02242 — Program Analysis

Course Project

Hanne Riis Nielson

DTU Compute, Technical University of Denmark

September 1, 2014

Course project: Buffer overflows and slicing

We are interested in a program analysis tool that allows us

- to identify potential buffer overflows in our programs, and
- to highlight the slice of the program of interest when fixing the code

The project is in four phases:

- Survey potential existing tools solving this problem and their use
- Design a program analysis module that solves the problem for a small programming languages
- Implement the program analysis module
- Experiment with the program analysis module to get useful insights into the power of the analyses

The four phases

- Survey phase (week 36)
    - Search the web to find information
- Design phase (week 37-43)
    - Exercise 1: reaching definitions analysis and program slicing
    - Exercise 2: detection of signs analysis and interval analysis
    - Exercise 3: algorithms and data structures
- Implementation phase (week 44-46)
    - Implement your solutions to the exercises
    - You may use your favorite programming language
- Experiment phase (week 47-48)
    - Evaluate your implementation on benchmark programs
    - Compare the precision of various analysis/algorthms

Report and exam:

- The final report should report on all four phases
- The report is due on Monday 1st December (week 49)
- The oral exam takes place on 10th, 11th and 15th December

Survey phase

The aim of this part of the project is to find and describe an existing tool that solve (at least part of) the problem: detect potential buffer overflows and highlight the part of the program where the flaw should be found.

Find information about the technology behind the system. How general is the tool and to what extend can it be used to solve (part of) our problem? Is it possible for you to try out the system and run it on some examples?

Please focus on one tool and try to get deeper into that – rather than cover several systems superficially.

You are expected to report on your findings in the final report of the course.

Design, Implementation and Experiment phase

Syntax of the Project Language

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid A[a] \mid a_1\ op_a\ a_2 \mid -a \mid (a) \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1\ op_r\ a_2 \mid b_1\ op_b\ b_2 \mid !b \mid (b) \\
S & ::= & x := a; \mid \texttt{skip;} \mid A[a_1] := a_2; \mid \texttt{read}\ x; \mid \texttt{read}\ A[a]; \mid \texttt{write}\ a; \\
 & \mid & S_1\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} \mid \texttt{while}\ b\ \texttt{do}\ S\ \texttt{od} \\
D & ::= & \texttt{int}\ x; \mid \texttt{int}\ A[n]; \mid \epsilon \mid D_1\ D_2 \\
P & ::= & \texttt{program}\ D\ S\ \texttt{end}
\end{array}
$$

Operators:

$$
\begin{array}{rcll}
op_a & \in & \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/}\} & \texttt{int} * \texttt{int} \to \texttt{int} \\
op_r & \in & \{\texttt{<},\ \texttt{>},\ \texttt{<=},\ \texttt{>=},\ \texttt{=},\ \texttt{!=}\} & \texttt{int} * \texttt{int} \to \texttt{bool} \\
op_b & \in & \{\texttt{\&},\ \texttt{|}\} & \texttt{bool} * \texttt{bool} \to \texttt{bool}
\end{array}
$$

Notation:

$x \in$ **Var** : variable names $\quad n \in$ **Z** : integer constant

$A \in$ **Arr** : array names

More on the Language

- the usual precedence rules for operators apply (as in C, Java, SML, ...)
- the `read` and `write` statements are only used to read integers and write integers
- the `skip` command has no effect
- all variables and arrays must be declared at least once before they are used; in case of redeclarations the rightmost declaration is the final one
- the declaration of an array determines its size; so $A[n]$ will have n elements indexed as $A[0]$, $A[1]$, $\ldots$, $A[n-1]$
- declarations initialise all variables and array entries to 0

Exercise 1: Program Slicing

The idea of *program slicing* is to determine the part of a program that may influence the values computed at a given point of interest; this part of the program is then called a *program slice*.

For example, we might like to slice the program

$$[x := 1]^1; [z := x + 2]^2; [x := 2 * y]^3; [z := x + 2]^4$$

using the statement labelled 4 as the point of interest, and this will result in the program slice

$$[x := 2 * y]^3; [z := x + 2]^4$$

since both the statements labelled 2 and 3 do not affect the computation of the value at label 4. Because this technique helps to focus on the important parts of the program with respect to some criterion, program slicing is used in debugging tools and is also very handy in tools for software validation and maintainance.

Exercise 1: Program Slicing and Reaching Definitions

We shall explore how the Reaching Definitions Analysis can be used to compute program slices.

(a) Using the syntax of the project language write an example program containing at least one while loop and allowing you to demonstrate the effects of program slicing. Provide the program slice according to a point of interest chosen by you.

(b) Modify and extend Table 2.2 in [NNH] to handle the project language.

(c) Construct the flow graph (or program graph) for your program in (a) and construct the data flow equations/constraints using your specification from (b). Compute the least solution to the equations/constraints by hand.

