Symbolic Quantitative Information Flow

Quoc-Sang Phan, Pasquale Malacaria
Queen Mary University of London

Oksana Tkachuk, Corina S. Păsăreanu
NASA Ames Research Center

ABSTRACT

Quantitative Information Flow (QIF) is a powerful approach to quantify leaks of confidential information in a software system. Here we present a novel method that precisely quantifies information leaks. In order to mitigate the state-space explosion problem, we propose a symbolic representation of data, and a general SMT-based framework to explore systematically the state space. *Symbolic Execution* fits well with our framework, so we implement a method of QIF analysis employing Symbolic Execution.

We develop our method as a prototype tool that can perform QIF analysis for a software system developed in Java. The tool is built on top of Java Pathfinder, an open source model checking platform, and it is the first tool in the field to support information-theoretic QIF analysis.

Keywords

Algorithms, Security, Verification

1. INTRODUCTION

Protecting confidential data is always a big concern when building a software system. Intuitively, a system is considered to be secure if confidential data cannot be inferred by an attacker through his observations of the system. This intuitive policy is formalized as *non-interference*: an observable output O is not affected by any confidential data H. *Non-interference* has been widely studied during the last four decades [14]. Although satisfying *non-interference* is a sound guarantee for a system to be secure, this classifies many intuitively secure programs as insecure. Consider for example the following password checking program:

```
if (H == L)    accept   else   reject
```

The program takes a public input L controlled by the user, grants access if L is equal to the confidential password H, and rejects otherwise. With a strong enough password, this program is intuitively secure. However, it violates *non-interference* since the result of "accept" or "reject" depends on the confidential data H. Therefore, qualitative methods would classify the program as insecure. This is the main weakness of qualitative methods, which makes the analysis imprecise for a large class of programs.

Quantitative Information Flow (QIF) has been developed to address the limitation of qualitative information flow [7, 8]. The key idea of QIF is simple: instead of accepting only programs with "zero interference" (*non-interference*) as secure, we quantify the interference and accept programs with "*small*" interference as secure. To illustrate the concept of QIF, we consider an attacker model in which an attacker observes the program P and tries to infer the value of the confidential data H of the program by observing the output O of P. Suppose we have a function F measuring the attacker's knowledge of the secret. Initially, the knowledge of H is $F(H)$; we note the attacker's knowledge about H after observing the output O by $F(H|O)$. We then define leakage as the difference between the *a priori* and *a posteriori* attacker's knowledge, i.e.

$$\Delta_F(H) = F(H) - F(H|O)$$

If the program satisfies *non-interference*, observing O will not affect the attacker's knowledge of H:

$$F(H|O) = F(H) \text{ thus } \Delta_F(H) = 0$$

In general the security policy is relaxed from 0 to an acceptable threshold k. This enables QIF to tolerate small leaks and accept more programs as secure.

The function F is in general based on Information Theory. A natural choice for F, when interpreting knowledge as information, is Shannon entropy. An alternative, if one views knowledge in terms of the *probability* of guessing the confidential data in one try, is to choose F as Renyi's min-entropy [15]. Alternatively, F can also be guessing entropy [11], if knowledge is interpreted as the *expected number of guesses* to reveal the confidential data. Readers who are interested in algebraic foundations of QIF may consult [9, 15] for more details.

2. BACKGROUND

There is always a trade-off between precision and computational complexity, and this is the case for QIF. While providing more precision in reasoning about security, QIF also introduces more complexity than traditional qualita-

DOI: 10.1145/2382756.2382791

tive methods. It is impractical to perform QIF analysis manually as in [8] for real-world applications. That leads to the need of automated, possibly approximated, methods for QIF. Previous work [10, 15] has proved that:

$$\Delta_F(H) \leq \log_2(N) \quad (1)$$

with N as the number of possible values for the output O. This result is important, since it frees us from the tedious and expensive task of calculating various (conditional) probabilities on software data. Thus, the problem of QIF analysis is reduced to the problem of counting N. From now on, when we talk about "precise" or "approximate" QIF analysis, we also mean precise or approximate calculation of N, since precise calculation of $\Delta_F(H)$ is almost impossible.

