Abstract Model Counting: a novel approach for Quantification of Information Leaks

Quoc-Sang Phan
Queen Mary University of London
q.phan@qmul.ac.uk

Pasquale Malacaria
Queen Mary University of London
p.malacaria@qmul.ac.uk

ABSTRACT

We present a novel method for *Quantitative Information Flow* analysis. We show how the problem of computing information leakage can be viewed as an extension of the *Satisfiability Modulo Theories* (SMT) problem. This view enables us to develop a framework for QIF analysis based on the framework DPLL($\mathcal{T}$) used in SMT solvers. We then show that the methodology of *Symbolic Execution* (SE) also fits our framework. Based on these ideas, we build two QIF analysis tools: the first one employs CBMC, a bounded model checker for ANSI C, and the second one is built on top of Symbolic PathFinder, a Symbolic Executor for Java. We use these tools to quantify leaks in industrial code such as C programs from the Linux kernel, a Java tax program from the European project HATS, and anonymity protocols.

Categories and Subject Descriptors

H.1.1 [**Systems and Information Theory**]: Information theory; D.4.6 [**Security and Protection**]: Information flow controls; D.2.4 [**Software/Program Verification**]: Formal methods, Model checking

Keywords

Quantitative Information Flow; Model Checking; Symbolic Execution; Satisfiability Modulo Theories

1. INTRODUCTION AND BACKGROUND

Quantitative information flow analysis (QIF [14, 23]) is a rigorous approach to *"measure"* information leakage. The motivation for this approach is that absolute security is often not achievable and programs with "*small*" leaks are usually accepted as secure. QIF has attracted considerable attention in recent years and has been applied to the formal analysis of, for example, confidentiality of software [21, 5, 18, 27, 26], loss of anonymity in communication protocols [10], and leakage of information via side-channel [22, 17].

ASIA CCS'14, June 03 - 06, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590328.

To introduce QIF, consider the data sanitization program P from [27, 26], shown in Figure 1. Clearly only integer

```
base = 8;
if (H < 16) then O = base + H;
else O  = base;
```

Figure 1: Data sanitization

values from 8 to 23 are possible outputs of this program. An attacker has hence available 16 possible output observations: observing outputs 9 .. 23 will know the secret H is 1 .. 15 and observing 8 will know the secret is 0 or greater than 15. Assuming the attacker has no prior knowledge of the secret H apart that is a 32 bits variable his *a-priori* probability of guessing the value of H in one try is $\frac{1}{2^{32}}$, and the expected probability of guessing the secret in one try after observing the outputs is:

$$\frac{15}{2^{32}} + \frac{2^{32}-15}{2^{32}}\frac{1}{2^{32}-15} = \frac{15}{2^{32}} + \frac{1}{2^{32}} = \frac{16}{2^{32}}$$

We can measure the leakage of the program as the difference (of the $-\log$ base 2) between the probability of guessing the secret before and after observing the outputs of the program; in this case:

$$-\log(\frac{1}{2^{32}}) - (-\log\frac{16}{2^{32}}) = \log(16) = 4$$

The fact that $\log(16) = \log$ (number of output observations) is not a coincidence. In fact a fundamental QIF result (the channel capacity theorem [24, 33]) shows that leakage for a program is always less or equal to the log of the number of observables of the program. More importantly the theorem holds if we consider not only the above notion of leakage based on the probability of guessing the secret [33] but also if we consider the notion of leakage based on information theory measuring the number of bits leaked [14].

For these reasons counting the number of observables is the basis of state-of-the-art QIF analysis, e.g. [18, 26, 22, 21], and also the basis for this work. The channel capacity theorem also justifies the following:

Definition 1. Given a program P, QIF is the problem of counting N, the number of possible outputs of P.

However the problem of an automated QIF analysis is still very challenging. A simpler problem, called bounding QIF, is considered in [34, 18]: deciding if a program P leaks less than a constant q. In previous work, Yasuoka and Terauchi

have proved that bounding QIF is not a *k-safety* problem for any k [34]. Cerny et al. then proved that in the case of Shannon entropy, bounding QIF is PSPACE-complete [12]. So QIF and bounding QIF remain a huge challenge.

1.1 Overview and Contributions

In order to better frame the contributions of this paper let us consider again the program P in Figure 1. In the computer memory, O is stored as a bit vector $b_1 b_2..b_{32}$ such that the first 27 bits, $\{b_1, .., b_{27}\}$, are 0 (otherwise O will exceed 23) and the last 5 bits range from 01000 to 10111.

Suppose we have a logical formula φ_P that describes the behaviour of P which contains a set of variables $p_1, p_2, ..p_{32}$ representing the bits of the output O. A model of φ_P then provides a truth assignment for the p_i which corresponds to a concrete value of the output O. In this way, the problem of counting all possible outputs, is reduced to the model counting problem on the logical formula φ_P. We name this problem *abstract model counting*, from the traditional *model counting* (#SAT) problem.

In summary, we introduce a theory-based technique which provides a dramatic improvement on state-of-the-art QIF analysis implementations. On the theoretical side, this work establishes a connection between fundamental verification algorithms and QIF. This connection is exploited to mitigate the state explosion problem by developing a novel approach for QIF based on SMT. More specific contributions are:

1. Introduction of a new research problem, #SMT, and its applications to QIF and Symbolic Execution [20].
2. A framework, called #DPLL($\mathcal{T}$), to build a solver for #SMT-based QIF.
3. The methodology of Symbolic Execution re-casted as #DPLL($\mathcal{T}$).
4. Two prototyping tools for QIF analysis: **sqifc** built on top of CBMC [15] and **jpf-qif** built on top of Symbolic PathFinder [32].
5. Analysis of complex code, including new vulnerabilities from the National Vulnerability Database of the US government [2] and anonymity protocols.

The rest of the paper is organized as follows: we describe the #SMT problem and our approach to QIF in Section 2. To illustrate the approach, we provide case studies of real-world applications in Section 3. Related work is mentioned in Section 4. Section 5 concludes our work.

2. ABSTRACT MODEL COUNTING

Before introducing our approach, Symbolic QIF (or SQIF), we recall the SMT problem, and define the new problem #SMT.

