Quantifying Information Leaks using Reliability Analysis

(work in progress)

Quoc-Sang Phan* Pasquale Malacaria* Corina S. Păsăreanu† Marcelo d'Amorim‡

* Queen Mary University of London † Carnegie Mellon Silicon Valley, NASA Ames ‡ Federal University of Pernambuco

ABSTRACT

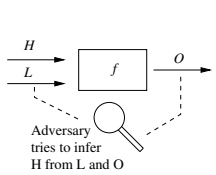
In recent work we have proposed a software reliability analysis technique that uses symbolic execution and model counting to quantify the probability of reaching designated program states, e.g. assert violations, under uncertainty conditions in the environment. The technique has many applications beyond reliability analysis, ranging from program understanding and debugging to analysis of cyber-physical systems. In this paper we report on a novel application of the technique, namely *Quantitative Information Flow* analysis (QIF). The goal of QIF is to measure information leakage of a program by using information-theoretic metrics such as Shannon entropy or Rényi entropy. We exploit the model counting engine of the reliability analyser over symbolic program paths, to compute an upper bound of the maximum leakage over all possible distributions of the confidential data.

We have implemented our approach into a prototype tool, called QILURA, and explore its effectiveness on a number case studies.

1. INTRODUCTION

Quantitative information flow analysis (QIF [7, 15]) is a rigorous approach to measure information leakage. The intuition is that absolute security is hard to achieve, consequently, under some circumstances, programs with *small* leaks are acceptable as secure. QIF has gained considerable attention in recent years. It has been used to analyse software confidentiality [4, 10, 20, 18], to measure loss of anonymity in communication protocols [6], and to assess leakage of information via side-channel [14, 8]. QIF builds on the hypothesis that a malicious user can make observations on the public input and output data used in a function call to infer confidential data. The technique measures the reduction in uncertainty about the secret as an hypothetical malicious user makes observations on public data.

In recent work we have developed a software reliability analysis technique [9] that uses a bounded symbolic execution to collect a set of symbolic paths over the analyzed programs. The path constraints associated with the paths are combined with given probabilistic usage profiles and analyzed using model counting techniques [1] to quantify the probability of reaching designated program states (e.g. successful termination or the opposite, failure states such as assert violations). In this work we adapt the reliability analysis to QIF by considering information leakage as the failure states and using model counting over the input constraints to quantify the likelihood of leakage assuming a uniform usage profile.




```
int check(int L, H){
  int O;
  if (H < 16)
    O = L + H;
  else O = L;
  return O;
}
```

Figure 1: 2^4 (=4 bits) distinct outputs for $H > 0$.

Example. Figure 1 shows an example function that we use to illustrate QIF. This example has been used in previous related work [20, 18]. It is a convention in the security literature to use the label L ("low") to denote non-sensitive input, to use the label H ("high") to denote sensitive *private* input, and to use the label O ("output") to denote the output. A malicious user has access to the public data, L and O, and tries to infer the hidden secret, H, from that.

Automating QIF analysis is a challenge. For example, to analyse the program above, in [18] and more recently [19], the authors manually transformed it into bit vector predicates. Other papers require users to have verification expertise to use an interactive theorem prover [12], or require user to write a driver following a template [10], or to modify the program under test [13].

In this paper, we introduce an automated tool, QILURA (**Q**uantify **I**nformation **L**eaks **U**sing **R**eliability **A**nalysis), for QIF analysis. Given a program, and inputs labelled as *high* and *low*, QILURA computes an upper bound on the maximum number of bits that the program can leak to a public observer. Our implementation is done in the context of Java bytecode programs and the SPF [22] symbolic execution engine, extended for reliability analysis [9]. However, the work is general and can be applied in the context of any programming language for which a symbolic execution tool exists.

At a high level, the architecture of QILURA is depicted in Figure 2. The user labels the inputs of the program with *high* and *low*. The program is then passed to SPF to collect

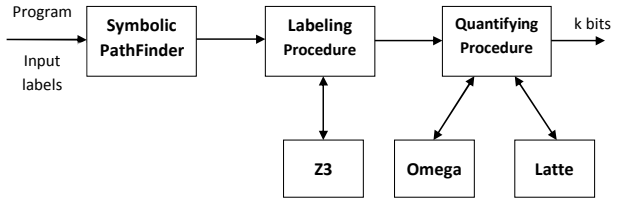


Figure 2: Architecture of QILURA

all possible symbolic paths. The *Labeling Procedure*, using a fine-grained self-composition [5], classifies all the paths into three categories: `clean`, `direct` and `indirect`. The procedure uses z3 [3] for satisfiability checking of self-composition condition.

