# SAT-based Analysis and Quantification of Information Flow in Programs

Vladimir Klebanov[1], Norbert Manthey[2], and Christian Muise[3]

[1] Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
klebanov@kit.edu
[2] Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@iccl.tu-dresden.de
[3] Department of Computer Science
University of Toronto, Toronto, Canada
cjmuise@cs.toronto.edu

**Abstract.** Quantitative information flow analysis (QIF) is a portfolio of security techniques quantifying the flow of confidential information to public ports. In this paper, we advance the state of the art in QIF for imperative programs. We present both an abstract formulation of the analysis in terms of verification condition generation, logical projection and model counting, and an efficient concrete implementation targeting ANSI C programs. The implementation combines various novel and existing SAT-based tools for bounded model checking, #SAT solving in presence of projection, and SAT preprocessing. We evaluate the technique on synthetic and semi-realistic benchmarks.

## 1 Introduction

Quantitative information flow analysis (QIF) is a collection of techniques for security assessment of software. The research in QIF is motivated by the observation that it is not feasible to completely prevent information leaks (i.e., the flow of confidential information to public ports) in realistic systems. Instead, practical security analysis demands a measure of leaked information in order to decide if a leak is tolerable.

QIF techniques have been applied to a variety of problems. Deciding whether a PIN generation algorithm produces PINs that are hard to guess [1], or whether a particular image transformation is a secure anonymization mechanism (cf. Figure 1) [17] are examples of QIF applications. While the information-theoretical foundations of QIF in deterministic programs are relatively well-understood, practical analysis techniques and tools are still under development.

So far, QIF analyses have been typically described *operationally*, i.e., with focus on algorithm development. One contribution of this paper is an *abstract* formulation, describing a whole class of QIF analyses in terms of verification

condition generation, logical projection (most notably), and model enumeration/counting. This view facilitates understanding and comparison of existing approaches and better connects QIF to the existing body of work in these areas.

Inspired by this connection is another contribution of this paper: a novel combination of SAT preprocessing, projection, and counting, resulting in a QIF analysis that is more efficient than its predecessors. Our toolchain for analysis of C programs consists of an off-the-shelf bounded model checker CBMC [6], a propositional formula preprocessor that we developed previously, and three tools for propositional model enumeration/counting under projection that we developed (resp. extended) for this paper.

The discerning properties of our analysis are: (1) The implemented analysis is general-purpose, i.e., it is not taylored to a particular software application domain. No restrictions on the shape of the indistinguishability relation are posed (in contrast to [2, 14, 13]). The implementation supports almost all of ANSI C by virtue of using CBMC. (2) The analysis is not compositional, but it is fully automated—the only required user input is the program under analysis. Loops are handled by bounded unwinding, which is computationally expensive, but fully automatic and complete (with unwinding assertion checking). (3) The analysis supports measuring both the conditional min-entropy (counting the number of program outputs) and conditional Shannon entropy (counting output preimage sizes). (4) The analysis is, conceptually, sound and precise (in contrast to [17, 18, 21]). Of course, QIF is a hard problem, so computational constraints may force the user to settle for merely deriving (more or less tight) leak bounds as program complexity increases. (5) The analysis outpferforms comparable previous approaches both for theoretical reasons (e.g., it avoids computationally expensive program self-composition used in [11, 12, 2, 13]) and practically due to the use of a number of well-connected novel and existing state-of-the-art techniques and tools for propositional reasoning.

(a) Original image

(b) Image (a) swirled by 720 degrees

(c) Image (b) swirled by −720 degrees

Fig. 1: Anonymization by image swirling. Details in Section 6.3

## 2 QIF Basics and Technical Preliminaries

**Programs, states, and transition relation.** A *program state* is a semantical structure assigning values to mutable program vocabulary of a program $p$. Let $S$ be the set of all (program) states for $p$. A program $p$ induces a *transition relation* $\rho_p \subseteq S \times S$ on states as follows: $(s, s') \in \rho_p$ iff $p$ started in state $s$ terminates in a state $s'$. A security analysis may sometimes wish to focus on a particular set of initial states $S_I \subseteq S$. In this case $\rho_p \subseteq S_I \times S$.

We only consider programs that are written in a deterministic (read: sequential) programming language and are terminating, i.e., we require that $\rho_p$ is a total function. The termination requirement is enforced by model checking (see Section 4). We call a pair of an initial and a final state $(s, s') \in \rho_p$ a *run* of $p$.

Unless stated otherwise, we establish the convention that the program takes its input in the variable $I$ and produces its output in the variable $O$. The shorthand phrase *value of* $I$ resp. $O$ is to be understood as referring to the value in the initial resp. final state of a given run of $p$. Whenever necessary, $I$ and $O$ are silently lifted to be vectors (with $I \cap O = \emptyset$). A treatment of C structs as program output is shown in Section 6.2.

More amenable to reasoning is a description of $\rho_p$ by a logical formula with two free variables $I$ and $O$. We denote such formula as $\langle p \rangle(I, O)$ or, later, simply $\langle p \rangle$. The formula $\langle p \rangle(I_0, O_0)$ evaluates to TRUE iff $p$ started with the input denoted by $I_0$ terminates with output denoted by $O_0$.

**Attacker model and indistinguishability relation.** We assume that the attacker knows the program $p$, and that the input $I$ is secret and the output $O$ is public. The attacker has observed the value of the output $O$ in a final state of some run of $p$ and wants to learn something about the value of $I$ in the initial state. It is the goal of QIF analysis to measure $p$'s vulnerability to such an attack.