Satisfiability Modulo Theories (SMT) is the problem of checking the satisfiability of logical formulas over one or more first-order theories $\mathcal{T}$. *Boolean abstraction* of an SMT formula φ, denoted by $\mathcal{BA}(\varphi)$, is a bijective function that maps Boolean atoms into themselves and theory atoms (or $\mathcal{T}$-atoms) into fresh Boolean atoms. For example, below is a SMT formula φ w.r.t. the theory of *Linear Arithmetic* and its Boolean abstraction:

$$\begin{aligned}\varphi := &\{\neg(x+y>1) \vee A_1\} \\ &\wedge \{(x+y>1) \vee \neg A_2\} \\ &\wedge \{\neg A_3 \vee (y-z<7)\}\end{aligned} \qquad \begin{aligned}\mathcal{BA}(\varphi) := &\{\neg B_1 \vee A_1\} \\ &\wedge \{B_1 \vee \neg A_2\} \\ &\wedge \{\neg A_3 \vee B_2\}\end{aligned}$$

We extend the traditional SMT problem to define a new research problem, namely #SMT:

Definition 2. Given a formula φ w.r.t. combinations of background theories $\mathcal{T}$ and its Boolean abstraction $\mathcal{BA}(\varphi)$, **propositional abstract model counting** or #SMT is the problem of computing the number of models of $\mathcal{BA}(\varphi)$ which are consistent with φ. Such models of $\mathcal{BA}(\varphi)$ are also called *abstract models* of φ.

Note that most SMT theories permit an infinite number of models, but #SMT is always a finite number. Note also that the result of #SMT depends on the syntax of the formula, i.e. in our context the program syntax. For instance, the two formulas φ_1, φ_2 as follows are equivalent but will have different results in #SMT: $\varphi_1 = (x > 0) \vee (x < 0)$ and $\varphi_2 = \neg(x = 0)$.

State-of-the-art SMT solvers are in general the integration of two components: (i) an *enumerator* integrating a SAT solver enumerates truth assignments satisfying the Boolean abstraction of the input formula; (ii) $\mathcal{T}$-solvers validate the consistency w.r.t. theories $\mathcal{T}$ of the (partial) assignment produced by the SAT solver. Naturally, an SMT solver can be extended into a #SMT solver by replacing the SAT solver with a #SAT solver that can explicitly enumerate all models.

2.1 Symbolic QIF as a #SMT problem

The key idea behind SQIF is that instead of checking every concrete value one by one, we process multiple values at a time. To this aim, we need a representation denoting a set of values. We consider a set of atomic propositions $\Phi := \{p_1, p_2, .., p_M\}$, in which each p_i corresponds to the bit b_i of the output O. For example the proposition $p_1 \wedge \neg p_2$ represents a family of sets representing up to 2^{M-2} concrete values: all the bit configurations over M bits where the first bit is 1 and the second bit is 0.

```
for all output bits b_i, 1 ≤ i ≤ M do
    if (b_i == 1) then
        p_i = True
    else
        p_i = False
    end if
end for
```

Figure 2: Symbolic representation

Obviously, without any constraints Φ can represent up to 2^M possible values. With the constraints on the output O imposed by the program P, the number of models of Φ is down from 2^M to a number N that we need to count. To generate the formula of these constraints: the program P, to which the code in Figure 2 has been appended at the end, is first transformed into a logical formula φ_P w.r.t. theories $\mathcal{T}$, e.g. by translating statements into Static Single Assignment (SSA) form. Once P is encoded as a logical formula φ_P, then we can apply model checking on P to verify the satisfiability of φ_P. In other words, a model checker like CBMC is used as a $\mathcal{T}$-solver.

$$\text{Program} \longleftrightarrow \text{Logical formula}$$
$$\text{Model checker} \longleftrightarrow \mathcal{T}\text{-solver}$$

If one accepts the view that each p_i is a Boolean abstraction of the $\mathcal{T}$-atom expressing the constraints on bit b_i, then the QIF problem of counting N can be viewed as a #SMT problem. In other words, with this view the QIF analysis (Definition 1) is a #SMT problem.

The transformation from programs to logical formulas is also exactly what bounded model checkers like CBMC do: once a program is converted into a logical formula then its satisfiability is checked with a SAT solver or an SMT solver. We however can also use non SAT-based model checkers, e.g. Java PathFinder (JPF) [1], as a $\mathcal{T}$-solver. Our approach does not depend on a programming language or the type of model checker.

At this point we have defined the notation of a symbolic representation Φ of the state space of the output O. In the next section, we will describe a DPLL-based framework to systematically explore Φ.

2.2 A #DPLL($\mathcal{T}$) for QIF

The first practical approach for #SAT is an extension of DPLL [11] that enumerates all models. Modern #SAT solvers, e.g. RelSat [9] and c2d [16], are more efficient thanks to smarter approaches that exploit the structure of the clauses and avoid explicitly enumerating models. In the context of #SMT, however, explicit enumeration of abstract models is necessary since we need to check the consistency of each abstract model w.r.t. the background theory $\mathcal{T}$. The Symbolic QIF framework we propose here is a #DPLL($\mathcal{T}$) tailored for QIF, by combining a simple DPLL-based #SAT solver and a $\mathcal{T}$-solver. Hence, this variant of #DPLL($\mathcal{T}$) is a #SMT solver in the context of QIF.

A high level framework to explore the state-space and quantify the leaks of confidential data is described by the procedures `SymbolicQIF` and `SymCount` in Figure 3 and 4.

function SymbolicQIF(Φ, φ_P)
  $\Psi = \epsilon$, $pc = \epsilon$, $N = 0$, $i = 1$
  EarlyPrunning(Φ)
  SymCount($\Phi, \Psi, \varphi_P, N, pc, i$)
return Ψ, $\log_2(N)$
end function

Figure 3: Symbolic QIF analysis

Φ, Ψ, φ_P and N are passed by reference, while pc and i are passed by value. Φ is the symbolic representation of the output described in the previous section, φ_P is the formula representing the program P and Ψ is the set of models of φ_P. N is the cardinality of Ψ, and the procedure `SymbolicQIF` returns $\log_2(N)$ as the channel capacity. M is the size of the output data type, e.g. $M = 32$ if O is a 32-bit integer, and i is the depth of the recursive call. The parameter pc is a partial assignment of Φ, it is incrementally updated when the search progresses. In `SymCount`, $\mathcal{T}$-solver(φ_P, pc) means the $\mathcal{T}$-solver is called to check if there is a model of φ_P where pc is (assigned to) *True*.