Finally, the *Quantifying Procedure* uses model-counting techniques [1] over the symbolic constraints (simplified using Omega [2]) collected by SPF to count the number of inputs that follow paths labeled with "direct" and provides an upper bound of k bit on the leaks.

2. PRELIMINARIES

2.1 Quantitative Information Flow

Consider again the program in Figure 1 in the case $L =$ `0x1000`. Clearly, only integer values from `0x1000` to `0x100f` are possible outputs for this function. An attacker has hence 16 possible output choices depending on the value of H: observing outputs `0x1001` .. `0x100f` reveals that H is in the range [1,15] and observing output `0x1000` reveals that the secret is 0 or greater than 15.

Let X_H , X_L and X_O be random variables representing the distribution of H, L, and and O, respectively. Assuming the attacker only knows that H is a 32-bit variable, his *a-priori* probability of guessing the value of H in one try is $\frac{1}{2^{32}}$. which leads to the uncertainty of H measured in Rényi's min-entropy [23] is: $E(X_H) = -\log_2(\frac{1}{2^{32}})$. Moreover, the expected probability of guessing the secret in one try after observing the outputs is:

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

which leads to the remaining uncertainty after observing O is: $E(X_H|X_L = \texttt{0x1000}, X_O) = -\log_2(\frac{16}{2^{32}})$. Leakage is calculated as the reduction of uncertainty of H after the observation

$$\Delta_E(X_H) = E(X_H) - E(X_H|X_L = \texttt{0x1000}, X_O) = \log_2(16)$$

Notice that log(16) = log (number of output observations): this is not a coincidence. A fundamental QIF result (the channel capacity theorem [16, 24]) 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 result holds if we consider not only the above notion of leakage based on the probability of guessing the secret [24] but also the notion of leakage based on Shannon's information theory measuring the number of bits leaked [7]. For these reasons counting the number of observables is the basis of state-of-the-art QIF analysis, e.g. [18, 10, 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. $\log_2(N)$ is the channel capacity (i.e. the maximum leakage) of the program P, denoted by $CC(P)$, measured by Shannon entropy or Renyi's min-entropy.*

2.2 Symbolic Execution

Symbolic Execution (SE) [11], is a program analysis technique which executes programs on unspecified inputs, by using symbolic inputs instead of concrete data. For each executed program path, the analysis builds a path condition pc, i.e. a conjunction of boolean conditions characterizing the inputs that follow that path. This pc is built according to the branching conditions in the code and it is checked for satisfiability using off-the-shelf solvers. If a pc becomes unsatisfiable it means that the corresponding path is not feasible in the program (and the analysis backtracks). The execution paths followed during the symbolic execution of a program are characterized by a symbolic execution tree. The nodes represent program (symbolic) states and the arcs represent transitions between states.

In the setting of SE, the program P is modelled as a (tree-like) transition system:

$$P = (\Sigma, I, F, T)$$

where Σ is the set of symbolic states. $I \subseteq \Sigma$ is the set of initial symbolic states; $F \subseteq \Sigma$ is the set of final symbolic states; and $T \subseteq \Sigma \times \Sigma$ is the transition function. A symbolic path of P is represented by a sequence of symbolic states:

$$\rho = \sigma_0\sigma_1..\sigma_n$$

such that $\sigma_0 \in I, \sigma_n \in F$ and $\langle\sigma_i, \sigma_{i+1}\rangle \in T$ for all $i \in \{0, \ldots, n-1\}$. The symbolic semantics of P is then defined as the set of all symbolic paths $\mathcal{R}$ (i.e. the symbolic execution tree). We define two functions *init* and *fin* to get the initial state and final state of ρ:

$$init(\rho) = \sigma_0 \text{ and } fin(\rho) = \sigma_n$$

We denote by $X|_y$ the value of the variable X at the state y. After symbolically executing the program P with initial input symbols $H = \alpha, L = \beta$, for each $\sigma_i \in F$, i.e. each leaf of the symbolic execution tree, we have a symbolic formula f_i for the value of the output O in the symbolic environment:

$$O|_{\sigma_i} = f_i(\alpha, \beta)$$

The path condition pc_i is a formula $c_i(\alpha, \beta)$ expressing the condition for state σ_i to be reachable. Each pc_i corresponds to a symbolic path ρ_i. We define the function *path* such that: $path(\rho_i) = pc_i \equiv c_i(\alpha, \beta)$.

