1.1 What is program analysis

Program analysis aims to automatically reason about the code and program executions to predict software behaviors.

Program analysis studies the relations of code, paths, inputs and executions:

- code: source code, intermediate code (IR), binary code
- paths: paths on the control flow graphs
- inputs: user inputs to the programs
- executions: a run of a program executable using an input
- traces: a sequence of instructions executed

1.1.1 The subject: different views of a program

A program is a string.
A program is a sequence of tokes.
A program can be organized into an abstract syntax tree (AST). Branches, loops, functions and statements are syntactic structures of a program and can be obtained via syntactical analysis. It is a solved problem via parsing algorithms.

Example 1.1 Source code, AST, parsing

While executing with input, a program produces a set of execution traces.
A program consists of a set of program states at each program point of an execution trace. A program state consists of the values of all the variables at a particular program point. A program property is a set of facts regarding program states hold for partial or all executions.

The hard problem is to determine the semantics of programs — any predictions regarding the values produced while executing the program.

Example 1.2 Buffer overflow detection
1.1.2 Program analysis as problem reduction

The software behaviors we care such as bugs, vulnerabilities, performance can be translated to some kind of program properties. The translation of security, compiler optimization or software engineering interests to program properties is the job of security, compiler or software engineering experts. The techniques of traversing code and program executions to determine program properties are the program analysis techniques.

- bug: a mistake in the code
- fault: a violation of the program property
- failure: deviation from expected dynamic behaviors, the symptoms include crashing, incorrect answers, hang ...
- vulnerability: a bug that can be exploited

Example 1.3 Buffer overflow detection

A naive approach is to enumerate all the possible program states and determine if the property holds for these program states. However, the state space of a program can be too large to be enumerated.

The two key ideas behind program analysis is to develop either abstraction via static analysis—merging states based on their common property/representing a set of states using their common property, or sampling via dynamic analysis—collecting partial states to determine program properties and understand the software behavior.

On the path of fully determining the semantics of a program, we develop the concepts of control flow, data flow, value flow, slice, dice, chop and information flow to represent a partial view or a hint of program states that are useful for various of applications. Typically these concepts build the bridge between code and execution, and are useful properties that can obtain from the code to infer potentially dynamically behaviors.

Example 1.4 Determining control flow of a program means ordering the statement sequences and obtain the potential feasible execution paths.

Example approaches of handling large state space:

Definition 1.5 Abstract interpretation is a theory of abstracting values to abstract domains and the completeness and soundness of such abstraction – The interpretation of the programs semantics with the abstract values is an abstract interpretation. (by Cousot couple) Concrete semantics can be defined by a function that maps an expression to a value. Abstract semantics can be defined by a function that maps an expression to an abstract value.

Definition 1.6 Symbolic execution Concolic execution

1.1.3 Applications of Program Analysis

- From application domains to program analysis:
  - Compiler optimizations
  - Bug Findings
Lecture 1: An Overview

- Generate exploits
- Test input generation
- Infer program specifications
- Debugging
- Repair
- ...

- From program analysis to SAT problems, graph reachability
  - Theorem prover
  - Boolean Satisfiability (SAT) solver
  - Satisfiability Modulo Theories (SMT) solver
  - ...

1.1.4 Static analysis

A static analysis of a program is a sound, finite, and approximate calculation of the program’s executions [david]

Safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program on a computer [book]

- reusing available results
- moving loop invariant computation out of the loop
- avoiding superfluous computations (results not needed)

safe: does not produce incorrect compilation optimization, the properties represent a superset of the computation paths

Static analysis can be performed on either source or binary code. It analyzes the code without running a program. It offers the techniques for predicting statically at compile-time the set of configurations or behaviors dynamically at run-time.

It is a way of ”automating code inspection”

How to perform it:

- Convert source code to some sort of program representation (graphs or trees)
- Traverse the representation to collect information (static analysis algorithm: how to traverse it?)
- Determine if the properties hold based on computations of the information

1.1.4.1 Theoretical complexity of static analysis

Static analysis is undecidable. Thus,

- The static analysis algorithm is an approximation.
• The results of static analysis tools are imprecise in general.

Reference: undecidability of static analysis – reduction of halting problems to an alias problem

1.1.4.2 Implementing static analysis

• Frameworks: Soot, LLVM, Phoenix

• Program representations and IR: AST, CFG, Callgraphs

• Symbol table: variable types

• Program analysis algorithms: points-to, loop analysis, value range information, definition and use of variables

• Other supporting facilities: optimizing program analysis information, storing in-progress program analysis, mapping between source and IR

• Standalone tools: codesurfer, atlas, findbugs ...

1.1.5 Dynamic analysis

Learning from executions:

• Run code/Execution (what is the input)

• Profiling and monitoring (how to get useful information while minimizing the overhead)

• Data analysis (offline or online):
  – Frequency, coverage and correlation of program entities
  – Build graph representation for the data
  – ...

1.1.6 Hybrid analysis

The two analyses can be performed on

• Different parts of code [Le:ICSE13]

• Different parts of executions [Godefroid:PLDI05]

• Different passes [Csallner:ICSE05]
<table>
<thead>
<tr>
<th>Requirement</th>
<th>static</th>
<th>dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
<td>code (compile, build, source, binary)</td>
<td>executable, input</td>
</tr>
<tr>
<td>Advantages</td>
<td>apply early in lifecycle</td>
<td>easy for debugging</td>
</tr>
<tr>
<td>Disadvantages</td>
<td>false positives</td>
<td>false negatives</td>
</tr>
</tbody>
</table>

1.1.7 Program analysis and other fields of computer science

- **Programming Languages**
  - Define a language, and then design the analysis for the programs written in the language
  - Examples of language dependent features: recursion and level of pointer dereferences

- **Compilers**
  - Code optimization: first perform program analysis to determine the optimization opportunities; then transform the code
  - Semantic analysis: check if the properties of correctness holds

- **Software Engineering**
  - Program analysis tools are software that analyzes other software (like a compiler), they also require software engineering skills to build

- **Software Engineering, Security, Systems, Graphics and Robotics**
  - Problems in the domains such as test input generation, malware analysis and performance tuning need information from software.

- **Theory**: An important part of the program analysis research is to develop algorithms and analyze their complexity, correctness and precision.

- **Machine Learning**: Program analysis generates data and thus we can apply machine learning to process these data and summarize the program properties.

- **Architecture**: We can develop architecture support to get information from executions to facilitate program analysis, e.g., hardware counter is useful to collect branch information for the execution paths

1.1.8 Future of program analysis

- Algorithms to enable more precise and scalable analysis
- Algorithms to compute new type of program information (e.g., probabilistic symbolic execution)
- Algorithms to address new types of programming languages and paradigms (e.g., javascripts, web applications, Android)
- Problem reduction: discover new types of problems (e.g., program synthesis)
- ...
References