We illustrate the algorithm `SymCount` by running it on a simple example (we ignore temporarily lines 2 and 3 that will be clarified in section 2.4). Consider again the case study

1: **function** SymCount($\Phi, \Psi, \varphi_P, N, pc, i$)
2: **if** ($N \geq 2^k$) **then return** *Insecure*
3: **end if**
4: Extract p_i from Φ
5: $pc_1 \leftarrow pc \wedge p_i$
6: **if** ($\mathcal{T}$-solver(φ_P, pc_1)) **then**
7: **if** ($i == M$) **then**
8: $\Psi \leftarrow \Psi \cup \{pc_1\}$
9: $N \leftarrow N + 1$
10: **else**
11: SymCount($\Phi, \Psi, \varphi_P, N, pc_1, i + 1$)
12: **end if**
13: **end if**
14: $pc_2 \leftarrow pc \wedge \neg p_i$
15: **if** ($\mathcal{T}$-solver(φ_P, pc_2)) **then**
16: **if** ($i == M$) **then**
17: $\Psi \leftarrow \Psi \cup \{pc_2\}$
18: $N \leftarrow N + 1$
19: **else**
20: SymCount($\Phi, \Psi, \varphi_P, N, pc_2, i + 1$)
21: **end if**
22: **end if**
23: **end function**

Figure 4: Symbolic counting for QIF

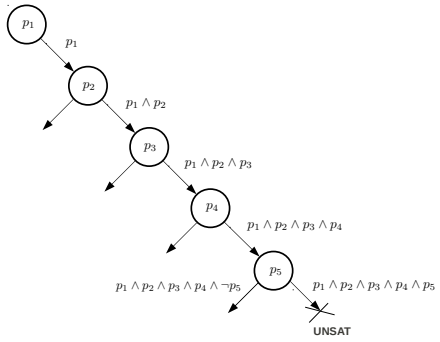

Figure 5: Partial exploration path of SQIF for the data sanitisation program from Figure 1.

of the data sanitization program in Figure 1. Only integer values from 8 to 23 are possible outputs of this program, which means the number of possible outputs is $N = 16$. At the beginning, all variables are initialised as in Figure 3, the method `EarlyPrunning` employs a heuristic that will be discussed later in this section. The method `SymCount` is then called to count the number of possible models of φ_P. When a variable $p_i \in \Phi$ is selected, we systematically explore in the same way for both p_i and $\neg p_i$. Hence, the block of code from line 5 to line 13, and the one from line 14 to line 22 in Figure 4 are symmetric: we only explain the first one.

A partial run of `SymCount` on the illustrative example is depicted in Figure 5. At the first call of `SymCount`: $i = 1$, the variable p_1 is in consideration and it is added to pc in line 5. Since pc is initialised to be empty, $pc_1 = p_1$. The $\mathcal{T}$-solver is called to check if there is a model of φ_P where p_1 is (assigned to) *True*. This can be done by using assertion to check the validity of $\neg p_1$ in a program as follows:

assert $!p_1$;

A model checking tool like JPF or CBMC can be used as a $\mathcal{T}$-solver to verify this assertion and it will return $True$ if the assertion fails, and $False$ otherwise. In this example, the $\mathcal{T}$-solver would return $True$ since p_1 stands for "first bit is 1" and all odd values from 9 to 23 are possible outputs satisfying the condition p_1. Hence, SQIF proceeds by calling `SymCount` with $i = 2$. Similarly, the procedure progresses until calling `SymCount` with $i = 5$, which means it needs to verify:

$$\textbf{assert } !(p_1 \;\&\&\; p_2 \;\&\&\; p_3 \;\&\&\; p_4 \;\&\&\; p_5);$$

This time the $\mathcal{T}$-solver would return $False$, since $p_1 \wedge p_2 .. \wedge p_5$ represents a set of outputs of which each element is at least $2^0 + 2^1 + .. + 2^4 = 31$, while the possible range of O is only from 8 to 23. For a program with an output of 32-bits, by using `EarlyPrunning`, SQIF trims a set of 2^{27} concrete values represented by the family of sets:

$$\{\Phi := \{p_1, p_2, .., p_{32}\} : p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5\}$$

This is how the state-space explosion problem is mitigated.

At the depth $i = 5$ as above, if SQIF takes the path of $\neg p_5$ from line 14, then the $\mathcal{T}$-solver returns $True$ ($O = 15$ is one of the models). Hence, the procedure continues with $i = 6$, and from this point until $i = 32$, only the path of $\neg p_i$ is SAT. At $i = 32$, SQIF finds a full path 00..01111 which represents an output $O = 15$. This path is added to Ψ, and SQIF increases N. Finally, at the end of the method `SymbolicQIF`, we have $\Psi = \{8, 9, .., 23\}$ and $N = 16$, thus we can conclude that the data sanitization program in Figure 1 leaks at most 4 bits.

The method `EarlyPrunning` implements the idea that if p_1 is unsatisfiable, then $p_1 \wedge C$ is also unsatisfiable for any C. Therefore, at the beginning of the `SymbolicQIF`, all p_i are checked for satisfiability, and the results are stored for later use. We note that `EarlyPrunning` speeds up `SymbolicQIF` dramatically when the number of possible models (outputs of the program) is small.

We have developed a prototyping tool for QIF analysis of C programs, **sqifc** built on top of CBMC.

2.3 Symbolic Execution as #DPLL($\mathcal{T}$)

Symbolic Execution (SE) [20] is a powerful technique, widely used in many domains of application such as test data generation, partial verification, symbolic debugging, and program reduction. Here we argue that:

Suppose a program is encoded into a logical formula then a Symbolic Executor can be viewed as a #SMT solver for this formula.

Intuitively, if we see a program as a logical formula, then a concrete execution of the program corresponds to a model of that formula. A symbolic path, which represents a set of concrete executions, can be viewed as corresponding to the Boolean abstraction of the set of models. Thus, a Symbolic Executor, returning all symbolic paths of a program, can be viewed as a #SMT solver.

$$\text{Program} \longleftrightarrow \text{Logical formula}$$
$$\text{Concrete execution} \longleftrightarrow \text{Model of the formula}$$
$$\text{Symbolic path} \longleftrightarrow \text{Boolean abstraction of models}$$

Example 1. To illustrate the idea, consider a simple C program as follows:

```
if (x > 1) { y = x < 5 ? x + 10 : x;} else y = 0;
```

We denote the propositional variables: C_1 as $(x > 1)$, C_2 as $(x < 5)$, A_1 as $(y_1 = x + 10)$, A_2 as $(y_2 = x)$, and A_3 as $(y_3 = 0)$. In this way, the program can be viewed as a Static Single Assignment-like (SSA) logical formula:

$$(C_1 \rightarrow ((C_2 \rightarrow A_1) \wedge (\neg C_2 \rightarrow A_2))) \wedge (\neg C_1 \rightarrow A_3)$$

