Pattern Matching Security Properties of Code using Dependence Graphs

John Wilander and Pia Fåk, {johwi, x05piafa}@ida.liu.se
Dept. of Computer and Information Science, Linköpings universitet

Abstract

In recent years researchers have presented several tools for statically checking security properties of C code. But they all (currently) focus on one or two categories of security properties each. We have proposed dependence graphs decorated with type-cast and range information as a more generic formalism allowing both for visual communication with the programmer and static analysis checking several security properties at once. Our prototype tool GraphMatch currently checks code for input validation flaws. But several research questions are still open. Most importantly we need to address the complexity of our algorithm for pattern matching graphs, the accuracy of our security models, and the generality of our formalism. Other questions regard the impact of security property visualization and heuristics for ranking of potential flaws found.

Keywords: security properties; dependence graphs; static analysis

1 Introduction

In November 2002 we published a comparative study of five static analysis tools checking C code for buffer overflows and format string vulnerabilities [19]. We used micro benchmarks and our study showed that tools performing lexical analysis produced a lot of false positives (52% to 71%), while syntactical and semantical analysis had problems with too many false negatives (70% to 73%). The latter mainly due to poor vulnerability databases, not the underlying techniques.

Since then many more tools have been developed [1, 3, 5, 12, 6, 15]. The research behind these tools and prototypes is excellent and the empirical results are promising, but it is not evident if and how the techniques can be combined to solve several security problems at once. They all (currently) focus on one or two categories of security properties each and make use of quite different system models, methods of analysis, and also require different amounts of user involvement. In our studies of the modeling formalisms used in the tools we identified a specific problem in modeling security properties of code—*the dual modeling problem*.

Some security problems are typically described as "If you do A you must do B" (e.g. *input validation*). Such properties are best modeled as good programming practice—"do like this". Other security problems are described as "If you do A then you must not do B" (e.g. *double free*). Such properties are best modeled as bad programming practice—"do not do like this". For a formalism to be able to cover the great variety of security properties it needs to be able to model both good and bad programming practice. The dual modeling problem is closely related to *safety* and *liveness properties* of code [11].

A drawback of static analysis tools in general is that they only *detect* vulnerabilities and therefore the user has to know how to patch the code. The aforementioned tools only offer textual information about analysis results. We believe visual information can be helpful for programmers.

Engler and Musuvathi have pointed out the problem of reporting huge amounts of potential bugs as the result of static analysis and model checking—"It's not enough to find a lot of bugs. (...) What users really want is to find the 5-10 bugs that really matter ..." [13]. Therefore we believe it is necessary to automatically *rank* the bugs reported from a security analysis tool.

Our research goal is to implement a tool that can:

- check several types of security properties;
- visually communicate with programmers; and
- rank the severity of potential flaws.

1.1 Paper Overview

In Section 2 we present decorated *dependence graphs* as a generic modeling formalism for code security properties covering control-flow, data-flow, type and range information. Models of two security vulnerability types—integer flaws and double `free()` are shown in Section 3 and 4. Section 5 and 6 briefly explain the implementation of our prototype tool and present our initial results. Finally, Section 7 covers future work and open research questions.

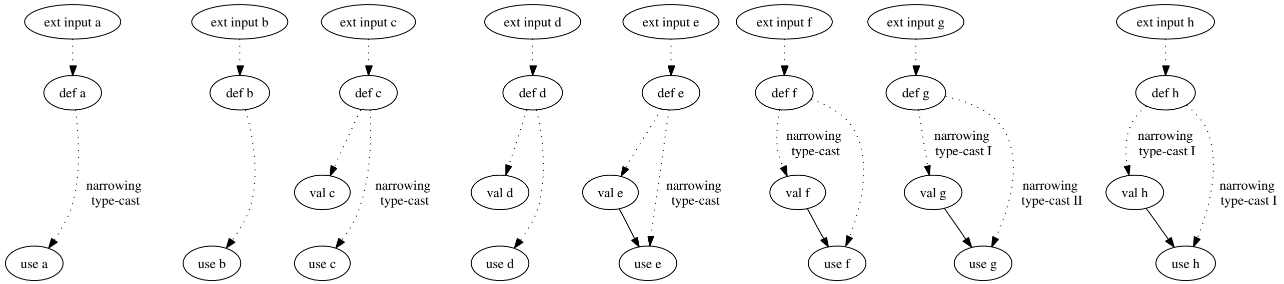

Figure 1. Eight incorrect graph patterns for integer validation. Solid edges represent control dependence and dotted edges represent data dependence. The vertices are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and sensitive use of the variable (use). Roman figures tell if type-casts are different. Severity ranking from left to right stretching from validation point absent to validation with implicit narrowing type-casts.

2 Dependence Graphs

We have proposed decorated dependence graphs as a more generic formalism for visualizing security properties, and performing static analysis of C code [18].