2.3 Reliability Analysis

In previous work we described a reliability analysis tool [9] based on symbolic execution and model counting. The tool takes as input a Java program and the usage profile and computes an estimate of the probability for satisfying (or violating) a property of interest, e.g. an assertion in the code or a designated observable event. Internally, the tool uses a bounded symbolic execution (SPF [22]) to produce a set of path constraints which are then classified in the sets pc^T and pc^F based on whether the paths lead to the the target event (T) or not (F). A third set pc^G (G stands for "grey") characterizes the paths for which the event did not occur, but the bound was hit (e.g. due to loops or recursion). The path constraints define disjoint input sets and cover the whole input domain of the program [11]. The

tool then quantifies the probability of each set. Note that the computed probability for pc^G results in a measure for the *confidence* in the results obtained within the bound (the lower the probability the higher the confidence); see [9]. For the sake of space we will not consider grey paths here; we will instead label them as "T".

The probability of satisfying the property is defined as the probability of an input distributed according to the usage profile to satisfy any of the path constraints in pc^T. Assuming a uniform usage profile, this probability is $\#(pc^T)/\#(D)$, where $\#(pc^T)$ is the number of solutions satisfying the disjoint constraints in pc^T and $\#(D)$ is the size of the finite (but possibly very large) input domain D. $\#(pc^T)$ can be computed efficiently using model-counting techniques such as Latte [1]. For QILURA we do not compute probabilities but use directly the counts over the computed symbolic constraints.

3. QILURA

At a high level, QILURA has two steps. First, SE is run to collect all symbolic paths of the program (up to a user-specified depth), then each path is assigned a label: (i) **clean**: if it leaks no information, (ii) **direct**: if it leaks information via direct flow, and (iii) **indirect**: if it leaks information via indirect flow. Secondly, we use the model-counting from[9] to count the number of possible inputs that go to "direct" path, and compute an upper bound on the leakage.

3.1 Fine-grained self-composition

Given a program P that takes secret input H, public input L and producing public output O, we denote by P' the same program as P, with all variables renamed: H as H', L as L' and O as O'. Following [5] we express self-composition by the Hoare triple:

$$\{L = L'\}P; P'\{O = O'\} \tag{1}$$

This Hoare triple states that if the precondition $L = L'$ holds, then after the execution of $P; P'$, the postcondition $O = O'$ also holds. Thus, satisfying the triple guarantees that the program P does not leak information.

We run SE on the self-composed program $P; P'$ with input symbols as follows: $H = \alpha$, $H' = \alpha_1$, $L = L' = \beta$. Thus the precondition automatically holds. Assume the symbolic semantics of P and P' is $\mathcal{R}$ and $\mathcal{R}'$ respectively. The self-composition formula in (1) can be re-written as:

$$\forall \rho \in \mathcal{R}, \rho' \in \mathcal{R}'.path(\rho) \wedge path(\rho') \rightarrow O|_{fin(\rho)} = O'|_{fin(\rho')} \tag{2}$$

In case (2) is violated, if ρ' and ρ are the same symbolic path up to renaming then ρ leaks information via direct flow, otherwise ρ and ρ' leak information via indirect flow.

The implementation for checking self-composition is built from (2) as in Figure 3. The function `isSAT` is implemented by calling the SMT solver z3 [3].