This is a formula modulo the theory of *linear arithmetic*. In the Boolean abstraction of the formula, A_1, A_2 and A_3 are pure literals and can be removed. The formula has hence three models that can be written as:

$$\{C_1 \wedge C_2, C_1 \wedge \neg C_2, \neg C_1\}$$

This is also the set of possible symbolic paths generated by running SE on the program.

```
 1: function SymEx(P, σ, pc, l)
 2:     Execute assignment statements, update σ
 3:     if (l = EOF) then
 4:         Ψ ← Ψ ∪ {pc}
 5:         Σ ← Σ ∪ {σ}
 6:         return
 7:     end if
 8:     Extract {c, l⊤, l⊥} from if-statement
 9:     if (T-solver(pc ⊢ c)) then
10:         SymEx(P, σ, pc, l⊤)
11:     else if (T-solver(pc ⊢ ¬c)) then
12:         SymEx(P, σ, pc, l⊥)
13:     else
14:         pc1 ← pc ∧ c
15:         if (T-solver(pc1)) then
16:             SymEx(P, σ, pc1, l⊤)
17:         end if
18:         pc2 ← pc ∧ ¬c
19:         if (T-solver(pc2)) then
20:             SymEx(P, σ, pc2, l⊥)
21:         end if
22:     end if
23: end function
```

Figure 6: Symbolic Execution

The above example suggests we can see SE as #SMT solver, we will now derive SE in the #DPLL($\mathcal{T}$) framework. Let us consider a program in SSA form and consisting of only assignment statements and conditionals. Tools such as CBMC and Soot [4] transform ANSI C and Java programs into SSA to verify properties.

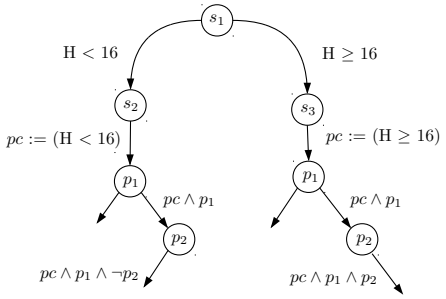

Figure 7: Partial exploration path of SQIF-SE for the data sanitisation program from Figure 1.

The algorithm of SE is depicted in Figure 6. Here σ is a symbolic environment, i.e. a map from program variables to formulas over symbolic inputs. The parameter pc holds the current path condition. At the end of the procedure, Ψ will be the set of all path conditions and Σ will be the set of all final symbolic environments, one for each path condition. Statements in the program P are identified by their program location l. For a conditional statement `if (c)` I `else` I', symbols $l_\top$ and $l_\bot$ denote the locations of the first statements in I and I' respectively. Notice that in `SymEx` lines 9 to 12 can be viewed as $\mathcal{T}$-propagation, and lines 14 to 21 can be viewed as splitting in DPLL($\mathcal{T}$) [28].

In `SymEx` $\mathcal{T}$-solver($pc \vdash c$) calls the solver to check whether c is a consequence of pc and $\mathcal{T}$-solver(pc) calls the solver to check whether pc is satisfiable.

We believe that the insight of SE as #DPLL($\mathcal{T}$) paves the way for the development of efficient Symbolic Executors, by exploiting techniques that have been successful in SMT such as non-chronological backtracking and clause learning [31].

2.3.1 SQIF by Symbolic Execution:

With the view of SE as #DPLL($\mathcal{T}$), we are able to make a Symbolic Executor work as `SymbolicQIF` with little effort. The key idea here is to enumerate all concrete values from symbolic executions.

For a program P that takes symbolic inputs $i_1, i_2, ..i_\alpha$, and produces an output O, the result of running SE on P is as follows:

$$O = \left\{ \begin{array}{ll} f_1(i_1, i_2.., i_\alpha) & \text{if } pc_1 \\ f_2(i_1, i_2.., i_\alpha) & \text{if } pc_2 \\ \dots & \dots \\ f_\beta(i_1, i_2.., i_\alpha) & \text{if } pc_\beta \end{array} \right\}$$

where $f_1, f_2, ..f_\beta$ are formulas over symbolic inputs $i_1, i_2, ..i_\alpha$. $pc_1, pc_2, ..pc_\beta$ are the path conditions. Notice f_i expresses a symbolic final value for O, i.e. in terms of `SymEx` instead of $f_i(i_1, i_2.., i_\alpha)$ we could write $\sigma_i(O)$ for $\sigma_i \in \Sigma$. The following proposition was proved by King [20]:

Proposition 1.

$$\forall i, j \in [1, \beta] \wedge i \neq j,\ pc_i \wedge pc_j = \bot$$

which means that path conditions are mutually exclusive.

Definition 3. For a path condition pc_i obtained from SE, the concretization set of pc_i, denoted $\mathcal{CS}(pc_i)$, is the set of all concrete values of output O that can be reached by executing the program following pc_i.

Consider again the *Example 1* in which there are three path conditions: $pc_1 = C_1 \wedge C_2$, $pc_2 = C_1 \wedge \neg C_2$, $pc_3 = \neg C_1$. The corresponding concretization sets of these path conditions are: $\mathcal{CS}(pc_1) = [12..14]$, $\mathcal{CS}(pc_2) = [5..2^M]$, $\mathcal{CS}(pc_3) = [0]$, where M is the number of bits of the variable x. The set of all possible values of output O is formed by the union of concretization sets of all paths, and thus:

$$N = \left| \bigcup_{i=1}^{\beta} \mathcal{CS}(pc_i) \right|$$

The set $\mathcal{CS}(pc_i)$ can be computed by inserting the code in Figure 2 at the end of the program and run SE: we add M conditions, each one tests whether bit b_i of the output O is 0 or 1. These M conditions test all the bits of the output O. Exploring all possible combinations of these conditions leads to enumerating all possible values of O. We denote by SQIF-SE the implementation of SQIF using SE. A partial exploration path of SQIF-SE is described as in Figure 7. SE as implemented by Symbolic PathFinder (SPF) returns a concrete values for each possible path. The number of distinct concrete values is the N that we need to count.

SQIF-SE is implemented into a prototyping tool **jpf-qif** built on top of SPF. The tool works on Java programs.

2.4 Soundness and Completeness

By soundness of the SQIF approach we mean that given Ψ, $\log_2(N)$ returned by `SymbolicQIF`(Φ, φ_P), each element of Ψ is a model of φ_P i.e. corresponds to a possible value of the output of the program P. By completeness of SQIF, we mean that Ψ is the set of all models of φ_P i.e. all values of the output of P.