However, even if the result in (1) reduces the complexity of a QIF analysis, the problem of an automated analysis is far from being solved: counting the number of possible outputs N remains a huge challenge. Suppose a program P takes k inputs $I_1, I_2, ..I_k$ and produces an output O. P can be viewed as a function:

$$f : \mathcal{D}_1 \text{ x } \mathcal{D}_2 \text{ x } ..\mathcal{D}_k \rightarrow \mathcal{D}_o$$

where $\mathcal{D}_1, \mathcal{D}_2, ..\mathcal{D}_k$ and $\mathcal{D}_o$ are domains of $I_1, I_2, ..I_k$ and O respectively. In a simple case where data are 32-bit integers, a domain D_i already ranges up to $2^{31}-1$. To count the number of possible outputs, one may be tempted to think of an exhaustive counting as follows:

```
N = 0
for all v in D_o do
    if (assert O != v is violated) then
        N ← N + 1
    end if
end for
return N
```

The assertion can be verified by formal methods such as model checking. Assuming we have a very powerful model checking tool that can verify each assertion in one second, the procedure would take around 2^{32} seconds, which is approximately 136 years. In fact, it is easy to prove by Rice's theorem[1] that the problem of counting possible values of a program output is *undecidable*.

The first automated method that could precisely quantify information leaks was proposed by Backes et al. [5]. The method can be divided into two stages: first, it employs model checking to compute an equivalent 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 [6] can be used to count the number of integer solutions of $\mathcal{R}$. While this is an important work as the first effort on automation of QIF analysis, there is little hope that real-world programs will satisfy this condition.

Meng et al. introduce an approximate method, which we call BitPattern, to calculate an upper bound on the number of possible outputs in [12]. The key idea is as follows: if all possible values of the output O are very close to each other in the state space, then the bit vectors representing them have a lot of similarities, called *bit patterns*. The authors discover these patterns as a SAT formula, then employing Mathematica [4], a heavyweight commercial #SAT solver, to count the number of solutions, which is an upper bound on the number of possible outputs. The authors restrict themselves to C-like programs without loops, and leave automation of the method for future work.

Contribution. In summary, our contributions in this paper are as follows:
1. We propose a novel method for precise QIF analysis.
2. We analyse the BitPattern method, redesign it to make it fully automated and able to handle Java programs.
3. We implement the two methods in an open-source prototyping tool: **jpf-qif**, the first one to support information-theoretic QIF analysis.

3. METHODOLOGY

This section describes our method which we call Symbolic Quantitative Information Flow (SQIF).

3.1 A symbolic representation

The key idea behind SQIF is that instead of checking every concrete value one by one, we build a procedure to process multiple values at a time. To do this, we need a representation that can denote a set of values. The way a computer stores data in vectors of bits suggests a symbolic representation that we define as follows: we consider a set of boolean variables $\Phi := \{p_1, p_2, .., p_K\}$, representing a variable of K bits in the computer memory. In order to count the models representing possible outputs of program P, we need a formula Γ such that each model of Γ corresponds to a concrete value of the output O. We also need a decision procedure $isSAT$ to check the satisfiability of Γ.

For a program P with an output O of data type of size K, O is stored in the computer memory as a bit vector bv_o of K bits: $bv_o = b_K b_{K-1}..b_1$. We set up Γ as a bijective mapping $\Gamma : bv_o \rightarrow \Phi$, which can be implemented by adding the code in Figure 1 to the original program. The symbolic

```
for all element b_i in vector bv_o do
    if (b_i == 1) then
        p_i = True
    else
        p_i = False
    end if
end for
```

Figure 1: Symbolic representation conversion

formula p_1 is a shorthand notation for the family of sets $\{\Phi := \{p_1, p_2, .., p_K\} : p_1 = True\}$. This family of sets can represent up to 2^{K-1} concrete values. Similarly, $p_1 \wedge \neg p_2$ represents a family of sets representing up to 2^{K-2} concrete values. 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[2] framework to systematically explore Φ.

[1] http://en.wikipedia.org/wiki/Rice's_theorem

[2] http://en.wikipedia.org/wiki/DPLL_algorithm

DOI: 10.1145/2382756.2382791

3.2 A symbolic QIF framework

The exhaustive counting procedure discussed in the previous section is inefficient, but intuitive: the only solution for precise counting is searching for all possible solutions. The source of inefficiency is that the procedure checks all concrete values of the data type, while the domain of the output O is often small (note that we aim to tolerate small leaks, not to quantify the big ones). The symbolic QIF framework we propose here systematically explores the symbolic representation Φ of the output, and trims infeasible values in the process. For example, if we discover that p_1 is a tautology, then we can trim 2^{31} concrete values represented by $\neg p_1$, thus approximately saving 68 years compared with the exhaustive counting.