3.2 Model counting for symbolic paths

For a symbolic path ρ, let $\#in(\rho)$ and $\#out(\rho)$ denote the number of concrete inputs and outputs of ρ respectively. Obviously $\#in(\rho_i)$ is $\#(pc_i)$ computed in [9]. After being labeled, all paths are classified into three categories: clean, direct and indirect. So the channel capacity is bounded by:

$$CC(P) \leq \log_2(\Sigma\#out(\rho_c) + \Sigma\#out(\rho_i) + \Sigma\#out(\rho_d))$$

```
for all ρ_i do {
    label[i] ← clean
    φ ← c_i(α,β) ∧ c_i(α_1,β) ∧ ¬(f_i(α,β) = f_i(α_1,β))
    if (isSAT(φ)) then label[i] ← direct
}
for i = 1 to n − 1 do
    for j = i + 1 to n do {
        φ ← c_i(α,β) ∧ c_j(α_1,β) ∧ ¬(f_i(α,β) = f_j(α_1,β))
        if (isSAT(φ)) then {
            if (label[i] = clean) then label[i] ← indirect
            if (label[j] = clean) then label[j] ← indirect
        }
    }
```

Figure 3: Fine-grained self-composition

where ρ_c is the clean path, ρ_i is the indirect path, and ρ_d is the indirect path.

- Since clean paths are not interfered by the confidential input we can replace $\Sigma\#out(\rho_c)$ with 1.
- An indirect path only reveals that the program follows that path, its output are not interfered, and each path has one output. Thus, $\Sigma\#out(\rho_i)$ is just the number of indirect paths.
- We hence only need to compute $\Sigma\#out(\rho_d)$.

A deterministic program can be viewed as a function that maps each input to exactly one output output (denotational semantics). Therefore, the number of inputs is always greater than or equal to the number of possible outputs. This means $\#in(\rho) \geq \#out(\rho)$, and $\Sigma\#in(\rho_d) \geq \Sigma\#out(\rho_d)$.

By using the model counting for symbolic path in [9], we can compute $\Sigma\#in(\rho_d)$, and hence compute an upper bound of channel capacity $CC(P)$.

4. EVALUATION

Automated QIF analysis is notoriously hard. To the best of our knowledge, the only tool for QIF analysis of Java bytecode is our own work jpf-qif [21] which uses SE for QIF analysis, but no model counting. Instead jpf-qif adds the conditions for testing each bit of the output at the end of the program, hence exploring all these conditions using SPF. We compare jpf-qif with QILURA below.

We also compare with BitPattern [18], which computes an upper bound on channel capacity by exploring the relations between every pair of bits of the output. In more recent work [19], BitPattern was improved using new heuristics. We compare QILURA with (the improved) BitPattern on several case studies taken from [18, 19].

Moreover, we consider a special case when the program does not leak any information to assess the effectiveness and precision of our technique in such a corner case. The program does not leak information because the output O is always 0 regardless of the value of the secret H. However, the assignment $O = H$ and the condition $H > 999$ make the program be rejected by other qualitative information-flow techniques, e.g. the ones based on type system [25] or taint analysis.

```
if (H > 999){
  O = -1;
} O = H; O = O - H;
```

Case Study 1: No Flow

Case Study	jpf-qif		QILURA		BitPattern	
	Capacity	Time	Upper Bound	Time	Upper Bound	Time
No Flow	0	2.304	0	0.790	-	-
Sanity check, base =`0x00001000`	4	45.324	4.09	1.066	4	0.036
Sanity check, base =`0x7ffffffa`	4	35.346	4.09	1.049	4.59	0.203
Implicit Flow	2.81	0.897	3	0.796	3	0.011
Electronic Purse	2	1.169	2.32	0.854	2	0.157
Ten random outputs	3.32	1.050	3.32	0.814	18.645	0.224

Figure 4: Capacity and bounds are in bits, times are in seconds. "-" means "not reported".

Results and discussions

Figure 4 summarises our experiment, we take the time from the faster version of BitPattern in [19]. Note that in both [18] and [19], the authors manually transform the programs into bit vector predicates, so there will be extra time if they automate this process.

As shown in the figure, the upper bound computed by QILURA only deviate to a small extent from the exact channel capacity computed by jpf-qif. However, by using a model counting tool, QILURA is much faster.

The BitPattern technique can also compute rather tight upper bound in most of the cases. However, by analysing the relations of pairs of bits, the technique is vulnerable when possible values of the output are not in a specific range, as shown in the last case study.