Theorem 1. *Given a sound (resp. complete) $\mathcal{T}$-solver the SQIF approach is sound (resp. complete) i.e.* `SymCount` *solves the QIF problem (Definition 1).*

Proof sketch 1. *The SQIF algorithm as described in Figure 4 is based on DPLL which itself is a depth-first search procedure. As the search space is a binary tree with bounded depth M, the number of bits of the output, the depth-first search procedure is complete. The soundness of SQIF is guaranteed by the soundness of the $\mathcal{T}$-solver, i.e. model checker.*

In reality $\mathcal{T}$-solvers are only complete in particular domains. Moreover, even with sound and complete $\mathcal{T}$-solvers, a large leak requires an exponential number of calls to the $\mathcal{T}$-solver and so in practice SQIF is complete only for programs with *small* leaks. SQIF-SE relies on a Symbolic Executor, and hence it is complete in programs with a bounded model of runtime behaviour, which means programs have no recursion or unbounded loops. These are well-known issues in SE and handling them is orthogonal to our work. Since our tools are based on bounded model checker and bounded SE, we choose to analyse only bounded programs. Notice however that Theorem 1 holds for general $\mathcal{T}$-solvers.

Because of these practical issues about completeness, it has been proposed to shift the focus from the question *"How much does it leak?"* to the simpler quantitative question *"Does it leak more than k?"* [18, 34]. This approach not only makes the problem easier to analysis, but it is also more intuitive in term of security, because the user policy, i.e. *threshold k*, is encoded in the analysis. The ultimate goal of security analysis is to determine whether a program is *secure* or *insecure*. As discussed in the previous section, the goal of QIF is to relax security policy from *non-interference* to an acceptable threshold *k bits of interference*, so that we can tolerate "*small*" leak, and accept more programs as secure. The SQIF approach can also be used in the same way: with a user policy k, if SQIF finds out more than $\mathcal{K} = 2^k$ possible outputs, we can stop the procedure and conclude that the program is *insecure*. This is the meaning of lines 2 and 3 of function `SymCount` in Figure 4.

A straightforward consequence of the Theorem 1 is that, assuming a sound $\mathcal{T}$-solver, given a user policy k, `SymCount` never returns *secure* for a program leaking more than k bits. This can be formally expressed as:

Corollary 1. *SQIF is sound w.r.t a user policy k.*

3. CASE STUDIES

Only few papers present QIF static code analysis of real-world applications: examples are [18], [22] and the more recent [21]. Of these three approaches, [22] uses a different attacker model, namely cache side-channels and so is not directly comparable with our approach. The other two, [18] and [21], use the same attacker model as we do but are at the moment both restricted to C programs, and hence only comparable to **sqifc**. We will concentrate on [18], to which we refer as **selfcomp**, because it is based on the well-known concept of self-composition [7]. **sqifc** is compared to [21] in section 3.7. For the analysis of anonymity protocols, we compare **sqifc** against **QUAIL** [3, 10], a state-of-the-art quantitative analyser for probabilistic programs. The case studies broadly fall in three categories:

- the first category, consisting of case studies from the National Vulnerability Database of the US government [2], is aimed to demonstrate how our analysis is able to deal with complex C-code,
- the case studies CRC and Tax show the applicability to quantify leakage in applications which leak by design,
- the case studies Grade and Dining cryptos protocols show how our technique, even if it is unable to analyse probabilistic programs, is able to computing channel capacity for anonymity protocols.

The experiments are conducted on a desktop machine with Intel Core i5 3.3GHz and 8GB of memory.

3.1 CVE-2011-2208

This case study is an example of a program that leaks information when the attacker can control the public input. It is taken from the National Vulnerability Database (NVD) of the US government [2], and it is released on 13/06/2012. The system call `osf_getdomainname`, depicted in Figure 8, in

```
1  int osf_getdomainname(char __user *name, int namelen)
2  {
3    unsigned len;
4    int i, error;
5
6    error = verify_area(VERIFY_WRITE, name, namelen);
7    if (error)
8      goto out;
9
10   len = namelen;
11   if (namelen > 32)
12     len = 32;
13
14   down_read(&uts_sem);
15   for (i = 0; i < len; ++i) {
16     __put_user(system_utsname.domainname[i],
17         name + i);
18     if (system_utsname.domainname[i] == '\0')
19       break;
20   }
21   up_read(&uts_sem);
22  out:
23    return error;
24 }
```

Figure 8: arch/alpha/kernel/osf_sys.c

the Linux kernel before 2.6.39.4 leaks sensitive information from kernel memory. This is caused by an integer signedness error: the signed parameter `namelen` is assigned to the unsigned variable `len` in line 10, so a negative value can be transformed into a big positive one. Therefore, although the condition in line 11 restricts `namelen` to 32, the number of characters returned to the user via the structure `name` may be much greater including bytes from kernel memory.

In order to quantify the information leakage caused by this vulnerability, we chose the thresholds of security policy $\mathcal{K} = 64$ and $\mathcal{K} = 256$, which means the program is secure if it leaks less than 6 and 8 bits respectively. After the times in Figure 11 , **sqifc** and **selfcomp** conclude that the program is insecure. We then apply the patch provided for this vulnerability, and run **sqifc** again. This time, **sqifc** found only one possible value for `name`, which means a leak of zero bit. Hence, we prove that the patch fixed the leak.

3.2 CVE-2011-1078

This case study is also taken from NVD, and it is released on 21/06/2012. The function `sco_sock_getsockopt_old` in the Linux kernel before 2.6.39, depicted in Figure 9, leaks sensitive information from kernel memory. As in line 24,

```
1  static int sco_sock_getsockopt_old(
2      struct socket *sock, int optname,
3      char __user *optval, int __user *optlen)
4  {
5    struct sock *sk = sock->sk;
6    struct sco_conninfo cinfo;
7    int len, err = 0;
8    ...
9
10   lock_sock(sk);
11
12   switch (optname) {
13     case SCO_OPTIONS:
14       ...
15
16     case SCO_CONNINFO:
17       ...
18
19       cinfo.hci_handle = sco_pi(sk)->conn->hcon->handle;
20       memcpy(cinfo.dev_class,
21         sco_pi(sk)->conn->hcon->dev_class, 3);
22
23       len = min_t(unsigned int, len, sizeof(cinfo));
24       if (copy_to_user(optval, (char *)&cinfo, len))
25         err = -EFAULT;
26         break;
27         ...
28       }
29
30       release_sock(sk);
31       return err;
32 }
```

Figure 9: net/bluetooth/sco.c

`cinfo` is copied to the user. Although its total size is 5 bytes, and all bytes are correctly assigned, when compiled it includes an additional padding byte for alignment purposes. This padding byte is not zeroed out, and hence it contains kernel memory, and is leaked to the user. Results of the analysis for $\mathcal{K} = 8$ and $\mathcal{K} = 64$, are shown in Figure 11.