In the above attacker model, each program induces a partition on secret inputs $\approx_p$ called the *indistinguishability relation*. Each block in this partition is a set corresponding to some output value of the program and containing exactly the input values leading to this output. Formally, $\approx_p = \{\rho_p^{-1}(s') \mid s' \in \rho_p \circ S_I\}$. One also speaks of blocks in $\approx_p$ as *preimages* of program outputs. For example, if $I$ is an unsigned 32-bit integer, then the program `if(I==42) O=1 else O=0;` induces $\approx_p = \{\{0, \ldots, 41, 43, \ldots, 2^{32} - 1\}, \{42\}\}$.

Intuitively, an attacker can discern secret inputs from different blocks but not within one block. Secure programs have a coarse $\approx_p$, while insecure a fine one. If $\approx_p$ is identity (very fine), then all blocks are singleton sets, and each output corresponds uniquely to a secret input: the attacker has perfect knowledge. Conversely, the coarsest indistinguishability relation $\approx_p = S_I \times S_I$ with only one block means that the attacker learns nothing about the secret inputs by observing program outputs (a scenario known as "non-interference").

Sometimes, a more powerful attacker is considered who can observe multiple runs while partially choosing the program inputs (so called *low inputs*). In this case, the indistinguishability relation becomes parametrized by a set of actualized

low inputs $L$. If the set $L$ is small, the QIF problem can be reduced to the no-low-input case by calculating the cartesian product of outputs for each low input $l \in L$. If the set $L$ is large, other approaches (typically based on computationally more expensive self-composition) must be used.

**Quantitative security measures.** Given the number and sizes of blocks in $\approx_p$, it is possible to compute a range of security measures summarizing information flow (leakage) in a program. The leaked information is the difference between the attacker's initial uncertainty about the secret inputs and the remaining (a.k.a. residual) uncertainty after observing the output of the program [23].

It should be noted that different security measures have different properties and are appropriate for different scenarios. It may also be necessary to consider several measures in order to give dependable operational guarantees. We focus on two popular measures and refer to [23] for an in-depth discussion.

For quantification purposes, we interpret $I$ and $O$ as random variables ranging over $S_I$ and $S$ respectively. The program $p$ restricts the values of $I$ and $O$ that can occur simultaneously. We assume that $I$ follows a uniform distribution, i.e., that all secret inputs are equally likely. If this is not the case, techniques exist for reducing the analysis to a uniform case [1].

Under these assumptions, and given $\approx_p = \{C_1, \ldots, C_n\}$ ($n$ is, thus, the total number of possible distinct outputs of $p$), the following measures can be computed [23, 2]:

$$H_\infty(I|O) = \log_2 \frac{|S_I|}{n} \qquad \text{and} \qquad H(I|O) = \frac{1}{|S_I|} \sum_{i=1}^{n} |C_i| \log_2 |C_i|$$

where the *conditional min-entropy* $H_\infty(I|O)$ is a measure in bit reflecting the probability of correctly determining $I$ in a single guess after observing $O$, and the *conditional Shannon entropy* $H(I|O)$ is a lower bound in bit on the expected message length needed to communicate the remaining secret about $I$ after observing $O$.

## 3 Analysis, Abstractly

In this section, we formulate our QIF framework in abstract logical terms. Our implementation, described later, is based on propositional logic, but other logics supporting model generation (e.g., QF_ABV) could be used just as well. We assume that logical formulas are built from usual logical connectives ($\wedge$, $\vee$, $\neg$, etc.) and user-defined vocabulary $\Sigma$. In propositional logic, $\Sigma$ is a set of propositional variables. A *model* is a logic-specific semantical structure used to give meaning to user-defined vocabulary of a formula. In propositional logic, a model $m\colon \Sigma \to \{\textsc{True}, \textsc{False}\}$ is a map assigning every variable in $\Sigma$ a truth value. In general, a given model $m$ can be homomorphically extended to give a truth value to a formula $\Phi$ according to standard rules for logical connectives. We call a model $m$ a *model of* $\Phi$, if $m$ assigns $\Phi$ the value True. A formula $\Phi$ is *satisfiable* if it has at least one model, and *unsatisfiable* otherwise.

**Definition 1.** *We build our analysis from a number of abstract operators, which we define below, using the following designations.* $\Sigma$ *and* $\Delta$ *are vocabularies with* $\Delta \subseteq \Sigma$*. A* $\Sigma$*-entity (i.e., formula or model) is an entity defined (only) over vocabulary from* $\Sigma$*. In the following,* $\Phi$ *is a* $\Sigma$*-formula,* $\Psi$ *is a* $\Delta$*-formula,* $i$ *is an integer,* $m$ *is a* $\Sigma$*-model,* $m_1$ *is a* $\Delta$*-model,* $M$ *is a set of models,* $p$ *is a program,* $I$ *and* $O$ *are program variables.*

