Software Security 05

Sven Schäge

Program Verification and Data Flow Analysis

Program Verification

- In general, program verification tries to establish correspondence between what the program is doing and what it should be doing
- **Specifications**, examples:
  - Informal documents
  - Other programs with the same functionality – compare two versions of the same program
  - Simple program or more complicated one that is more efficient
- Program verification often involves manipulation of arbitrary programs
- Halting problem and Rice‘s theorem say that tasks which involve arbitrary programs cannot in certain cases be performed by algorithms that always terminate.
  - Program verification in general form undecidable

Program Verification

- Task to verify some correctness statement (establish correctness of program)
- Correctness (with respect to a specification) means that the program behaves as specified
- Program (positively) **verified** with respect to property=> Program is **guaranteed** to have this property
  - Properties usually have a narrow specification
- Functional correctness
  - Program does what specification says
  - Specification may be wrong – does not describe what user had in mind
  - Needs validation (i.e. is the specification realistic/useful/good)!
- Absence of non-functional errors
  - Absence of run-time errors
    - Division by zero
    - Out-of-bounds arithmetic, over or underflows
  - Satisfaction of space constraints
    - Program does not exceed space allocated to it
  - Satisfaction of timing constraints
    - Not longer than deadline permits

Program Verification

- Program verification tries to **approximate** all the possible behaviours
  - As a result we may not be able to establish all properties that are of interest
  - **This does not mean that specific problems of program verification are hopeless!**
- Goal
  - Programs whose correctness can be argued **in simple ways** can be checked
  - Programs that are so tricky that nobody knows why they work may be beyond the reach of automated verification tools
  - Similar to math (Gödel‘s theorem)
- Various methodological approaches of constructing tools to perform program verification

Program Verification

- Acceptable
  - **Desire to establish correctness is compatible with desire to keep program simple enough to reason about them**
  - **Attempt to engineer systems that we can reason about, simple enough so that we can verify them independently using automated tools**
  - **Human written code is not arbitrary**
  - Higher level languages where certain concepts can be expressed in a more natural way make it easier to verify programs (than machine code)
    - Functional programming is a nice match, closer to mathematical description
      => building tools that use mathematical reasoning to prove program correctness is more effective
    - More cost effective for programs that we can express naturally using functional code

Realistic Expectations

- Don‘t expect a verification method that is
  - Powerful – able to prove non-trivial properties
  - Automatic
  - Sound – never proves the property if it does not hold
  - Complete – always proves the property if it holds
- May give up automatic analysis
  - Complex user interaction
- In the context of abstract interpretation, Completeness and Soundness have a specific meaning:
- **Completeness:** A static analysis is complete if it can discover all true properties of the program. In other words, it finds all possible issues and accurately characterizes all aspects of program behavior.
- **Soundness:** A static analysis is sound if it never produces false positives (no false alarms). In the context of abstract interpretation, this means that if the analysis declares a certain property to be true, then that property is indeed true in the actual program.
- Static Analysis usually gives up completeness but guarantees soundness. All found properties indeed hold.
  - Not every property true of the original system is true of the abstract system
  - Every property true of the abstract system can be mapped to a true property of the original system
  - Never gives incorrect results but may output maybe
- Achieving soundness requires reasoning about all executions of a program (usually an infinite number). This is typically done by making suitable abstractions of the executions to make the analysis terminate.
- May keep soundness and completeness only in certain simple cases
  - Academic environments

Conservative Approximations

- Ideally, the approximations we use are conservative (or safe):
  - All errors lean to the same side, which is determined by our intended application, e.g. all approximations are either only over-approximations or under-approximations
- Example, approximating the memory usage of programs is conservative if
  - the estimates of how much memory is used are never lower than what is actually possible when the programs are executed.
- Conservative approximations are closely related to the concept of soundness and completeness of program analyzers.
- If all approximations are over-approximations we may preserve soundness.
  - Only output “No division by 0!” if this is always true
- If all approximations are under-approximations we may preserve completeness.
  - Only output “Division by 0!” if always true
- It can also be thought of as dealing with uncertainty in an organized manner
  - Analyzers that output No vs maybe = No vs (Yes or Don’t Know)
  - Analyzers that output Yes vs maybe = Yes vs (No or Don’t Know)

