A quantitative analyser for programs in a core imperative language

Chunyan Mu
School of Computing Science
Newcastle University, Newcastle NE1 7RU
Email: chunyan.mu@ncl.ac.uk

David Clark
Department of Computer Science
University College London, London WC1E 6BT
Email: david.clark@ucl.ac.uk

***Abstract*—This paper presents a tool for analysing quantified information flow (QIF) for programs written in a core imperative language. The intended application is measuring leakage of secrets. The tool can provide either exact leakage or an upper bound depending on the trade off chosen by the user between exactitude or computation speed. Approximations are created via abstractions derived from partitions on the initial store. We outline the workings of the tool and summarise results derived from running the tool on a range of example programs with either concrete or abstract initial stores.**

I. Introduction

This paper presents an implementation [1] of the concrete leakage analyser discussed in [2] and an extension incorporating the abstract interpretation techniques discussed in [3]. The leakage analyser is a static analyser incorporating probability distribution transformations and leakage computations based on the updated probability distributions. Given the description of an initial state space (in either the concrete or abstract representation) and a source program to analyse, the analyser reads the input files, and parses and interprets the program via the building of an abstract syntax tree. In addition, the analyser computes leakage based on the analysis rules over the syntax tree, and produces a leakage analysis report. The report describes the exact quantity of the leakage or a leakage upper bound into the public output variable of interest. Section II presents the overall structure of the system, and some discussions based on tests. Further details about the QIF analyser can be found in our technical report [4].

II. The QIF Analyser

We built our leakage analyser on the basis of the structure of the CoCo/R Compiler generator [5] in Java. We extended it to be a probability distribution transformer incorporating leakage computation. Figure 1 describes the basic structure of the system. The initial configuration (joint distribution for the vector of variables $\vec{X}$, `.CFG`) and the example program (`.QIF`) are fed into the analyser. The analyser then calculates the distribution transformation and the measurement of the secret information leaked to low components by executions of the program. The semantic rules for transforming distributions used by the parser are based on the concrete probabilistic semantics discussed in [2] and the interval-based abstract probabilistic semantics discussed in [3].

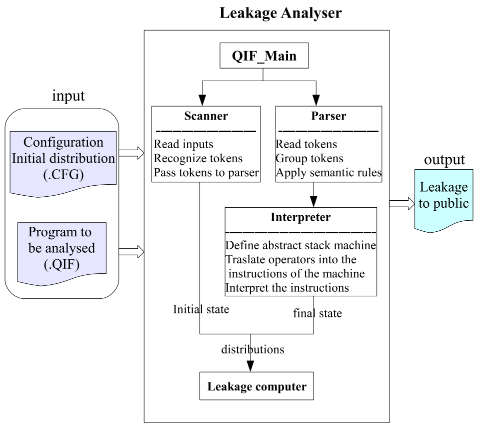

Fig. 1. Implementation: the overall structure

A. An Overview of the Analyser

The analyser has a parser adapted to simulate the semantics. Specifically, our analyser analyses a *core* programming language called "`QIF`". It has variables of type `INTEGER` and `BOOLEAN`. It allows assignments, sequential composition, if and while statements. It has arithmetic expressions $(+, -, *, /, \%)$ and relational expressions $(=, <, >)$. Here is an example of a `QIF` program ("`test.QIF`").

```
PROGRAM test;
VAR h: INTEGER;
VAR l: INTEGER;

BEGIN
  IF h>l THEN
        IF l>5 THEN l := l+1 ELSE l:=l*2; END;
  ELSE l:=0;  END;
  WRITE(l);
END test.
```

Of course the `QIF` language is too restrictive to be used as a real programming language. Its purpose is just to give a taste of how to build automatic leakage analysers based on the techniques discussed in [2], [3]. In addition, the state in our concrete analyser is slightly more complicated than program state in a compiler. Normally the store can be viewed as a map from a tuple of variables to a tuple of values. Here we map a tuple of variables to a probability distribution on the tuples of values. For example, assume we have the initial state space as follows:

$$\langle h, l\rangle \mapsto (\langle 0,1\rangle \to 0.25,\ \langle 1,1\rangle \to 0.25,\ \langle 2,1\rangle \to 0.25,\ \langle 3,1\rangle \to 0.25)$$

In order to deal with *nested* if statements or while loops, we build the measure spaces in a tree like structure. During the interpreting of the program by the analyser, a measure space is updated based on the semantic rules. Intuitively, the analyser will build a tree as follows (see Figure 2) with regard to program `test.QIF` and the above initial state space.