| *Expression* | *Meaning* |
|---|---|
| $\Phi := \langle p \rangle$ | *formula encoding the behaviors of* $p$ *(i.e., its transition relation or the set of traces)* |
| $\Delta := \langle I \rangle, \Delta := \langle O \rangle$ | *vocabulary denoting in* $\langle p \rangle$ *the input and output variables of* $p$ *(while* $p$ *is implied)* |
| $m := model(\Phi)$ | *some model satisfying* $\Phi$*. If* $\Phi$ *is unsatisfiable, the result is a special value* $\perp$*.* |
| $M := models(\Phi)$ | *the set of all models satisfying* $\Phi$*. If* $\Phi$ *is unsatisfiable, the result is the empty set* $\emptyset$*.* |
| $i := count(\Phi)$ | $i := \lvert models(\Phi) \rvert$ *(number of models satisfying* $\Phi$*)* |
| $m_1 := m\big\vert_{\Delta}$ | *the* $\Delta$*-model that coincides with the* $\Sigma$*-model* $m$ *on the vocabulary* $\Delta$ |
| $\Psi := \Phi\big\vert_{\Delta}$ | *the strongest* $\Delta$*-formula that, when interpreted as a* $\Sigma$*-formula, is entailed by* $\Phi$ *(projection of* $\Phi$ *on* $\Delta$*).* $models(\Phi\big\vert_{\Delta}) = \{m\big\vert_{\Delta} \mid m \in models(\Phi)\}$ |
| $\Psi := \Delta \simeq m_1$ | *a* $\Delta$*-formula that is true in* $m_1$ *and false in all other* $\Delta$*-models.* |
| $\Psi := \Delta \not\simeq m_1$ | $\neg(\Delta \simeq m_1)$*, a* $\Delta$*-formula that is false in* $m_1$ *and true in all other* $\Delta$*-models.* |

The most interesting operator in the list above is projection. It makes the formula $\Phi\big\vert_{\Delta}$ say the same things about $\Delta$ as $\Phi$ does—but nothing else. Projection allows isolating aspects of program behavior along syntactical boundaries. For instance, the formula $\langle p \rangle\big\vert_{\langle O \rangle}$ describes (just) the set of outputs that are compatible with the behavior of program $p$. Orthogonally, the formulas $\Delta \simeq m_1$ and $\Delta \not\simeq m_1$ allow—when conjoined with $\langle p \rangle$—selecting or rejecting particular runs of the program. These formulas are easier to illustrate if the underlying logic is first-order; an implementation in propositional logic is given later. For instance, the first-order formula $\langle p \rangle \wedge \langle O \rangle = 5$ would be a particular instance of $\langle p \rangle \wedge \langle O \rangle \simeq m_1$ (for $m_1$ where $\langle O \rangle$ has the value 5) and describe all those runs of $p$ that terminate with $O = 5$. Employing projection, we can describe the set of inputs that produce the output $O = 5$ by $(\langle p \rangle \wedge \langle O \rangle = 5)\big\vert_{I}$. In this light, we now formulate a general result:

**Proposition 1.**

$$H_\infty(I|O) = log_2 \frac{|S_I|}{count(\langle p \rangle\big\vert_{\langle O \rangle})} \quad \text{and} \quad H(I|O) = \frac{1}{|S_I|} \sum_{o \in M} |C(o)|\, log_2 |C(o)|$$

*where* $M = models(\langle p \rangle\big\vert_{\langle O \rangle})$ *and* $|C(o)| = count(\big(\langle p \rangle \wedge \langle O \rangle \simeq o\big)\big\vert_{\langle I \rangle})$.

We note that computing $H(I|O)$ requires model enumeration *and* counting, while $H_\infty(I|O)$ only requires counting. We also note that since searching for models is computationally expensive, determining the residual min-entropy is easier the more secure the program of a given complexity is (fewer blocks in $\approx_p$). This does not hold for the Shannon entropy, as there is a tension between the number and size of blocks in $\approx_p$ (fewer blocks entail larger block sizes and vice versa).

## 4 From Program to Transition Relation with Bounded Model Checking

**SAT-based bounded model checking.** To implement the $\langle\cdot\rangle$ operator for translating programs into (propositional) logic, we use the SAT-based model checker CBMC [6] for C programs. CBMC is a very mature and popular verification tool supporting almost all ANSI C language features, including pointer constructs, dynamic memory allocation, recursion, and the float and double data types [6]. A similar, if less mature, tool for Java is JForge [8].

Given a C program $p$ and a specification *spec* (given by `assert` statements in the code), CBMC generates a formula $\langle p\rangle \wedge \neg\langle spec\rangle$ in propositional logic, where $\langle p\rangle$ encodes the behaviors of the program $p$, and $\neg\langle spec\rangle$ encodes the behaviors that a specification-compliant program should not exhibit. This *verification condition* $\langle p\rangle \wedge \neg\langle spec\rangle$ is passed to a SAT solver. If it is unsatisfiable, then the program is correct w.r.t. the specification; otherwise, any model of $\langle p\rangle \wedge \neg\langle spec\rangle$ describes a violation of the specification.

During CBMC operation, functions are inlined and loops are unwound to the user-specified depth. CBMC warns the user if the unwinding depth is insufficient to cover all of the program behaviors (this is known as *unwinding assertion* checking). The unwound program is transformed into the static-single-assignment (SSA) form. In this form, statements can be interpreted as equations over bit vectors. The equations are combined and reduced to a formula of propositional logic in a process resembling synthesis of arithmetic circuits. The formula is flattened into conjunctive normal form (CNF) $\bigwedge_i \bigvee_j L_{i,j}$, where each literal $L_{i,j}$ is either a propositional variable or its negation. The formula can be exchanged with other tools by means of a standard DIMACS format.

