Quantifier Elimination and Information Flow Control for Software Security

Gregor Snelting[1]

[1] Universität Passau, Lehrstuhl Softwaresysteme, 94032 Passau, Germany

Abstract

Program Dependency Graphs and Constraint Solving can be combined to achieve a powerful tool for information flow control, allowing to check source code for security problems such as external manipulation of critical computations. The method generates path conditions for critical information flows, being conditions over the program variables necessary for flow. As all variables are existentially quantified, quantifier elimination and in particular the REDLOG system developed at Volker Weispfenning's group, are used to solve path conditions for the input variables, thus generating witnesses for security leaks.

1 Introduction

Software safety and security is more important than ever, and international standards such as the Common Criteria [Com99] define a comprehensive set of security techniques. *Information Flow Control* (IFC) is one of the technologies required by the Common Criteria. IFC has two main tasks:

- guarantee that confidential data cannot leak to public variables;
- guarantee that critical computations cannot be manipulated from outside.

State-of-the-art IFC exploits *program analysis* to assign and propagate security levels to variables and expressions, guaranteeing that any potential security leak is found [SM03].

But most contemporary analysis methods are based on non-standard type systems, which are not flow sensitive, context sensitive, or object sensitive. This leads to imprecision and thus to a high number of false alarms. For example, the well-known program fragment

```
if confidential=1 then public:=0 else public:=1; public:=0;
```

is considered insecure by type-based IFC, as it does not see that the potential security leak in the if-statement is guaranteed to be killed by the following assignment.

We therefore proposed to base IFC on a combination of *dependency graphs* and *constraint solving*. While the original idea was already published in 1996 [Sne96], it took us several years to make the approach work and scale for full C and realistic programs [RS02, SRK05].

```
(1)  a = u();
(2)  while (n>0) {
(3)    x = v();
(4)    if (x>0)
(5)      b = a;
(6)    else
(7)      c = b;
     }
(8)  z = c;
```

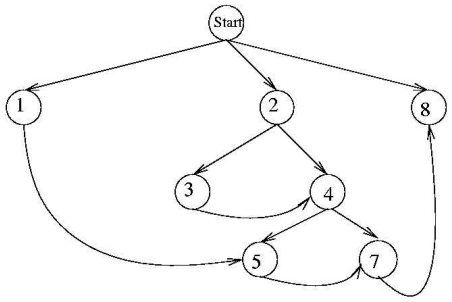

Figure 1: A small program and its dependency graph

In this short contribution, we will describe the basic idea, its connection to constraint solving and quantifier elimination, and our application of Volker Weispfenning's Redlog system.

2 Dependency Graphs and Path Conditions

Dependency graphs are a standard tool to model information flow through a program. Program statements or expressions are the graph nodes. A data dependency edge $x \to y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A control dependency edge $x \to y$ means that the mere execution of y depends on the value of the expression x (which is typically a condition in an if- or while-statement).

A path $x \to^* y$ in the graph means that information can flow from x to y; if there is no path, it is guaranteed that there is no information flow. In particular, all statements influencing y (the so-called *backward slice*) are easily computed as

$$BS(y) = \{x \mid x \to^* y\}$$

For the small program and its dependency graph in figure 1, there is a path from statement 1 to statement 8, indicating that input variable `a` will eventually influence output variable `z`. Since there is no path $(1) \to^* (4)$, there is definitely no influence from `a` to `x`.

But note that the dependency graph is a conservative approximation; due to imprecision of the underlying program analysis algorithms it may contain too many edges (but never too few). For the full C language, the computation of precise dependence graphs and slices is absolutely nontrivial; there is ongoing research worldwide since 15 years. The state of the art in dependency graphs and slicing is summarized in the recent work by Krinke [Kri03].