A high level framework to explore the state-space and quantify the leaks of confidential data is described in Figure 2. The recursive call *SymCount* is described in Figure 3. Φ, Ψ and N are passed by reference, while pc and i are passed by value. Φ is the symbolic representation described in the previous section, and Ψ keeps the models of Γ being explored. N is the cardinality of Ψ, and the procedure *SymbolicQIF* returns $\log_2(N)$ as the maximum leakage of confidential data. K is the size of the data type, e.g. $K = 32$ if O is a 32-bit integer, and i is the depth of the recursive call. In the process of exploring the state-space, the explored predicates are kept in a *trace*, which we name pc in referring to path condition in symbolic execution, discussed further in the next section; pc is incrementally updated when the search progresses.

function SymbolicQIF(Φ)
  $\Psi = \epsilon$, $N = 0$, $i = 1$
  $pc \leftarrow InitializePC()$
  SymCount(Φ, Ψ, N, pc, i)
return Ψ, $\log_2(N)$
end function

Figure 2: Symbolic QIF analysis

```
1: function SymCount(Φ, Ψ, N, pc, i)
2:     Extract p_i from Φ
3:     pc_1 ← pc ∧ p_i
4:     if (isSAT(pc_1)) then
5:         if (i == K) then
6:             Ψ ← Ψ ∪ {pc_1}
7:             N ← N + 1
8:         else
9:             SymCount(Φ, Ψ, N, pc_1, i + 1)
10:        end if
11:    end if
12:    pc_2 ← pc ∧ ¬p_i
13:    if (isSAT(pc_2)) then
14:        if (i == K) then
15:            Ψ ← Ψ ∪ {pc_2}
16:            N ← N + 1
17:        else
18:            SymCount(Φ, Ψ, N, pc_2, i + 1)
19:        end if
20:    end if
21: end function
```

Figure 3: Symbolic counting for QIF

At the heart of any QIF method is always a counting technique, and ours is described in the method *SymCount* in Figure 3. To illustrate the technique, we consider a case study of data sanitization from [12], which is shown in Figure 4. All the data are 32-bit integers, the confidential data

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

Figure 4: A data sanitization program

H is only manipulated if it is in an acceptable range, namely from 0 to 15. However, while manipulating H, the program leaks information in the process. It is trivial to prove that only integer values from 8 to 23 are possible outputs of this program, i.e. the number of possible outputs $N = 16$.

At the beginning, all variables are initialised as in Figure 2, the method *InitializePC* sets pc as empty. The method *SymCount* is then called to count the number of possible models of Γ. When a variable p_i is in consideration, we

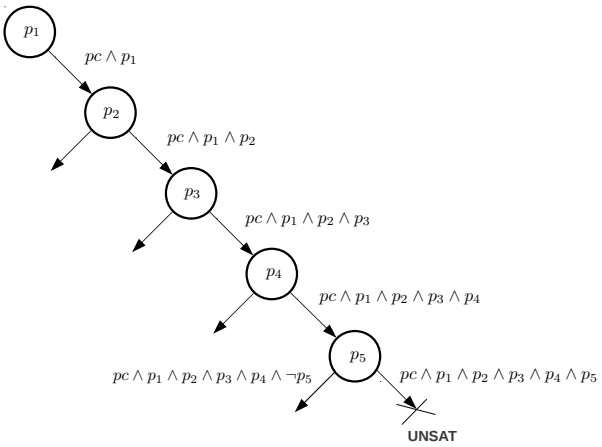

Figure 5: An exploration path of SQIF

systematically explore in the same way for both p_i and $\neg p_i$. Hence, the block of code from line 3 to line 11, and the one from line 12 to line 20 in Figure 4 are symmetric; we only explain the first one.