**Translating programs into logic.** First, we carry out a preliminary verification pass, during which we incrementally increase the unwinding depth until CBMC reports no more unwinding assertion violations. This ensures that all program behaviors are covered and also that the program terminates for all inputs. In the main CBMC pass, we augment the program with the specification `assert(0);` (i.e., an assertion that is never fulfilled) before each return statement and make CBMC export the verification condition formula $\langle p\rangle \wedge \neg\langle spec\rangle$. The specification reduces the $\neg\langle spec\rangle$ conjunct to *true*, leaving the desired $\langle p\rangle$. The process may consume large amounts of memory but it is not a computational bottleneck as long as the unwinding depth is reasonable. The runtimes in our examples ranged from instantaneous to under a minute.

**Identifying program variables in the transition relation formula.** Internally, CBMC represents each program variable bit-wise according to its type (and machine architecture). For example, the initial value of a `char`-typed program variable is represented by 8 propositional variables. More precisely, CBMC tracks the evolution of each program variable over a series of *time frames* (relative to each variable). Typically, we are only interested in time frame one for $I$ (i.e., initial state) and the highest time frame for $O$ (final state). The mapping from program variables and time frames to sets of propositional variables is embedded as comments in the CBMC-generated formula (lines starting with `c`, at the bottom of the DIMACS file). These comments have the following structure: `c` *function_id* `::` *prg_var_id* `!` *thr_nr* `@` *rec_depth* `#` *time prop_var_list*. Thus, `c c::main::1::I!0@1#1 1 2 3 4 5 6 7 8` means that the variable `I` in function `main` in thread 0 at recursion depth 1 during time frame 1 is represented by propositional variables $v_1, \ldots, v_8$. We extract this information with a simple parser.

## 5 Model Enumeration and Counting

In this section, we present two conceptually different approaches and three tools that we developed to implement $\mathit{models}(\Phi\big|_\Delta)$ resp. $\mathit{count}(\Phi\big|_\Delta)$.

### 5.1 Iterative Model Enumeration/Counting

**Proposition 2.** *The algorithm shown in Figure 2 implements model enumeration of a formula under projection.*

We have implemented this projection-capable version of a well-known model enumeration algorithm in a tool named SHARPCDCL[4]. The basic $\mathit{model}(\Phi)$-finding functionality is offered by the SAT solver MINISAT [9]. Implementing model projection $m\big|_\Delta$ is trivial, as one simply restricts the domain of the mapping $m$ to the scope $\Delta$. The formula $\Delta \not\simeq m\big|_\Delta$ can be constructed as $\bigvee_{v\in\Delta} \mathit{flip}(v, m)$, where $\mathit{flip}(v, m) \equiv v$, if $m(v) = \textsc{False}$, and $\mathit{flip}(v, m) \equiv \neg v$, if $m(v) = \textsc{True}$. This way, the truth value of at least one variable in $m\big|_\Delta$ must flip in order to satisfy $\Delta \not\simeq m\big|_\Delta$. Conjoining this formula (which is already in CNF) with $\Phi$ ensures that the current model $m$ will not be found again. After the loop terminates (or SHARPCDCL is interrupted), the set resp. number of found models is returned.

**input** : $\Sigma$-Formula $\Phi$,
projection
scope $\Delta \subseteq \Sigma$
**output**: $\mathit{models}(\Phi\big|_\Delta)$

$M \leftarrow \emptyset$
$m \leftarrow \mathit{model}(\Phi)$
**while** $m \neq \bot$ **do**
  $M \leftarrow M \cup m\big|_\Delta$
  $\Phi \leftarrow \Phi \wedge (\Delta \not\simeq m\big|_\Delta)$
  $m \leftarrow \mathit{model}(\Phi)$
**end**
**return** $M$

Fig. 2: An algorithm for enumerating $\mathit{models}(\Phi\big|_\Delta)$

[4] Available at `http://tools.computational-logic.org/`

### 5.2 Model Counting via Compilation to d-DNNF

State-of-the-art deterministic #SAT solvers implement $count(\cdot)$ via compilation of the formula to Deterministic Decomposable Negation-Normal Form (d-DNNF). We have extended two such tools[5], SHARPSAT [25] and DSHARP [20], with projection capabilities—something that has not been available in #SAT solvers so far. While iterative model enumeration/counting works better on large formulas with few models, d-DNNF-based #SAT solvers are useful to analyze smaller formulas with a large number of models. Empirical evidence is presented in Section 6. Below, we briefly sketch the necessary theoretical results for integrating model counting and projection.

**Definition 2.** *A formula in d-DNNF is a rooted tree such that:*

- *The label of each leaf node is either true, false, or a literal (i.e., negation can only appear attached to variables), while the label of each internal node is either a conjunction ($\wedge$) or a disjunction ($\vee$).*
- *Decomposability holds: any two children $c_i$ and $c_j$ of a conjunctive node share no vocabulary: $\Sigma_{c_i} \cap \Sigma_{c_j} = \emptyset$.*
- *Determinism holds: let $\Phi(n)$ be the formula represented by the subtrees rooted at node $n$. For any two children $d_i$ and $d_j$ of a disjunctive node, $\Phi(d_i)$ and $\Phi(d_j)$ must be contradictory, i.e., $\Phi(d_i) \wedge \Phi(d_j)$ is unsatisfiable.*

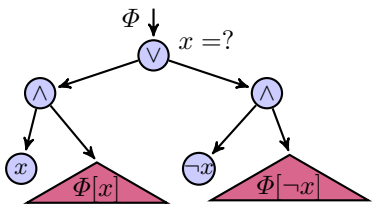


Fig. 3: A typical d-DNNF fragment

A propositional formula is typically compiled to d-DNNF by an exhaustive DPLL-style algorithm alternating systematic case distinctions (decisions) and unit propagation. Each decision gives rise to an $\vee$-node as per equality $\Phi = \Phi[v] \vee \Phi[\neg v]$. Figure 3 illustrates such a decision on variable $x$. The #SAT solvers also employ a number of optimizations (e.g., subtree caching, clause learning, etc.), but these are of no interest here. After a (computationally hard) compilation to d-DNNF, both projection computation and model counting—though not in combination—can be carried out in linear time [7].