Approaches to Program Verification 1

- Model Checking
  - Essentially exhaustive search over all states of some model where groups of states are nicely summed up (generalization or symmetry)
  - Exploring all states and transitions and consider whole groups of states in a single operation and reduce computing times
  - + Often fully automatic
  - - does not in general scale to large systems (supports only a few hundred bits of state)
  - Often artificial bounds are required to deal with complexity, e.g. bounds on the number of loops
    - Not sound anymore

Approaches to Program Verification 2

- Deductive Verification
  - Generate from program and specification a collection of proof obligations
    - If proof obligations are true then conformance of program to specification shown
  - Proof obligations are tackled
    - with proof assistants like HOL, ACL2, Isabelle, Coq or PVS or
    - automatic theorem provers including in particular satisfiability modulo theories (SMT) solvers. SMT stands for satisfiability modulo theories, a generalization of the (Boolean) SAT problem.
  - This approach has the disadvantage that it may require the user to understand in detail why the system works correctly, and to convey this information to the verification system,
    - either in the form of a sequence of theorems to be proved
    - or in the form of specifications (invariants, preconditions, postconditions) of system components (e.g. functions or procedures) and perhaps subcomponents (such as loops or data structures).

Approaches to Program Verification 3

- Abstract interpretation
  - Approximate possible behavior by replacing sets of possible values with some more abstract representations of this set.
    - Example: Replace set of integers with some intervals which is an approximation of the set of integers
    - This reduces complexity
      - More manageable
      - Less likely to “hit” fundamental impossibility results
    - Idea (in the context of software security): if bug is absent in (strictly) worse (approximated) program, then it must be absent in original program too
      - Approximation should preserve this property
    - Abstract Control Flow
  - Next, transform the computation on exact sets of program states into some **fixed-point computation** that in case of appropriate approximations can be solved in finite amount of time
  - Idea for finite termination
    - Sets grow **monotonously.** This is guaranteed if approximations are organized as a (mathematical) lattice, a partial order where unions and intersections of all elements exist.
      - (CodeIsDead, CodeNotDead, No Information Yet/Uninitialized, Don’t Know - Can be Both)
    - The number of all possible configurations of all the sets is finite. (We only really need that the lattice has finite height.)

Abstract Interpretation

- Establish more complicated properties:
  - Take into account program and property -> **property directed approach**
- Could go both ways, e.g. from properties to see what are the sufficient conditions to establish these properties
- Verification can be a process that establishes easier properties sooner – some properties will never succeed in establishing
- Properties can be refined. Assume we want to exclude division by zero. Consider a program point with x/y.
  We can now ask several questions with ranging preciseness. All of them give useful information on the question.
  - Is there any integer assignment that reaches the statement x/y?
  - Is there any assignment to y that reaches the statement x/y?
  - Is there any assignment to zero that reaches x/y?
  - Is there any assignment of y to zero that reaches x/y?
- Practically more success on typical programs and typical properties
- - Needs educated users
  - But less so than model checking and deductive verification
- For imperative programs like those written in TinyLang that do not use dynamic allocation or recursion, abstract interpretation is a successful technique for proving the absence of memory management vulnerabilities automatically and efficiently, e.g. buffer overflows.

Example - Non-functional Property: No Division by Zero

- Easy (trivial)
  - x/const
  - If const=0 bad
  - Otherwise good
- Non-trivial
  - x/y
  - Need to know that in no execution of program, y will be set to 0!
  - For all inputs and thus for all program paths!
- Strategy
  - Know all values that y may take on when this expression is reached!

} Can efficiently be addressed via Dataflow Analysis!

- Even Harder
  - x/Equations.LHS[i]
  - Need to know that in no execution of program, Equations.LHS[i] (referenced to by a pointer) will be set to 0!
  - For all inputs and thus for all program paths!

} involves pointers and dynamic allocation

- Divisions by zero can be security relevant:
  - In c++, division by zero on integers is undefined behavior. One of the rules about the C++ language is that the compiler may assume that undefined behavior will never occur, and thus is allowed to do anything if it does.
  - Often an exception handler will be invoked to handle the division by zero. In general, attackers know that exception handlers are not as well-tested as regular code flows.