3.3 Cyclic Redundancy Check

The program in Figure 10 performs Cyclic Redundancy Check[1] (CRC) and shifts right the result `sft` bits. We also have a Java version of the program to test with **jpf-qif**. We quantify the amount of information of the confidential input `ch` revealed by observing the output of function `GetCRC8`. We analyse this program with **sqifc**, **jpf-qif** and **selfcomp** for `sft` values of 3 and 5 giving a maximum leakage for this program of 5 (**selfcomp** times out on this case) and 3 bits respectively which is consistent with the design of the program. Results of the analysis for are shown in Figure 11. In the case value of `sft` is 5, i.e. $\mathcal{K} = 8$, **selfcomp** is faster as the state-space is still small enough, and **selfcomp** requires only one call to CBMC. When `sft` $= 3$, i.e. $\mathcal{K} = 32$, the state-space explosion makes **selfcomp** fail to solve. SQIF

[1] http://en.wikipedia.org/wiki/Cyclic_redundancy_check

```
unsigned char GetCRC8(
        unsigned char check , unsigned char ch)
{
  int i, sft ;
  for ( i = 0 ; i < 8 ; i++ ) {
    if ( check & 0x80 ) {
      check<<=1;
      if ( ch & 0x80 ) { check = check | 0x01;}
      else { check =check & 0xfe; }
      check = check ^ 0x85;
    } else {
      check <<=1;
      if ( ch & 0x80 ) { check = check | 0x01; }
      else { check = check & 0xfe; }
    }
    ch<<=1;
  }
  check >>= sft;
  return check;
}
```

Figure 10: Cyclic Redundancy Check

requires several calls to the solver, but it is less vulnerable to state-space explosion.

3.4 Tax Record

Balliu et al. [6] provided a very interesting case study of information flow security in Java programs derived from the EU-funded FP7-project HATS. The program contains 8 classes/interfaces and 267 LoC. In our analysis we assume the year 2011-2012 basic tax rate in UK, which is applied for a person whose income does not exceed 35 thousand pounds per annum, is $F = 20\%$[2]. Thus, the tax is less than 7 thousands pound per annual. We assume that donations to charities is below the amount of tax to be paid. Obviously, one cannot pay more than what one earns. Following [6] we are interested in leaks of a taxpayer's income and donations to a Tax checker.

3.4.1 QIF vs. Declassification

Balliu et al. considered two cases of declassification: the first one, called taxChecker1, is associated with the policy "$income \times F\% + donation > payment$", and the second one, called taxChecker2, is associated with the policy "$income \times F\% + donation - payment$". They claimed that: *"The value declassified in the taxChecker1 case, resp. taxChecker2 case, is a lower bound, resp. upper bound, of the value revealed to the tax checker in the fixed tax rate variant."*

We notice in this sentence the use of terms like *"value revealed"* and *"bound"*: the authors were trying to describe *quantitative* concepts. From the result of **jpf-qif**, we can give hence quantitative answers to these questions. In the case of *taxChecker1*, the observable is whether the payment is greater or smaller than the sum of the tax and donations, which means there are 2 possible outputs. This corresponds to the leak of 1 bit. In other words, the user policy or threshold $k = 1$. Regarding the taxChecker2 case, under the assumptions listed above, the leakage is upper bounded by 4.86 bits obtained in 24.988 seconds.

3.5 The Grade Protocol

This case study was used to illustrate protocol analysis in [19, 10]. This anonymity protocol is designed to enable a group of students to compute the sum of their grades (e.g., to compute the average) without revealing individual grades.

[2]Since SPF can only handle conditions with integer values, we simplify the code by replacing $(income \times 20)/100$ with tax as an integer. The simplification we made does not change the secrecy, i.e. entropy, of $income$, so it will not affect the result of our analysis.

We denote $S_1, ..., S_k$ be the k students arranged in a ring, each one is given a secret grade g_i between 0 and $m-1$. To compute the sum of g_i without disclosing them, the students produce k random numbers between 0 and $n = (m-1)k+1$ such that the number r_i is known only to the students S_i and $S_{(i+1)\%k}$. Each student s_i then outputs a number $d_i = g_i + r_i - r_{(i+1)\%k}$ and the sum of all grades is equivalent to the sum of the outputs modulo n.

```
1  int func(){
2    size_t S = 5, G = 5, i = 0, j = 0;
3    size_t n = ((G-1)*S)+1, sum = 0;
4    size_t numbers[S], announcements[S], h[S];
5
6    for (i = 0; i < S; i++) h[i] = nondet_int() % G;
7
8    for (i = 0; i < S; i++)
9          numbers[i] = nondet_int() % n;
10
11   while (i<S) {
12     j=0;
13     while (j<G) {
14       if (h[i]==j)
15         announcements[i] =
16           (j+numbers[i]-numbers[(i+1)%S])%n;
17       j=j+1;
18     }
19     i=i+1;
20   }
21
22   for (i = 0; i < S; i++)
23     sum += announcements[i];
24
25   return sum % n;
26 }
```

Figure 12: The Grade protocol

This protocol is implemented as a probabilistic program in both [19] and [10]. Here we implement it in standard ANSI C with the built-in non-deterministic functions of CBMC. The source code, shown on Figure 12, is based on the one provided in [10]. The array `h[S]` stores the grades of all students, i.e. the secret. The attacker can observe `sum % n`.

To compare **QUAIL** with our tool, **sqifc**, we repeat the experiment of the authors for the grade protocol with the tool and examples provided in [3]. However, **QUAIL** timed out after 1 hours for most of the cases, as showed in Figure 15 (we had the same results of leakage with the authors in the cases the tool did not time out). Therefore, we take the result in Figure 13 directly from the paper [10]. Comparing Figure 13 and Figure 14, it is easy to realise that the bounds on the leaks, measured by **sqifc**, do not exceed the real leaks, measured by **QUAIL**, by more than 1 bit, while **sqifc** required no more than 1 minutes in all cases as showed in Figure 16.