**Proposition 3.** *If $\Phi$ is a $\Sigma$-formula in d-DNNF, then $\Phi\big|_\Delta$ can be computed in polynomial time by replacing every satisfiable $(\Sigma \setminus \Delta)$-subtree in $\Phi$ by true, and every unsatisfiable $(\Sigma \setminus \Delta)$-subtree in $\Phi$ by false.*

This is a direct consequence of [7, Theorems 3 and 9]. Unfortunately, projection can destroy determinism (e.g., if the nodes $x$ and $\neg x$ in Figure 3 are removed), making later model counting impossible. Yet, it is easy to see that:

[5] Available at `http://formal.iti.kit.edu/~klebanov/software/`

**Proposition 4.** *Determinism is retained during projection of a d-DNNF formula $\Phi$ on scope $\Delta$, if every subtree rooted at an $\vee$-node associated with a decision on variable $v \in \Sigma \setminus \Delta$ only contains variables from $\Sigma \setminus \Delta$.*

In other words, losing determinism in a subtree is not harmful, if the whole subtree is bound to be removed.

We enforce projection determinism by modifying the variable selection heuristic of the #SAT solvers' d-DNNF compilers to always perform decisions on variables from $\Delta$ first. Further, we implemented the satisfiability check of Proposition 3 by integrating MINISAT into DSHARP. We omitted a similar check in projecting SHARPSAT; the latter can thus report a result that is higher than the actual model count (though this never happened in our benchmarks). When computing min-entropy, such overapproximation entails an error on the conservative side.

### 5.3 Boosting Counting Performance with Formula Preprocessing

In [10], model counting has been improved by a few preprocessing techniques, namely unit propagation, equivalence reduction and hyper binary resolution. In general, any equivalence-preserving preprocessing technique can be applied before model counting, because the set of models does not change. However, there are also many powerful preprocessing techniques that are merely satisfiability-preserving but not equivalence-preserving, such as variable elimination or blocked clause elimination. For general model counting, these techniques cannot be applied. The situation changes when projection is involved.

**Proposition 5.** *Let $\Phi$ be a propositional $\Sigma$-formula and $\Delta \subseteq \Sigma$ a projection scope. Applying satisfiability-preserving preprocessing on $\Sigma \setminus \Delta$ in $\Phi$ does not change the set of models of the projection $\Phi\big|_\Delta$.*

We use the propositional preprocessor COPROCESSOR 2[6] [15, 16], which we developed earlier, for equivalence-preserving simplification and scope-restricted satisfiability-preserving simplification. While the more advanced simplification techniques are not always beneficial, preprocessing boosts model counting performance in most cases, as shown in the next section. The benchmark results are given for default settings. The exact set of applied techniques can be configured by the user.

## 6 Benchmarks and Evaluation

### 6.1 Synthetic Benchmarks

A number of microbenchmarks for general-purpose QIF have appeared in [2] and [21]. The collection has later been consolidated and extended in [18]. It is valuable as it is quite varied and targets different bottlenecks in QIF analyses.

[6] Available at `http://tools.computational-logic.org/`

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

(a) Mix and duplicate

```
if (I == R1) O = R1;
else if (I==R2) O = R2;
...
else if (I==R9) O = R9;
else O = R10;
```

(b) Ten random outputs

```
O = 0;
for (i = 0; i < N; i++) {
  m = 1 << (31-i);
  if (O + m <= I) O += m;
}
```

(c) Binary search

Fig. 4: Benchmarks from [21] and [18]

The only drawback is that the majority of the benchmarks no longer pose a challenge. Below we report results on five benchmarks (out of eleven total) that are still difficult or interesting in some sense.

Table 1 summarizes the performance results. The experiments were performed on a machine with an Intel Core i7 860 2.80GHz CPU. We have included the timings published in [18] for comparison, though no hardware description is available in that paper. The code is presented in Figure 4. Unless noted otherwise, the variable `I` is the secret input, `O` is the observable output, and the type of variables is `uint32_t` (32-bit unsigned integer).

The mix and duplicate benchmark (Figure 4a)—we cite [21]—"combines the two halves of its input word with XOR, and then duplicates these 16 bits in both the upper and lower halves of its output", leaking 16 bit of the secret. "[This leak] is too large to be effectively measured exhaustively, too small for effective sampling, and too uniformly distributed for range queries, so only our probabilistic #SAT strategy gives an accurate estimate" [21]. We can see that neither iterative model enumeration nor modern precise #SAT solvers have difficulties with this benchmark.

The ten-random benchmark (Figure 4b) is essentially a program with ten outputs that do not follow a particular pattern. On this benchmark, the two-bit abstraction from [18] overapproximates the leak.