(d) Present an algorithm for calculating a program slice with respect to an arbitrary point of interest. The algorithm has to use the results of a Reaching Definitions Analysis of the program, but may have to use other inputs as well. Apply your algorithm to your example program to recalculate the program slice you determined in (a).

Exercise 2: Buffer Overflow

Array bound checking refers to the problem of determining whether all indices into the arrays of a program are within their declared ranges – and thereby whether or not buffer overflows may arise.

Languages, such as C, which do not have array bound checks are prone to buffer overflows: data may be stored beyond the bounds of the array, thereby overwriting other than the intended memory locations.

This situation can clearly lead to an abnormal termination of the program, but may furthermore be maliciously exploited and therefore compromises the security of the entire system.

We shall consider two analyses

- Detection of Signs Analysis – for checking the lower bounds
- Interval Analysis – for checking lower and upper bounds

Exercise 2: Detection of Signs

(a) Write an example program in the project language containing at least one while loop and with both admissible and inadmissible array references at the lower as well as the upper bounds of the arrays.

(b) Specify a Detection of Signs analysis for the project language as an instance of a monotone framework; argue that it indeed is an instance of a monotone framework. How will you argue that the analysis is correct?

(c) Present an algorithm for array bound checking for the lower bounds using the detection of signs analysis.

(d) Use the specification of the analysis to construct a set of equations (or contraints) for the program considered in (a); present a solution to the equations (constraints).

Exercise 2: Interval Analysis

We shall now consider a more general approach to array bound checking where we use an Interval Analysis.

(a) Present an instance of a monotone framework for an Interval analysis of the project language taking care of lower as well as upper bounds of the arrays; argue that it indeed is an instance of a monotone framework. How will you argue that the analysisi is correct?

(b) Present an algorithm for array bound checking making use of the interval analysis.

(c) Show how the analysis handles the example program you came up with in the previous exercise.

(d) Discuss the precision of the analysis and thereby the buffer overflow checking. How can the precision of your analysis be improved?

Exercise 3: Algorithms and Data Structures

The various analyses implemented in the program analysis module will make use of a number of general data structures and algorithms and in this exercise we shall focus on those.

(a) The parser provides a representation of the abstract syntax tree; specify this data structure in your implementation language.

(b) In the previous exercises you have already decided whether to use flow graphs or program graphs; present the data structure for these graphs and give an algorithm for constructing a flow graph/program graph from an abstract syntax tree.

(c) We shall use a worklist algorithm for computing the smallest solution to a set of data flow equations/constraints. Specify the data structures used for this algorithm; in particular, how will you represent the equations/clauses to be solved so that general monotone frameworks can be handled by the algorithm?

Exercise 3: Advanced Algoritms

For the ambitious!

We study a number of worklist algorithms for solving the data flow equations and the aim of this exercise is to compare their performance in the special case of *Bit Vector Frameworks*.

(a) Implement a version of the worklist algorithm that is parameterised on the representation of the worklist and the operations on it.

(b) Construct instantiations of the algorithm corresponding to two or more of the following: (*i*) Round Robin, (*ii*) LIFO, (*iii*) FIFO, (*iv*) reverse postorder and (*v*) strong components with local reverse postorder.

(c) Design and perform a number of experiments to compare the performance of these algorithms; do your results reflect the theoretical worst-case complexity results?

Important dates

- Monday 20th October: (week 43)
  Deadline for preliminary report presenting the findings for the survey and design phases
  - The reports must be submitted on CampusNet
  - Tuesday 21st October: We shift the reports between the groups
  - Monday 27th October: The groups give oneanother feedback on the reports
- Monday 17th November: (week 47)
  Deadline for submission of benchmarks (on CampusNet)
- Monday 1st December: (week 49)
  Deadline for the final report
  - The reports must be submitted on CampusNet (one for each group)

Preliminary Report: Deadline 20th October 2014

- submit your report using CampusNet (only one report per group)
- remember to include your names and email addresses on the front page

You will give oneanother feedback on the reports on Monday 27th October.

- You will be paired with another group and get their report on 21st October
- Before Monday 27th October at 12.00 please send a short report to
  - the members of the other group and
  - Alessandro and Hanne

  containing

  - 3 points you believe the other group have done very well
  - 3 constructive suggestions where the other group could improve their solutions

Preliminary Plan

Week	Lecture	Course project	Deadlines
36	Introduction	Survey phase	
37	Reaching Definitions	Design phase (ex 1)	
38	Forward and Backwards Analyses	Design phase (ex 1)	
39	Monotone Frameworks	Design phase (ex 2)	
40	Non-distributive Frameworks	Design phase (ex 2)	
41	Equation Solving	Design phase (ex 3)	
42	Autumn vacation		
43	Advanced Algorithms	Design phase (ex 3)	Prelim. report
44	Discussing report	Impl. phase	
45	Types in Program Analysis	Impl. phase	
46	Abstract Interpretations	Impl. phase	
47	Logic in Program Analysis	Exp. phase	Benchmarks
48	Interprocedural Analysis	Exp. phase	
49	End of Course		Final report