A trace of the method is described in Figure 5. At the first call of *SymCount*: $i = 1$, the variable p_1 is in consideration, SQIF takes one step forward by adding p_1 to the trace like in line 3. Since pc is initialised to be empty, $pc_1 = p_1$. The method *isSAT* is called to check whether $\Gamma \vDash p_1$ holds, which can be done by using assertion to check the validity of $\neg p_1$ in a Java program as follows:

assert $!p_1$;

A model checking tool like Java Pathfinder (JPF) [1] can be used to verify this assertion, *isSAT* will return *True* if the assertion fails, and *False* otherwise. In this example, *isSAT* would return *True* since all odd values from 9 to 23 are possible outputs satisfying the condition p_1. Therefore, SQIF proceeds by calling *SymCount* with $i = 2$. Similarly,

DOI: 10.1145/2382756.2382791

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 $isSAT$ 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-bit integer, namely $K = 32$, 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 12, then $isSAT$ 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 4 leaks at most 4 bits.

4. IMPLEMENTATION

We choose to apply our method to programs developed in Java. For Java or C/C++, extracting the i^{th} bit from a variable O is fairly easy:

$$b_i = (O \gg i) \;\&\; 1 \tag{2}$$

In this way, we can construct the bit vector $bv_o = b_K b_{K-1} .. b_1$ for the output O. The constraint mapping Γ is implemented by adding the code in Figure 1 to the original program as we have already explained. In the previous section, we explained our framework in which $InitializePC$ sets pc to empty, and $isSAT$ is implemented by verifying an assertion. However, to implement the framework from scratch is costly, and there is a software analysis technique that comes with a similar idea of using symbols to present sets of concrete values, which is symbolic execution. This technique fits with our framework by nature, so we have adapted it into a QIF analysis tool.

4.1 SQIF by Symbolic execution

Symbolic execution (SE) is a hybrid of verifying and testing, which means that it can be either complete or incomplete. In order to use SE as a verifying technique, we assume a bounded model of runtime behaviour, which means programs always terminate and have no recursion, loops can be unfolded and so on. These are well-known issues in symbolic execution and handling them is orthogonal to our work.

In our framework in Figure 4, we name the trace pc in referring to path condition because of the similarity in the way they are updated. For an **if** statement with condition c, there are three possible cases: (**i**) $pc \vdash c$: SE chooses the **then** path; (**ii**) $pc \vdash \neg c$: SE chooses the **else** path; (**iii**) $(pc \nvdash c) \wedge (pc \nvdash \neg c)$: SE executes both paths: in the **then** path, it updates the path condition $pc_1 = pc \wedge c$, in the **else** path it updates the path condition $pc_2 = pc \wedge \neg c$. The third case (**iii**) is interesting to us, since it is similar to how we update the trace pc. Moreover, for a trace pc, the first two cases (**i**) and (**ii**) cannot occur because when we consider a variable p_i, the trace only contains variables from p_1 to p_{i-1}.

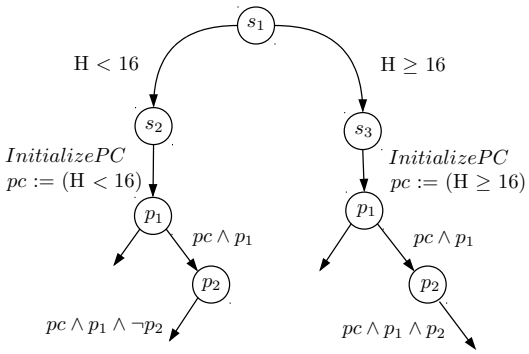

Figure 6: An exploration path of SQIF-SE

In Figure 1, since each bit can take only two possible values, 0 or 1, the way we set up the constraints Γ explicitly imposes the conditions that describe all combination of p_i. Thus, SE is forced to explore all the state space of O. SE as implemented in Symbolic Pathfinder (SPF) [13] also returns a concrete value satisfying the path condition. For example, with the condition $H < 16$ in the program in Figure 4, SPF may return $H = 1$. However, a concrete value of the set $p_1 p_2 ... p_K$ can represent only one value of O. Hence, we only need to count the concrete values of O that SPF generates for the constraints in Figure 1.

The implementation of SQIF by SE, which we call SQIF-SE, is briefly described in Figure 6; the example is still the one that we analysed with the SQIF framework. At the beginning, the trace pc is not initialised as empty but as the path condition of the program, e.g. $H < 16$ in this example. For each possible path, we explore the state space of O as we discussed in the previous section.