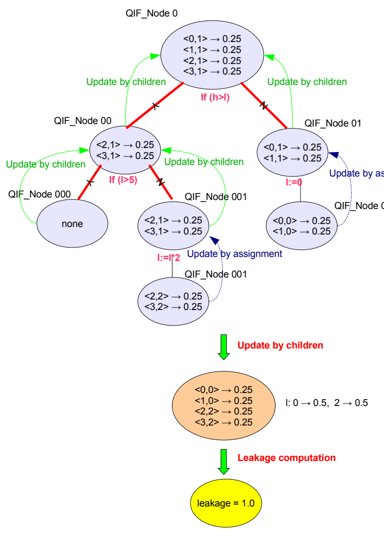

Fig. 2. A tree like structure storing the measure spaces in the analyser

For example, assignments update the space of the current node `QIF_Node`; if statements create partitions on the space of the current node: the left child node stores the partition that enters the true branch and is updated later on, the right child node stores the partition that enters the false branch and is updated later on, then the parent is updated by combining the two children and then removing the children at the end of the if statement; while loops keep creating partitions on the space of the current node with each iteration until no new partitions are created.

Something similar to the above description occurs in the analyser for abstract analysis which uses the abstract semantic rules discussed in [3]: the abstract store is a map from variables to the abstract domain (the interval-based partitions on the probability distributions).

Figure 3 gives the basic flow program of the main class “QIF-Main”. The main function accepts the user’s options, creates scanner and parser objects, and calls the method `parse()` to start parsing and analysing. Finally, it outputs the results of the analysis.

B. Discussion

Having run some examples (for details see [4]), we briefly summarise some points as follows:

- the analyser correctly analyses programs in a concrete or abstract way depending on the chosen option;
- for a fixed program under analysis, the speed of execution of the concrete analyser is affected by the size of the variables: the analyser needs more time when the size of the variables becomes larger. This also implies that the speed of execution of the analyser is affected by the size of the program: as the size of the variable becomes larger the number of iterations of the loops of the program

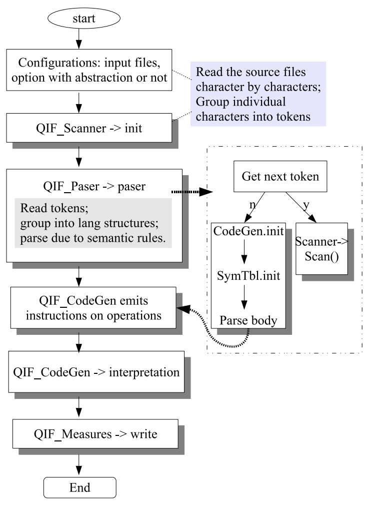
start
Configurations: input files, option with abstraction or not
Read the source files character by characters; Group individual characters into tokens
QIF_Scanner -> init
QIF_Paser -> paser
Read tokens; group into lang structures; parse due to semantic rules.
Get next token
n
y
CodeGen.init
SymTbl.init
Parse body
Scanner-> Scan()
QIF_CodeGen emits instructions on operations
QIF_CodeGen -> interpretation
QIF_Measures -> write
End

Fig. 3. The Main function

become larger, and therefore the unfolded statements of the program become larger;
- the abstract analyser runs much faster than the concrete analyser for a specific program under analysis with a specific initial state space and can tackle the scalability problem of the concrete analyser.
- for a fixed program under abstract analysis, different abstract partition strategies affect the precision of the analysis and the speed of execution of the abstract analyser. There are no general rules specifying how to choose an abstract partition strategy to provide a better abstract analysis with regard to speed and precision of the analysis. This depends on the initial space and the program under analysis. However, intuitively, the more closely the abstraction of the initial space simulates the behaviour of the partitioning by the program under analysis and the behaviour of the uniformalisation, the more precise the result produced by the abstract leakage analysis.

III. Conclusions

In this paper, we presented a tool for providing automatic, quantitative information flow analysis. We gave the basic structure of the system to show how the system works. We discussed the tractability problems associated with the concrete analysis, outlined how the abstraction version attacks tractability problems and how different partitions in the abstract domain affect the result of the leakage analysis.

References

[1] C. Mu, “Qif analyser,” 2010. [Online]. Available: http://www.dcs.kcl.ac.uk/pg/cmu/qif
[2] C. Mu and D. Clark, “Quantitative analysis of secure information flow via probabilistic semantics,” in *ARES*, 2009, pp. 49–57.
[3] ——, “An interval-baseed abstraction for quantifying information flow,” in *Electronic Notes in Theoretical Computer Science*, vol. 59, Elsevier, 2009, pp. 119–141.
[4] C. Mu, “An implementation on the qif analyser,” Department of Computer Science, King’s College London, Tech. Rep. TR-10-08, December 2010.
[5] H. Mossenbock, “Coco/r - a generator for fast compiler front-ends,” 1990.