3.6 The Dining cryptos protocol

This case study is a variation of the dining cryptographers protocol of Chaum [13], one of the most popular problem in anonymity protocol. There is a group of cryptographers gathering around a table for dinner. After the meal, they are informed that the bill has been paid by someone, who could be one of them or the National Security Agency (NSA). Even though the cryptographers respect each other's right to make an anonymous payment, they want to find out whether the NSA paid. To determine this, they use a protocol as follows: each pair of adjacent cryptographers toss a coin hidden from everybody else, so that each cryptographer only knows the values of the coin shared with the one on his left and with the one on his right; then each cryptographer declares aloud the exclusive OR of the two coins he sees, i.e. 0 if they have the same value and 1 otherwise. However if the payer is one

Case Study	LoC	Language	**sqifc** time	**jpf-qif** time	**selfcomp** time
Data Sanitization	< 10	C/Java	11.898	20.695	timed out
CVE-2011-2208 (64)	> 200	C	22.759	-	119.117
CVE-2011-2208 (256)		C	88.196	-	timed out
CVE-2011-1078 (8)	> 200	C	10.380	-	13.853
CVE-2011-1078 (64)		C	37.899	-	timed out
CRC (8)	< 30	C/Java	1.209	8.386	0.498
CRC (32)		C/Java	8.657	9.357	timed out
Tax Record	267	Java	-	24.988	-

Figure 11: Times in seconds, timeout is 30 minutes. "-" means inapplicable.

(a)

Grades \ Students	**2**	**3**	**4**	**5**
2	1.500	1.811	2.030	2.198
3	2.197	2.525	2.745	2.910
4	2.655	2.984	3.201	3.365
5	2.999	3.325	3.541	timed out

Figure 13: Leakage measured by QUAIL

(b)

Grades \ Students	**2**	**3**	**4**	**5**
2	1.585	2.000	2.322	2.585
3	2.322	2.807	3.170	3.459
4	2.807	3.322	3.700	4.000
5	3.170	3.700	4.087	4.392

Figure 14: Leakage measured by sqifc

```
1  size_t func(){
2
3    size_t N = 5, output = 0, i = 0;
4    size_t coin[N], obscoin[2], decl[N];
5    size_t h;
6
7    h = nondet_uchar() % (N+1);
8
9    for (i = 0; i < N; i++){
10     coin[i] = nondet_uchar() % 2;
11   }
12
13   for (i = 0; i < N; i++){
14     decl[i] = coin[i] ^ coin[(i+1)%N];
15     if (h==i+1){
16       decl[i] = !decl[i];
17     }
18     i = i+1;
19   }
20
21   for (i = 0; i < N; i++){
22     output = output + decl[c];
23   }
24
25   return output;
26 }
```

Figure 18: The Dining cryptos protocol

of the cryptographers, he declares the opposite. In the end, if the sum of all declared values is even, then it is concluded that the NSA paid the bill. On the other hand, the sum is odd means one of the cryptographers did it.

We are interested in knowing how much information about the payer can be leaked by the sum of all declared values (in the dining cryptographers the observation are the declared values instead). The input code for the protocol is depicted in Figure 18. `h` is the identity of the payer, i.e. the secret, `output` is the observable. The coin toss is modelled by a built-in non-deterministic function in line 10. This model is less precise than implementation in probabilistic programs where it is possible to select random values from a specific distribution. By modelling with non-deterministic function and computing channel capacity, we can only compute the maximum leakage in all possible distributions. The channel capacity computed by **sqifc** is showed in Figure 17.

3.7 Optimizing SymCount

One source of inefficiency in the implementation of **sqifc** is that: in each call to CBMC, the transformation from the source code to the logical formula φ_P is recomputed. This can be very costly when the program is large or many calls to CBMC are needed. A simple optimisation to tackle this problem is to use CBMC to compute φ_P once and for all, and then use a SAT or SMT solver to check φ_P together with the appropriate assertion at line 6 or 15 of `SymCount`. We denote the resulting implementation **sqifc+**: it analyses CRC(8) in 0.289 and CRC(32) in 0.475 seconds respectively: an average improvement well over 1000% over **sqifc**.

We believe the performance of **sqifc+** is comparable to the technique in [21] for programs with small leaks but it is outperformed when analysing programs with large leaks (a precise comparison is not possible as all but one programs in [21] have large leaks), however we don't see this as a big issue: the main point of QIF is to determine whether a program leaks a small amount and hence can be considered a secure program, and while the meaning of "small leak" is context dependent it is difficult to see contexts where leaks much larger than 10 bits can be considered small.

4. RELATED WORK

Meng and Smith introduce an *approximate* technique to calculate an upper bound on channel capacity in [26]. The authors' implementation of the method is largely manual, and we proposed an automation for it in [30]. While the work of Meng and Smith is very inspiring, the technique can be very imprecise, for example when the leaks are sparse in the state space. Moreover, the user policy is not encoded in the analysis which makes it infeasible when the leaks are not small. Take an example of a program that leaks all 32 bits of integral confidential data, it needs to make 64 calls to STP solver to determine that all bits are *Non-fixed*. Then, in order to determine two bit patterns of $(31*32)/2 = 496$ pairs of *Non-fixed* bits, it needs to make another $496 * 4 = 1984$ calls to STP solver, so it is 2048 calls in total.

The first automated method for QIF was proposed by Backes et al. [5]. The method can be divided into two stages: first, it employs model checking to compute an equivalence relation $\mathcal{R}$ on the set of confidential inputs w.r.t. observable outputs; secondly, if this relation $\mathcal{R}$ can be represented by a system of *linear integer inequalities* $A\bar{x} \geqslant \bar{b}$, which means it is a bounded integer *polytopes*, then a variant of Barvinok's algorithm [8] can be used to count the number of integer

(c)

Grades	Students 2	3	4	5
2	1.306	241.483	-	-
3	28.613	-	-	-
4	508.313	-	-	-
5	-	-	-	-

Figure 15: Elapsed time in seconds of QUAIL

(d)

Grades	Students 2	3	4	5
2	5.657	7.029	10.767	9.469
3	9.145	11.597	17.987	20.930
4	10.095	16.872	21.869	18.579
5	14.639	20.666	33.298	40.399

Figure 16: Elapsed time in seconds of sqifc

cryptos	3	4	5	6	50	100	200	300
Channel capacity	2	2.32	2.59	2.81	5.672	6.658	7.651	8.234
Time in seconds	2.145	3.496	3.632	18.634	46.970	158.517	587.670	3326.915

Figure 17: The dining cryptos protocol analysed by sqifc

solutions of $\mathcal{R}$. While this work is important as the first effort on automation of QIF analysis, it is not clear however how this approach can be applied to real-world programs because of, for example, bit-wise operators in the CRC case study or non-linear relations and so on.

Closer to our work is the paper of **selfcomp** [18] discussed in the previous section. However, as already outlined their approach to address the question *"Does it leak more than k?"* is quite different from ours. Köpf et al. [22] also apply QIF to real-world applications, i.e. leakage of cache side-channels; their technique is based on abstract interpretation and hence not based on bounded models. Because of this however they over-approximate channel capacity.