In the sum benchmark `O = I1 + I2 + I3;`, we increase the difficulty compared to [18], dropping the restriction of the summands to the range 0–10. In-

Table 1: Benchmark runtimes (seconds)
w/PP=with preprocessing, t/o=timeout at 1h, *=result overapproximates number of models. Preprocessing time was negligible in all cases.

| | | Iterative enum. | | Precise #SAT | | Overapprox. #SAT | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | models | sharpCDCL | w/PP | Dsharp | w/PP | sharpSAT | w/PP | [18] |
| mix-n-dup | $2^{16}$ | 16.2 | <0.1 | <0.1 | <0.1 | <0.1 | <0.1 | 1.3 |
| ten-random | 10 | <0.1 | <0.1 | <0.1 | <0.1 | <0.1 | <0.1 | 4.6* |
| sum-three-32 | $2^{32}$ | t/o | t/o | t/o | <0.1 | t/o | <0.1 | n/a |
| bin-search-16 | $2^{16}$ | 6.9 | 9.6 | 166.6 | 39.3 | 12.9 | 5.7 | 6.4 |
| bin-search-32 | $2^{32}$ | t/o | t/o | t/o | 48.2 | t/o | 9.8 | 55.5 |

```
1  int atalk_getname(struct socket *sock, struct sockaddr *uaddr,
2                    int *uaddr_len, int peer)
3  {
4      struct sockaddr_at sat;
5      struct sock *sk = sock->sk;
6      struct atalk_sock *at = at_sk(sk);
7      int err;
8
9      // lock_sock(sk);
10     err = -ENOBUFS;
11     if (sock_flag(sk, SOCK_ZAPPED)) if (atalk_autobind(sk) < 0) goto out;
12
13     *uaddr_len = sizeof(struct sockaddr_at);
14     // memset(&sat.sat_zero, 0, sizeof(sat.sat_zero)); // leak patch
15
16     if (peer) {    err = -ENOTCONN;
17                    if (sk->sk_state != TCP_ESTABLISHED) goto out;
18                    sat.sat_addr.s_net  = at->dest_net;
19                    sat.sat_addr.s_node = at->dest_node;
20                    sat.sat_port        = at->dest_port;
21
22     } else {       sat.sat_addr.s_net  = at->src_net;
23                    sat.sat_addr.s_node = at->src_node;
24                    sat.sat_port        = at->src_port;
25     }
26
27     err = 0;
28     sat.sat_family = AF_APPLETALK;
29     memcpy(uaddr, &sat, sizeof(sat));
30
31 out:
32     // release_sock(sk);
33     unsigned char O; int i;
34     for (i=0; i<sizeof(struct sockaddr_at); i++) O=((char *)uaddr)[i];
35     assert(0);
36     return err;
37 }
```

Fig. 5: AppleTalk driver function leaking kernel memory (CVE 2009-3002)

stead, we consider the sum of three arbitrary 32-bit secret values (variable type `int32_t`). Unsurprisingly, iterative enumeration of models is ineffective, while preprocessing quickly simplifies $\langle p \rangle\big|_O$ to *true*, corresponding to $2^{32}$ outputs.

The binary search benchmark (Figure 4c) is valuable, because it is parametric and can help assess analysis scalability. The program leaks the most significant `N` bit of the secret by repeated dichotomy. We note the improved rates of slowdown between $\mathtt{N} = 16$ and $\mathtt{N} = 32$ with our tools.

### 6.2 Linux Kernel

This benchmark has originally appeared in [12], where the authors analyze a number of vulnerabilities previously found in the Linux kernel. The goal is to measure the amount of unsanitized kernel memory leaking to applications in the userland. The value of the benchmark stems less from the operational significance of the leak size, but rather from its origin in actual systems software. We revisit the most complex example presented in the above paper, a leak in the `atalk_getname` routine in the AppleTalk driver (Figure 5).

The leak is as follows. The kernel allocates a 16-byte structure `sat` (the secret input) and initializes it. Later, the content of the structure is copied to userland. Due to a programming error, parts of the structure are not initialized properly. The official patch fixing the bug is shown in line 14.

In order to deal with an output that is a C struct, we introduce an auxiliary variable `O` and a loop reading the structure before the return statement (lines 33–34). The observable output is then the last `sizeof(struct sockaddr_at)` values of `O`. This is possible since CBMC actually encodes the full trace of variable values rather than merely the initial and the final states.

We have used the code from `net/appletalk/ddp.c` of the Linux kernel 3.4.28 (minus the bugfix). The only simplification that we performed was to remove the locking calls in line 9 and 32, caused by technical difficulties with the code organization of the kernel. It took the analysis in [12] one hour and 39 minutes to find at least 64 blocks in $\approx_p$ of the function—a time explained by an extreme form of self-composition exploring the function behavior 64 times. With SHARP-CDCL, finding 64 blocks was instantaneous, while finding 65536 blocks (i.e., a 16-bit leak) took 20 seconds resp. 14 seconds with preprocessing. The full size of the leak is too large to be established precisely.

### 6.3 Image Anonymization

This benchmark has originally appeared in [17], where the authors assess the effect of several image anonymization techniques on a $125 \times 125$ pixel test image. While effective leakage bounds could be established for blurring or pixelation, no useful bounds (in either direction) could be established for image swirling demonstrated in Figure 1. While, the analysis in [17] sacrifices soundness and precision for scalability (cf. next section), we test our tools by establishing precise leakage in a variant of this application.