In order to make the analysis more precise, we introduced *path conditions*, which are necessary conditions for information flow between two nodes. The formulae for the generation of path conditions are quite complex, and only the most fundamental formula will be given

here:

$$PC(x,y) = \bigvee_{P \text{ Path } x \to^* y} \bigwedge_{u \text{ node in } P} E(u)$$

where $E(u)$ is a necessary condition for the execution of u. $E(u)$ is defined by a similar formula containing control conditions from if- and while-statements. Program variables in a path condition are existentially quantified, as they are necessary conditions for potential information flow.

In Figure 1, we have

$$E(1) \equiv true, \quad E(3) \equiv (n > 0), \quad E(5) \equiv (n > 0) \wedge (x > 0)$$
$$PC(1,5) \equiv E(1) \wedge E(5) \equiv \exists n, x.(n > 0) \wedge (x > 0)$$

Slightly more interesting are the following path conditions:

```
(1)  a[i+3] = x;
(2)  if (i>10)
(3)    y = a[2*j-42];
```

$$PC(1,3) \equiv \exists i, j.(i > 10) \wedge (i + 3 = 2j - 42)$$

and

```
(1)  a[i+3] = x;
(2)  if ((i>10)&&(j<5))
(3)    y = a[2*j-42];
```

$$\begin{aligned} PC(1,3) &\equiv \exists i, j.(i > 10) \wedge (j < 5) \\ &\qquad \wedge (i + 3 = 2j - 42) \\ &\equiv false \end{aligned}$$

These examples indicate that path conditions give precise conditions for information flow and can even determine that such flow is impossible even though there is a path in the graph. We will not go into the details of path condition generation. The reader should be aware that making path conditions work for full C and realistic programs required years of theoretical and practical work [RS02, SRK05, Kri03, Rob05]. Today, our implementation ValSoft can handle C programs up to approx. 10000 LOC and generate path conditions in a few seconds or minutes.

3 Solving Path Conditions

Path conditions for realistic programs can be quite complex. But if they can be solved for the program's input variables, they act as a *witness* for a security leak. Providing input values according to the solved conditions makes any illegal information flow visible immediately. This feature is helpful e.g. in law suits against vendors of insecure software.

Solving path conditions for realistic programs is not easy, as they consist of huge heaps of conditions extracted from control statements such as if, while, switch; combined into substantial amounts of conjunctions and disjunctions with existential quantifiers upfront. As a

first step, a minimal disjunctive normal form is computed for the quantifier body (which is a propositional formula) before any further constraint solving is attempted.

Since path conditions are existentially quantified, it is a natural idea to apply quantifier elimination [Wei97, Wei94] and use systems such as Redlog [SW96, DS97]. Quantifier elimination replaces an existentially quantified variable by constraints on other variables, and the theory guarantees that both formulae are equivalent. Thus, we connected Redlog to ValSoft, acting as an external solver after simplification.

For small examples, this works fine. Consider the second and third path condition example from Section 2, and let us assume that j is an input variable but i is an auxiliary variable. Thus we want to eliminate i and solve for j. Eliminating i in the second path condition yields $2j > 55$, while Redlog immediately says *false* (i.e. unsatisfyable) for the third path condition. Even medium-sized conditions can be handled, provided they contain just arithmetic formulae.

The Challenge: Combining Theories

In practice, path conditions not just contain arithmetic formulae, but contain arbitrary expressions from the C language. First of all, C programs contain both integer arithmetic and real arithmetic, while quantifier elimination was originally designed for real arithmetic only. In theory, machine integers can be replaced by a finite disjunction, but for ValSoft this approach is not realistic as it will immediately generate combinatoric explosion in path conditions.

Recently, an elimination algorithm for mixed real-integer arithmetic was proposed [Wei99], but was not yet integrated into REDLOG. For our purposes real quantifier elimination can be applied anyway: if there are no real solutions, there are no integer solutions. This is consistent with the ValSoft principle of conservative approximation, thus possible information flow can never get lost. Real solutions which are not integer will however lead to false alarms.