5. RELATED WORK

Backes et al. [4] describe how to use the model checker ARMC and Latte for QIF analysis. Their technique is very precise but also extremely expensive: it involves input counting to compute the pre-image of the observables; in contrast our input counting is used for counting the observable. The work was extended in [12] which uses KeY, an interactive theorem prover, instead of ARMC, but requires significant user effort. Other works on QIF analysis [17, 20] do not provide formal guarantees and bounds as we do here. The only technique that can precisely determine if a program leaks information is self-composition [5]. QILURA also uses self-composition with the key difference that it is able to determine if a single symbolic path leaks information.

6. CONCLUSION AND FUTURE WORK

In this paper we presented QILURA which embodies a novel application of reliability analysis based on symbolic execution and model counting to Quantitative Information Flow analysis. QILURA is still just a prototype but our preliminary experiments show encouraging results. We plan to make the tool available and to perform larger case studies. We also plan to investigate approximate exploration techniques (instead of the exact, complete exploration presented here), for increased scalability, but with formal statistical guarantees on the results.

Acknowledgment. We wish to thank Antonio Filieri for his support during the development of QILURA. This work is partially supported by the Google Summer of Code 2013 program.

7. REFERENCES

[1] Latte. `http://www.math.ucdavis.edu/~latte/`.
[2] Omega. `http://www.cs.umd.edu/projects/omega/`.
[3] Z3. `http://z3.codeplex.com/`.
[4] Backes, M., Kopf, B., and Rybalchenko, A. Automatic discovery and quantification of information leaks. SP '09, IEEE Computer Society, pp. 141–153.
[5] Barthe, G., D'Argenio, P. R., and Rezk, T. Secure information flow by self-composition. CSFW '04, IEEE Computer Society.
[6] Biondi, F., Legay, A., Traonouez, L.-M., and Wasowski, A. Quail: A quantitative security analyzer for imperative code. CAV'13, Springer-Verlag.
[7] Clark, D., Hunt, S., and Malacaria, P. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur. 15*, 3.
[8] Doychev, G., Feld, D., Köpf, B., Mauborgne, L., and Reineke, J. Cacheaudit: A tool for the static analysis of cache side channels. SEC'13, pp. 431–446.
[9] Filieri, A., Păsăreanu, C. S., and Visser, W. Reliability analysis in symbolic pathfinder. ICSE '13.
[10] Heusser, J., and Malacaria, P. Quantifying information leaks in software. ACSAC '10.
[11] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.
[12] Klebanov, V. Precise quantitative information flow analysis using symbolic model counting. QASA '12.
[13] Klebanov, V., Manthey, N., and Muise, C. Sat-based analysis and quantification of information flow in programs. vol. 8054 of *QEST '13*. pp. 177–192.
[14] Köpf, B., Mauborgne, L., and Ochoa, M. Automatic quantification of cache side-channels. CAV'12, pp. 564–580.
[15] Malacaria, P. Assessing security threats of looping constructs. POPL '07, ACM, pp. 225–235.
[16] Malacaria, P., and Chen, H. Lagrange multipliers and maximum information leakage in different observational models. PLAS '08, ACM, pp. 135–146.
[17] McCamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. PLDI '08, ACM, pp. 193–205.
[18] Meng, Z., and Smith, G. Calculating bounds on information leakage using two-bit patterns. PLAS '11.
[19] Meng, Z., and Smith, G. Faster two-bit pattern analysis of leakage. QASA '13.
[20] Newsome, J., McCamant, S., and Song, D. Measuring channel capacity to distinguish undue influence. PLAS '09, ACM, pp. 73–85.
[21] Phan, Q.-S., Malacaria, P., Tkachuk, O., and Păsăreanu, C. S. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes 37*, 6, 1–5.
[22] Păsăreanu, C. S., and Rungta, N. Symbolic pathfinder: symbolic execution of java bytecode. ASE '10, ACM, pp. 179–180.
[23] Rényi, A. On measures of entropy and information. In *Proc. of the 4th Berkeley Symposium on Mathematical Statistics and Probability* (1961), pp. 547–561.
[24] Smith, G. On the foundations of quantitative information flow. FOSSACS '09, pp. 288–302.
[25] Volpano, D., Irvine, C., and Smith, G. A sound type system for secure flow analysis. *J. Comput. Secur. 4*, 2-3 (Jan. 1996), 167–187.