As in [17], the source code is derived from the `SwirlImage()` function of the popular ImageMagick[7] image manipulation suite. We deviate from [17] by discounting target image interpolation. This simplification eliminates the influence of image data, and leaves us with a function that merely transforms a pair of integer coordinates into another pair (Figure 6). The coordinates that are not reachable by the swirling transformation appear as black pixels in the illustration in Figure 1b.

Two things should be noted about analyzing code with non-integral data types. First, CBMC approximates data types such as `double` with a 16+16 bit fixed-point representation. We think, it is reasonable to assume that this precision is sufficient in this example. Second, modern general-purpose processors typically implement mathematical functions such as sine, cosine, and square root in hardware. For the analysis, we have used their software counterparts from the popular Freely Distributable C Math Library FDLIBM[8]. Thus, the main function shown in Figure 6 accounts for only 23 out of 379 total lines of

[7] `http://imagemagick.org/`. The `SwirlImage()` function is in `fx.c`.
[8] `http://www.netlib.org/fdlibm/`

```
1  int main(int argc, char **argv) {
2
3      unsigned char x,y; // secret inputs
4      __CPROVER_assume(x>=0 && x<125); // range limit
5      __CPROVER_assume(y>=0 && y<125);
6      unsigned char newx, newy; // observable outputs
7
8      double center = (double) 125/2.0;
9      double radius = center;
10     double degrees = (double) (3.141593*720.0/180.0);
11     double deltay = (double) (y-center);
12     double deltax = (double) (x-center);
13     double distance = deltax*deltax + deltay*deltay;
14
15     if (distance < radius*radius) {
16         double factor=1.0-sqrt((double) distance)/radius;
17         double d = (double) (degrees*factor*factor);
18         double sine=sin(d);
19         double cosine=cos(d);
20
21         newx = ((cosine*deltax-sine*deltay)+center);
22         newy = ((sine*deltax+cosine*deltay)+center);
23     } else { newx = x; newy = y; }
24     assert(0);
25     return 0;
26 }
```

Fig. 6: Image anonymization main function

analyzed code (about 6%). The code contains four loops (maximal unwinding depth 32). A total of 2079 bitvector equations encode the transition relation $\langle p \rangle$.

It took SHARPCDCL 4h58m to find all 12228 blocks in $\approx_p$, which corresponds to a leak of 13.58 out of 13.93 bit, if one measures min-entropy. In other words, swirling is not a good anonymization technique. The deanonymization in Figure 1c actually underestimates the leakage, as the unswirling transformation used is also lossy.

Measuring the residual Shannon entropy of the secret was quite more costly. It took SHARPCDCL 5h23m to find all six inputs in the preimage of a single output (`newx=87`,`newy=62`), and this was only possible as we used—in this case only—MINISAT's randomized variable selection heuristic, which can sometimes produce much faster SAT solver runs at the price of generally unpredictable performance.

Altogether, while we do obtain proof of the secret leaking almost completely, we clearly cannot claim a practical benefit of our analysis in this case. The reasons for this are twofold. First, the analyzed code is too large, and we see this benchmark as marking the frontier of what is barely possible with current technology. Second, the size of the secret is too small, making simple exhaustive simulation an attractive alternative

A new perspective opens if our toolchain were used as part of probabilistic QIF for large secrets (and on programs of more appropriate size). Köpf and Rybalchenko show in [14] that it is sufficient to randomly choose $\frac{(\log_2|S_I|)^2}{(1-P)\delta^2}$ input samples and measure the respective size of the enclosing block in $\approx_p$, in order

to probabilistically estimate the residual Shannon entropy to a degree of precision $\delta$ and a confidence level $P \in [0, 1)$. As the size of the secret increases, the polylogarithmic probabilistic approach remains feasible in contrast to exhaustive simulation.

## 7 Related Work

We survey most recent and relevant works in the field; a further survey of QIF models and techniques is available in [19].

Backes et al. [2] describe a precise QIF analysis for programs with affine indistinguishability relations based on self-composition and Barvinok's counting algorithm. This was later extended in [14] to improve scalability. In order to maintain automation, the latter approach gives up precise computation of the leak and opts for an approximative characterization, deriving lower and upper bounds on residual min-entropy, as well as probabilistic bounds on residual Shannon entropy. A different extension based on symbolic Barvinok counting was proposed by Klebanov in [13].

Heusser and Malacaria have developed two relevant QIF approaches: [12] and [11]. The former encodes detection of (small) leaks as a pure model checking problem via self-composition in CBMC. The latter one and our SAT-based analysis are quite similar in spirit, though [11] builds on expensive self-composition, supplementing it with a model enumerator and a #SAT solver.

The following three approaches compute leakage bounds by approximating the projection $\langle p \rangle\big|_{\langle O \rangle}$ with a series of entailment queries on $\langle p \rangle$, followed by precise or approximative model counting.

Newsome et al. [21] use a series of SMT entailment queries to identify and narrow down in-/feasible output ranges. Such approximations of $\langle p \rangle\big|_{\langle O \rangle}$ are amenable to simple model counting. The approach is complemented by sampling and probabilistic counting. Models generated by the SMT solver are used to identify the presence of a feasible output within a range, but this procedure is not leveraged fully as an output-finding technique.

Phan et al. [22] encode a full binary search for feasible outputs (models of $\langle p \rangle\big|_{\langle O \rangle}$) in a bounded model checker. This approach is precise, but requires in practice more than one call to the underlying solver to find a single feasible output. It is useful when the program verification system does not expose the underlying logical representation or when the used solver cannot generate models.