4.2 Automation of BitPattern method

In order to discover *bit patterns*, Meng et al. [12] transform the program into a set of Static Single Assignment predicates. These predicates are then translated into the language of STP solver [3] The whole process is performed manually by the authors. Bit patterns are discovered by making queries about the bits of the output O: for example the following STP query tests whether bit i of O is necessarily 0.

$$\text{QUERY}(O[i:i] = 0\text{bin}0)$$

We redesign the BitPattern method, and make it fully automated and able to handle Java programs: we extract bits from the output as in (2), then replace the queries about the bits by making assertions about them. For example, the following assertion is logically equivalent to the query above:

$$\textbf{assert } b_i == 0;$$

The assertions are checked by JPF. With the case study in Figure 4, BitPattern discovers that the assertion above is valid for $6 \leq i \leq 32$. For $1 \leq i \leq 5$, neither $b_i == 0$ nor $b_i == 1$ are valid. Thus, the bit vectors representing possible outputs of O have the same pattern: 00..0*****, where a '*' represents a bit that can be flipped. Also by verifying assertions, BitPattern discovers further that $b_4 b_5$ together can only take two values 01 or 10. Therefore, the number of solutions for the bit patterns is 2 x 2^3 = 16.

DOI: 10.1145/2382756.2382791

In general, we transform bit patterns into CNF formula in DIMACS format, then use RelSat [2], a lightweight open-source #SAT solver, to perform model counting.

Both SQIF-SE and BitPattern are implemented in an open-source prototyping tool, **jpf-qif**, as an extension of JPF.

5. PRELIMINARY EXPERIMENTS

In the illustrative example in Figure 4, there is a *direct flow* from the confidential data H to the output O via assignment. Consider another example in which the program *indirectly* copies H to O if $H \leq 6$ as follows:

```
O = 0;
if (H == 0) O = 0;
else if (H == 1) O = 1;
else if (H == 2) O = 2;
else if (H == 3) O = 3;
else if (H == 4) O = 4;
else if (H == 5) O = 5;
else if (H == 6) O = 6;
else O = 0;
```

Meng et al. reported an elapsed time of 45 ms, and BitPattern returns a bound of 3 bits. With the same example, it takes SQIF-SE 717 ms to count 7 possible outputs, and concludes a bound of 2.81 bits.

In previous case studies, BitPattern performs quite accurately when possible outputs are in the same range, e.g. 8..23 and 0..6. Consider a similar case study of a family of programs that each have exactly 10 feasible outputs:

```
if (H == r1) O = r1;
else if (H == r2) O = r2;
...
else if (H = r9) O = r9;
else O = r10;
```

When r1 to r9 are generated uniformly and independently, the possible outputs diverge in the state space: BitPattern greatly overestimates the bound on information leakage. With 20 such programs, Meng et al. reported an average result of a bound of 18.645 bits in 5 seconds. On the other hand, SQIF-SE always finds exactly 10 outputs in around 1 second, which results in a bound of 3.322 bits.

6. CONCLUSION

In this paper we propose SQIF, a novel method to mitigate the state-space problem in QIF analysis. We have built **jpf-qif**, the first tool to support information theoretic QIF analysis. Compared with the precise QIF analysis in [5], SQIF can handle non-linear relations of various data types. Compared with BitPattern, SQIF is always more precise, but it is slower when leaks are in the same range. In contrast, SQIF is better in both effectiveness and efficiency when leaks diverge in the state space. An immediate direction for investigation is to combine SQIF with a cheap qualitative method, e.g. type system, to decide quickly if the program satisfies *non-interference*. This will improve the efficiency of the tool when programs are strictly secure.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. The development of this project is funded by the Google Summer of Code 2012 program.

7. REFERENCES

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

[2] http://code.google.com/p/relsat/.

[3] http://sites.google.com/site/stpfastprover/.

[4] http://www.wolfram.com/mathematica/.

[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] 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.

[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 (Aug. 2007), 321–371.

[8] 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.

[9] Malacaria, P. Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. *CoRR abs/1101.3453* (2011).

[10] 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.

[11] Massey, J. L. Guessing and entropy. In *In Proceedings of the 1994 IEEE International Symposium on Information Theory* (1994), p. 204.

[12] 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.

[13] Păsăreanu, C. S., and Rungta, N. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 179–180.

[14] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21* (2003), 2003.

[15] Smith, G. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (Berlin, Heidelberg, 2009), FOSSACS '09, Springer-Verlag, pp. 288–302.

DOI: 10.1145/2382756.2382791