A preliminary version of the algorithm `SymCount` has been presented in a workshop [30]. In our previous work, we also used Symbolic Execution for *qualitative* information flow analysis [29]. A recent paper [21] explores QIF in a pure logical framework. The approach is powerful and elegant, however it is more limited when compared to our approach as it relies on the solver to generate models whereas our approach can use any solver instead. For example we can analyse Java by using JPF as a solver for bytecode even if JPF doesn't generate a model in the sense of [21].

McCamant and Ernst released FlowCheck [25], a tool for security testing based on dynamic taint analysis. What FlowCheck measures is the number of tainted bits, not an information-theoretic bound, so it is significantly different from our approach. Another tool is described in [27], it is able to analyse large programs using the notion of channel capacity in the context of dynamic taint analysis, while our approach is based on verification techniques. In this sense, our work comes with stronger theoretical guarantees.

5. CONCLUSION AND FUTURE WORK

We introduce *Abstract Model Counting*, a novel approach based on SMT for the quantification of information leaks for real-world applications. Although our implementation is far from being optimised, it drastically outperforms the existing technique based on self-composition. Our approach is applicable to programs with difficult data structures such as pointers, and to Java bytecode.

An original contribution of this work on the theoretical side is: the establishment of connections between measuring confidentiality leaks and fundamental verification algorithms like Symbolic Execution, SMT solvers and DPLL. As a first consequence of these connections we have developed new QIF techniques implemented on two prototype tools for C and Java respectively. We have demonstrated the potential of these tools on C and Java code and argued that while measuring large leaks remains an infeasible task, the most interesting case, i.e. verifying whether a program only leaks a small amount of the secret may be dramatically improved by the techniques here introduced.

An immediate direction for investigation is: instead of dealing with programs as hidden formulas as in this paper, we may view it as an SMT formula w.r.t. the theory of Quantified Bit-Vector, and work directly with the formula. Other interesting directions include to explore the possibility of analysing anonymity protocols under specific probability distributions.

6. ACKNOWLEDGEMENTS:

We thank Dino Distefano, Nikos Tzevelekos, Vladimir Klebanov and the anonymous reviewers for their valuable constructive comments. Pasquale Malacaria's research was supported by grant EP/K032011/1.

7. REFERENCES

[1] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/.

[2] National Vulnerability Database. http://nvd.nist.gov/.

[3] QUAIL. https://project.inria.fr/quail/.

[4] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[5] Backes, M., Kopf, B., and Rybalchenko, A. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009), SP '09, IEEE Computer Society, pp. 141–153.

[6] Balliu, M., Dam, M., and Guernic, G. L. Encover: Symbolic exploration for information flow security. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium* (Washington, DC, USA, 2012), CSF '12, IEEE Computer Society, pp. 30–44.

[7] Barthe, G., D'Argenio, P. R., and Rezk, T. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations* (Washington, DC, USA, 2004), CSFW '04, IEEE Computer Society, pp. 100–.

[8] Barvinok, A. I. A polynomial time algorithm for counting integral points in polyhedra when the

dimension is fixed. *Math. Oper. Res. 19*, 4 (Nov. 1994), 769–779.

[9] Bayardo, R. J. RelSat: A Propositional Satisfiability Solver and Model Counter. http://code.google.com/p/relsat/.

[10] Biondi, F., Legay, A., Traonouez, L.-M., and Wasowski, A. Quail: A quantitative security analyzer for imperative code. In *Proceedings of the 25th international conference on Computer Aided Verification* (Berlin, Heidelberg, 2013), CAV'13, Springer-Verlag.

[11] Birnbaum, E., and Lozinskii, E. L. The good old davis-putnam procedure helps counting models. *J. Artif. Int. Res. 10*, 1 (June 1999), 457–477.

[12] Cerny, P., Chatterjee, K., and Henzinger, T. A. The complexity of quantitative information flow problems. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium* (Washington, DC, USA, 2011), CSF '11, IEEE Computer Society, pp. 205–217.

[13] Chaum, D. The dining cryptographers problem: unconditional sender and recipient untraceability. *J. Cryptol. 1*, 1 (Mar. 1988), 65–75.

[14] Clark, D., Hunt, S., and Malacaria, P. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur. 15*, 3 (Aug. 2007), 321–371.

[15] Clarke, E., Kroening, D., and Lerda, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (2004), vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 168–176.

[16] Darwiche, A. The c2d Compiler. http://reasoning.cs.ucla.edu/c2d/.

[17] Doychev, G., Feld, D., Köpf, B., Mauborgne, L., and Reineke, J. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 431–446.

[18] Heusser, J., and Malacaria, P. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 261–269.

[19] Kiefer, S., Murawski, A. S., Ouaknine, J., Wachter, B., and Worrell, J. Apex: an analyzer for open probabilistic programs. In *Proceedings of the 24th international conference on Computer Aided Verification* (Berlin, Heidelberg, 2012), CAV'12, Springer-Verlag, pp. 693–698.

[20] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[21] Klebanov, V., Manthey, N., and Muise, C. Sat-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 177–192.

[22] Köpf, B., Mauborgne, L., and Ochoa, M. Automatic quantification of cache side-channels. In *Proceedings of the 24th international conference on Computer Aided Verification* (Berlin, Heidelberg, 2012), CAV'12, Springer-Verlag, pp. 564–580.

[23] Malacaria, P. Assessing security threats of looping constructs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 225–235.

[24] Malacaria, P., and Chen, H. Lagrange multipliers and maximum information leakage in different observational models. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security* (New York, NY, USA, 2008), PLAS '08, ACM, pp. 135–146.

[25] McCamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 193–205.

[26] Meng, Z., and Smith, G. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2011), PLAS '11, ACM, pp. 1:1–1:12.

[27] Newsome, J., McCamant, S., and Song, D. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2009), PLAS '09, ACM, pp. 73–85.

[28] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM 53*, 6 (Nov. 2006), 937–977.

[29] Phan, Q.-S. Self-composition by Symbolic Execution. In *2013 Imperial College Computing Student Workshop* (Dagstuhl, Germany, 2013), vol. 35 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 95–102.

[30] Phan, Q.-S., Malacaria, P., Tkachuk, O., and Păsăreanu, C. S. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes 37*, 6 (Nov. 2012), 1–5.

[31] Prosser, P. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence 9* (1993), 268–299.

[32] Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., and Rungta, N. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering* (2013), 1–35.

[33] Smith, G. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures* (Berlin, Heidelberg, 2009), FOSSACS '09, Springer-Verlag, pp. 288–302.

[34] Yasuoka, H., and Terauchi, T. On bounding problems of quantitative information flow. *Journal of Computer Security 19*, 6 (2011), 1029–1082.