Dependence graphs were first presented by Ottenstein and Ottenstein as an intraprocedural intermediate form—the *program dependence graph*, or *PDG* [14]. Vertices represent statements and predicates (program points), and edges represent control- and data-flow *dependence*. A program point B is *control dependent* on another program point A, if A controls whether B is executed or not. A program point B is *data dependent* on a program point A if some variable x is defined in A and later used in B without any new defines in-between. This means that only necessary temporal constraints are encoded in the graph—it does not include a complete control-flow graph. The interprocedural version, called *system dependence graph*, or *SDG*, was presented by Horwitz *et al* [9]. Dependence graphs were designed to allow for deep analysis of code. They are the underlying structure for *program slicing* [16].

Several so called *narrowing integral type-casts* have constituted security vulnerabilities. Chen *et al* have studied this category of security bugs and summarized the insecure conversions [4]. To detect such flaws we decorate the original SDGs be with type information, specifically implicit type conversions. Type conversion information belongs to edges in the SDG since it is the data-flow between two program points that can include such a conversion, and a program point can be data-flow dependent on several others.

Weber *et al* have used SDGs decorated with range constraint information for string buffers to statically detect buffer overflow vulnerabilities [15]. We will use this technique to check both buffer and integer ranges.

3 Integer Flaws

Several security vulnerabilities prove that handling integers is difficult. The problems mostly arise when integers are used as memory offsets, in pointer arithmetic, and/or when the integer representation changes from signed to unsigned or vice versa [1, 2, 10]. For proper input validation in such sensitive cases, two crucial steps need to be taken; (1) validate integral variables so that narrowing type-casts do not lead to unintended behavior, and (2) validate upper and lower bounds of user affected integral variables before they are used in memory references and calculations.

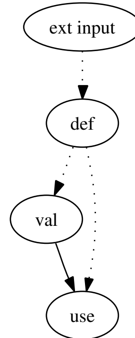

Figure 3. Correct code pattern for integer input validation.

The graph to the left is an example of a model of good programming practice—correct integer input validation. The integer has to be validated before it is used which means that the use point (use) has to be control dependent on the validation point (val), and both use point and validation point have to be data dependent on the input without narrowing type-casts. Modeling of validation points is abstracted away from these models. Using range constraints is a feasible way of doing this [1].

To allow for severity ranking of reported flaws we can encode the dual to the correct code pattern, ending up with a collection of incorrect code patterns, i.e. models of bad programming practice (see Fig. 1).

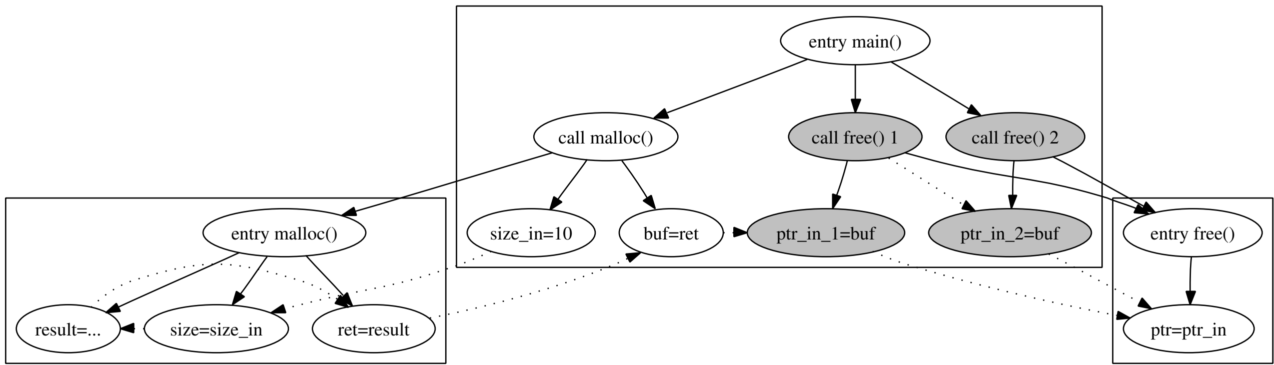

Figure 2. Incorrect graph pattern for malloc and free, where free is called twice. The grey nodes are the bad programming model where two free use the same pointer (shown by a data dependence).

How to model external input is not obvious. Still, many bugs become security vulnerabilities because the user can affect data input. The solution is system and API specific. Apart from file accesses and command line arguments we have followed the pointers by Wheeler who mentions *Environment variables* as untrustworthy sources [17], and Ashcraft and Engler who add another three categories—System calls, routines that copy data from user space, and network data [1].

4 The Double `free()` Flaw

To allocate heap memory, a C program calls `malloc()` and gets a pointer to the allocated memory as return value. When the program is done using the memory it has to be released, which is done with a call to `free()`. To keep track of which parts of heap memory are allocated and which are free, the operating system has to store information. For scalability reasons this information is stored together with each allocated chunk of memory; it is stored "in-band". When memory is freed the in-band information is used to relink the memory chunk with the list of free memory.