Meng and Smith [18] use "two-bit-pattern" SMT entailment queries to calculate a propositional overapproximation (w.r.t. the number of models) of $\langle p \rangle\big|_{\langle O \rangle}$ and count its instances with a #SAT solver of the computer algebra system Mathematica.

McCamant and Ernst combine in [17] a dynamic bitwise taint analysis with static analysis to derive bounds on information leakage in C programs. The technique has been applied to large programs used in practice. On the other

hand, it only measures leakage along one or a few selected program paths, leaving it to the user to supply "representative" inputs.

Another tool for dynamic analysis is reported by Chatzikokolakis in [5]. The tool automatically derives bounds of information leakage in terms of mutual information and capacity from trial runs of the system, which is treated as a black box.

The theoretical hardness of QIF has been shown by Terauchi et al. in [27, 24]. As with other hard problems (e.g., SAT), these results do not preclude the existence of efficient analyses for individual instances or subclasses of the problem.

## 8 Conclusion

We presented a unifying abstract formulation of a class of QIF analyses for imperative programs and an instance of this class outperforming previous comparable approaches. We demonstrated that logical projection is a useful framework for understanding and implementing QIF. In the future, we are interested in exploring more advanced projection computation techniques [3, 4, 26].

Though our implementation is not a single tool, all its components are available publicly. A part of the performance improvement is due to advances in the underlying reasoning technology, which have been fueled by regular SAT competitions and associated benchmark collections. Maintaining and extending a set of canonical benchmarks would benefit the QIF field as well.

**Acknowledgments.** This work was in part supported by the German National Science Foundation (DFG) under the priority programme 1496 "Reliably Secure Software Systems – RS3." The authors would like to thank Christoph Wernhard for his comments on projection computation.

## References

1. M. Backes, M. Berg, and B. Köpf. Non-uniform distributions in quantitative information-flow. In *ASIACCS 2011*, pages 367–375. ACM, 2011.
2. M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *S&P 2009*, pages 141–153. IEEE Computer Society, 2009.
3. J. Brauer and A. King. Approximate quantifier elimination for propositional boolean formulae. In *NFM 2011*, pages 73–88. Springer-Verlag, 2011.
4. J. Brauer, A. King, and J. Kriener. Existential quantification as incremental SAT. In *CAV 2011*, pages 191–207. Springer-Verlag, 2011.
5. K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In *TACAS 2010*, pages 390–404. Springer-Verlag, 2010.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 2004*, volume 2988 of *LNCS*, pages 168–176. Springer Berlin / Heidelberg, 2004.
7. A. Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, July 2001.

8. G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA 2006*, pages 109–120. ACM, 2006.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2004*, volume 2919 of *LNCS*, pages 333–336. Springer Berlin / Heidelberg, 2004.
10. Q. Guo, J. Sang, and Y.-m. He. Effective preprocessing in #SAT. In *ICMV 2011*. SPIE, 2011.
11. J. Heusser and P. Malacaria. Applied quantitative information flow and statistical databases. In *FAST 2009*, pages 96–110, Berlin, Heidelberg, 2010. Springer-Verlag.
12. J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ACSAC 2010*, pages 261–269. ACM, 2010.
13. V. Klebanov. Precise quantitative information flow analysis using symbolic model counting. In F. Martinelli and F. Nielson, editors, *Proceedings, International Workshop on Quantitative Aspects in Security Assurance (QASA)*, 2012.
14. B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF 2010*, pages 3–14, Washington, DC, USA, 2010. IEEE Computer Society.
15. N. Manthey. Coprocessor 2.0: a flexible CNF simplifier. In *SAT 2012*, pages 436–441, Berlin, Heidelberg, 2012. Springer-Verlag.
16. N. Manthey, M. J. H. Heule, and A. Biere. Automated reencoding of boolean formulas. In *Proceedings of Haifa Verification Conference 2012*, 2012.
17. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI 2008*, pages 193–205. ACM, 2008.
18. Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *PLAS 2011*, pages 1–12. ACM, 2011.
19. C. Mu. Quantitative information flow for security: a survey. Technical Report TR-08-06, Department of Computer Science, King's College London, 2008. Updated 2010, available at `http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-08-06.pdf`.
20. C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: fast d-DNNF compilation with sharpSAT. In *Proceedings, Canadian AI'12*, pages 356–361, Berlin, Heidelberg, 2012. Springer-Verlag.
21. J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS 2009*, pages 73–85, New York, NY, USA, 2009. ACM.
22. Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu. Symbolic quantitative information flow. In P. Mehlitz, N. Rungta, and W. Visser, editors, *Proceedings, Java Pathfinder Workshop*, pages 1–5, 2012.
23. G. Smith. On the foundations of quantitative information flow. In *FOSSACS 2009*, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag.
24. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS 2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
25. M. Thurley. sharpSAT: counting models with advanced component caching and implicit BCP. In *SAT 2006*, pages 424–429, Berlin, Heidelberg, 2006. Springer-Verlag.
26. C. Wernhard. Tableaux for projection computation and knowledge compilation. In *TABLEAUX 2009*, volume 5607 of *LNAI*, pages 325–340. Springer, 2009.
27. H. Yasuoka and T. Terauchi. Quantitative information flow – verification hardness and possibilities. In *CSF 2010*, pages 15–27. IEEE Computer Society, 2010.