Data structures provide even more challenging problems, as arithmetic conditions are often combined with data structures or function calls. Here is a small example using a stack:

```
(1)  a[i+3] = x;
(2)  s = push(s, a[2*j-42]);
(3)  if (i>10)
(4)    y = top(s);
```

In the presence of functions, dependencies between parameters must be known. In the example, there is a dependency from the second to the first parameter of `push`. Thus there is a dependency path $(1) \rightarrow^* (4)$, indicating a potential information flow. But the standard path condition is just $PC(1,4) \equiv \exists i.(i > 10)$. This path condition is a necessary condition for information flow, hence correct, but much less precise than the one from section 2.

If source code for all procedures is available, interprocedural dependences and path conditions [Kri03] will handle this problem. But for library functions, there is a smarter approach not requiring source code. For standard data types from libraries such as lists, stacks, or trees,

an equational specification will often be available, which can be exploited for generating path conditions. In the example, the equations $top(push(s,x)) = x$, $pop(push(s,x)) = s$ hold. Application of these equations can be intertwined with path condition generation [SRK05]:

$$\begin{aligned} PC(1,4) &\equiv \exists i,j.(i > 10) \wedge (top(push(s, a[2j-42])) = a[i+3]) \\ &\equiv \exists i,j.(i > 10) \wedge (a[2j-42] = a[i+3]) \\ &\equiv \exists i,j.(i > 10) \wedge (2j-42 = i+3) \end{aligned}$$

which is the same conditions as in section 2. Thus source code for library functions need not be analysed. If nothing is known about a function except its parameter dependencies, this situation is equivalent to the empty equational specification.

Mathematically, we thus want to apply quantifier elimination not just to arithmetic structures, but to combinations of arithmetic with either free algebras (containing uninterpreted function symbols) or with equationally specified algebras. Hence our challenge to the quantifier elimination community:

Challenge: *We need quantifier elimination for combinations of arithmetic with free algebras, or with equationally specified algebras.*

It is outside the expertise of this author to judge whether the challenge is realistic at all, or doomed to fail due to undecidability problems. But our experiences with ValSoft show that many path conditions cannot be solved as they contain not just arithmetic, but other C expressions as well.

4 Conclusion

Path conditions in dependency graphs are very helpful to discover security leaks in software, and quantifier elimination is a natural approach for solving path conditions. Our ValSoft system, implementing this approach, can generate and simplify path conditions for medium-sized C programs (approx. 10 kLOC) in a few minutes. A version for Java is under way.

Unfortunately, current elimination algorithms do not allow for combinations of arithmetic expressions with other algebraic structures. If new algorithms can be found, a boom in program analysis technology will result. And unsafe software will have a hard time in court as quantifier elimination will provide witnesses against it.

References

[Com99] Common Criteria Project Sponsoring Organizations. Common criteria for information technology security evaluation. *ISO/IEC 15408*, 1999.

[DS97] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, July 2003.

[Rob05] Torsten Robschink. *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*. PhD thesis, Universität Passau, Januar 2005. in German.

[RS02] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *Proceedings International ACM/IEEE Conference on Software Engineering (ICSE'02)*, pages 478–488, Orlando, FL, May 2002.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[Sne96] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *Proc. Static Analysis Symposium*, volume 1145 of *LNCS*, pages 332–348, 1996.

[SRK05] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, to appear 2005.

[SW96] Thomas Sturm and Volker Weispfenning. Computational geometry problems in REDLOG. In *Automated Deduction in Geometry*, pages 58–86, 1996.

[Wei94] Volker Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, FMI, Universität Passau, D-94030 Passau, Germany, April 1994.

[Wei97] Volker Weispfenning. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189–208, 1997.

[Wei99] Volker Weispfenning. Mixed real-integer linear quantifier elimination. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, pages 129–136, 1999.

Gregor Snelting is a full professor for computer science at the University of Passau, leading the software technology group. He received his Diploma and PhD from the Technical University of Darmstadt with work on generic type inference in language-based editors. His current interests include program analysis and deductive techniques for software engineering purposes, such as refactoring, reengineering or security checks.

`snelting@fmi.uni-passau.de` `www.fmi.uni-passau.de/st/`