA, 2008.
19. T. Terauchi and A. Aiken. Secure information flow as a safety problem: In SAS, volume 3672 of LNCS, pages 352-367, 2005.
20. E.A.Unger, L.Harn and V.Kumar Entropy as a Measure of Database Information, Proceedings Sixth ACSAC, IEEE 1990