Quantifying Information Leaks using Reliability Analysis

Quoc-Sang Phan† Pasquale Malacaria† Corina S. Păsăreanu‡ Marcelo d’Amorim‡
† Queen Mary University of London, UK
‡ Carnegie Mellon Silicon Valley, NASA Ames, USA
‡ Federal University of Pernambuco, Brazil

{q.phan,p.malacaria}@qmul.ac.uk
corina.s.pasareanu@nasa.gov
damorim@cin.ufpe.br

ABSTRACT

We report on our work-in-progress into the use of reliability analysis to quantify information leaks. 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 analyzer 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 of case studies.

Categories and Subject Descriptors
D.2.4 [Software/Program Verification]: Formal methods; D.4.6 [Security and Protection]: Information flow controls

General Terms
Algorithms, Performance, Security, Theory, Verification

Keywords
Quantitative Information Flow; Reliability Analysis; Model Counting; Symbolic Execution

1. INTRODUCTION

Quantitative information flow analysis (QIF [7, 14]) is a rigorous approach to measuring information leakage. The intuition is that absolute security is hard to achieve, consequently, under some circumstances, programs with small vulnerabilities can leak sensitive information. In this work we adapt the reliability analysis to QIF by considering information leakage as the failure states and 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).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Figure 1 shows an example function that we use to illustrate QIF. 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 secret, $H$, from that.

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

Automating QIF analysis is a challenge. For example, to analyze the program above, in [16] and more recently [17], 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 instrument the application 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 introduce an automated tool, QILURA (Quantify Information Leaks Using Reliability Analysis), for QIF analysis. Given a program, and inputs labeled as high and low, QILURA computes an upper bound on the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
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 [21] 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.

![Figure 2: Architecture of QILURA](image)

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 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 bits 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 \( O \), 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 R´enyi’s min-entropy [22] is:

\[
\log_2\left(\frac{2^{32}}{2}\right)
\]

Moreover, the expected probability of guessing the secret in one try after observing the outputs is:

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

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

\[
\Delta E(X_H) = E(X_H) - E(X_H|X_L = 0x1000, X_O) = \log_2(16)
\]

Notice that \( \log(16) = \log \) (number of output observations); this is not a coincidence. A fundamental QIF result (the channel capacity theorem [15, 22]) shows that leakage for a program is always less than 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 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. \([16, 10, 20, 13, 17, 19]\), 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 R´enyi’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: \( P = (Σ, I, F, T) \), where \( Σ \) is the set of symbolic states; \( I ⊆ Σ \) is the set of initial symbolic states; \( F ⊆ Σ \) is the set of final symbolic states; and \( T ⊆ I × Σ \) is the transition function. A symbolic path of \( P \) is represented by a sequence of symbolic states: \( ρ = σ_0σ_1...σ_n \), such that \( σ_0 ∈ I, σ_n ∈ F \) and \( (σ_i, σ_{i+1}) ∈ T \) for all \( i ∈ \{0,1,...,n−1\} \). The symbolic semantics of \( P \) is then defined as the set of all symbolic paths \( R \) (i.e. the symbolic execution tree).

We define two functions \( init \) and \( fin \) to get the initial state and final state of \( ρ \): \( init(ρ) = σ_0 \) and \( fin(ρ) = σ_n \).

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 \( H = α, L = β \), for each \( σ_i ∈ F \), 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: \( O|_i = f_i(α, β) \). The path condition \( pc_i \) is a formula \( c_i(α, β) \) 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: \( path(ρ_i) = pc_i ≡ c_i(α, β) \).

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 [21]) 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 target event (T) or not (F). A third set \( pc_C \) (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
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 performs a two-step analysis. 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, a model counter for symbolic paths from [9] is used to count the number of leaks information via indirect flow. Secondly, a model counter labeled, all paths are classified into three categories: clean, direct, and indirect. So the channel capacity is bounded by:

$$CC(P) \leq \log_2(\Sigma\#\text{out}(\rho_i) + \Sigma\#\text{out}(\rho_d) + \Sigma\#\text{out}(\rho_d))$$

where $\rho_i$ is the clean path, $\rho_d$ is the indirect path, and $\rho_d$ is the indirect path.

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

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

By using the model counting engine for symbolic paths in [9], we can compute $\Sigma\#\text{in}(\rho_d)$, 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 [20] 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 [16], which computes an upper bound on channel capacity by exploring the relations between every pair of bits of the output. In more recent work [17], BitPattern was improved using new heuristics. We compare QILURA with (the improved) BitPattern on several case studies taken from [16, 17].

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 systems or taint analysis.

**Results and discussions**

Figure 4 summaries our experiment, we take the time from the faster version of BitPattern in [17]. Note that in both [16] and [17], 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 bounds computed by QILURA only deviate to a small extent from the exact channel capacities computed by jpf-qif. However, by using a model counting tool, QILURA is much faster.

The BitPattern technique can also compute rather tight upper bounds 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 ARMCL, but requires significant user effort.

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. A preliminary idea on verifying self-composition using SE has been reported in a student workshop [18].

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 perform larger case studies, and to investigate approximate exploration techniques (instead of the exact, complete exploration presented here), for increased scalability, but with formal statistical guarantees on the results.

7. 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.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>jpf-qif</th>
<th>QILURA</th>
<th>BitPattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Capacity</td>
<td>Time</td>
<td>Upper Bound</td>
</tr>
<tr>
<td>No Flow</td>
<td>0</td>
<td>2.304</td>
<td>0</td>
</tr>
<tr>
<td>Sanity check, base =0x00001000</td>
<td>4</td>
<td>45.324</td>
<td>4.09</td>
</tr>
<tr>
<td>Sanity check, base =0x7fffffff</td>
<td>4</td>
<td>35.346</td>
<td>4.09</td>
</tr>
<tr>
<td>Implicit Flow</td>
<td>2.81</td>
<td>0.897</td>
<td>3</td>
</tr>
<tr>
<td>Electronic Purse</td>
<td>2</td>
<td>1.169</td>
<td>2.32</td>
</tr>
<tr>
<td>Ten random outputs</td>
<td>3.32</td>
<td>1.050</td>
<td>3.32</td>
</tr>
</tbody>
</table>

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

8. REFERENCES