Normally, attempting to free the same memory twice or more will lead to undefined behavior, often a *segmentation fault*. But if an attacker can change the memory in between two calls to `free()` he or she can inject false in-band information and potentially compromise the process.

This is an example of bad programming practice. We show in Fig. 2 why the double free has to be modeled as a bad security property. The bad model *contains* the good one. If we ignore one of the calls to `free()` we have a match for correct usage of `free()`. Thus we cannot say a piece of code is secure simply because we have pattern matched a good use of `free()`, we also have to look for bad use of `free()`.

5 Tool Implementation

We have implemented a prototype tool called *GraphMatch* that performs pattern matching using dependence graphs. We build graph models of the programs with Grammatech's tool *CodeSurfer* [8]. Currently GraphMatch can detect integer input validation flaws by following a straight forward algorithm (compare with vertex labels in Fig. 3):

1. Begin at some external input vertex (ext input)
2. Follow transitive data-flow to match definitions (def)
   (a) Follow data-flow to all sensitive uses (use)
   (b) Follow data-flow to all validations (val)
3. Check that all the sensitive uses from 2(a) are control-dependent on some validation in 2(b)

If some part of the program model deviates from the model of correct integer input validation it is reported as a potential flaw. This algorithm has a complexity of $O(E * V^h)$, where E and V are the number of edges and vertices in the program model, and h is the depth of the security property model.

6 Initial Results

The GraphMatch prototype performs well on our synthesized micro benchmarks whereas real-life applications pose a harder problem. We checked wu-ftpd 2.6-4 which consists of approx. 20.000 lines of code and produces a dependency graph with approx. 130.000 vertices. An analysis for integer input validation flaws took 15h on a 2.66 GHz Pentium 4. GraphMatch produced three warnings, two false positives and one true positive. The false positives were due to inaccuracy of our "sensitive use" model. The true positive was clearly a missing input validation but didn't seem exploitable.

7 Future Work

Defining the modeling formalism was the first step toward a single tool able to check for several security properties. Apart from modeling other security properties and checking them with real-life code, we have several open research questions to address:

Complexity. Not too surprisingly, our initial results show that our graph matching has high complexity. It might be that dependence graph matching can be reduced to the *subgraph isomorphism problem* which is shown to be NP-complete [7]. Even so, we will investigate how heuristic trade-offs leading to unsoundness and/or incompleteness can affect practical performance.

Accuracy. How much does the inevitable inaccuracy of the underlying program analysis affect the accuracy of our pattern matching?

Generality. Are dependency graphs suitable for modeling a great variety of security properties of code? Are they suitable for analysis of other languages than procedural ones such as C?

Usability. Can visualization of code properties with dependence graphs help the programmers fix vulnerable code? Can it help in secure programming education?

Heuristic Ranking. Can we find effective heuristics for ranking of potential security bugs found through analysis?

Model Updates. Will our security property database be fairly static or will it need continuous updates with new flavors of the security properties?

References

[1] K. Ashcraft and D. Engler. Using programmer written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[2] Blexim. Basic integer overflows. Phrack Magazine 60 `http://www.phrack.org/phrack/60/p60-0x0a`, 2002.

[3] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington DC, USA, 2002.

[4] W. Chen, B. Rudiak-Gould, and B. Schwartz. Automatic detection of implicit type cast errors in C. Paper in graduate course, `http://www.cs.berkeley.edu/~wychen/papers/261.ps`, 2002.

[5] B. V. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[6] J. S. Foster, R. Johnson, J. Kodumal, T. Terauchi, U. Shankar, K. Talwar, D. Wagner, A. Aiken, M. Elsman, and C. Harrelson. Cqual: A tool for adding type qualifiers to C. `http://www.cs.umd.edu/~jfoster/cqual/`, 2003.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H.Freeman and Company, 1979.

[8] Grammatech Inc. Codesurfer. `http://www.grammatech.com/products/codesurfer/`.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[10] M. Howard. Reviewing code for integer manipulation vulnerabilities. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode%/html/secure04102003.asp`, April 2003.

[11] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[12] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Helsinki, Finland, 2003.

[13] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proceedings of the Second Workshop on Software Model Checking*, Boulder, Colorado, USA, 2003.

[14] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177—184, Pittsburg, Pennsylvania, 1984.

[15] M. Weber, V. Shah, and C. Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.

[16] M. Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, USA, 1981.

[17] D. A. Wheeler. Secure programming for Linux and Unix HOWTO v3.010. `http://www.dwheeler.com/secure-programs/`, March 2003.

[18] J. Wilander. Modeling and visualizing security properties of code using dependence graphs. In *Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden (to appear)*, Vasteras, Sweden, `http://www.idt.mdh.se/serps-05/`, October 2005.

[19] J. Wilander and M. Kamkar. A comparative study of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002.