Finding Bugs

CIS 410/510 Program Analysis and Transformation

Last time

- Dependence analysis
- Vectorization

Today:
- Bug finding through static analysis
  - Approaches and example tools
  - Quality assurance
Problem

- **What is a bug?**
  - A path in the code that causes a run-time exception
  - A path through the code that causes incorrect results

- **Issues**
  - exponentially many paths
  - cannot statically determine the path a program will take
  - “Program testing can be used to find the presence of bugs, but never to show their absence.” [Dijkstra 1972]

- **Undecidability**
  - soundness (miss no errors) and completeness (no false positives) together are undecidable
  - confusion in literature: which is which?
    - every reported error is genuine (no false positives)
    - if the program has any errors then the checker will report some error (no false negatives)

Motivation for the automatic detection of bugs

- **Time spent in program maintenance**
  - most software engineers spend the majority of their time doing maintenance
  - most time spent doing maintenance is time spent debugging

- **Costs due to bugs that allow security exploits (e.g., through Jan 31, 2003, approximations by CNET News.com):**
  - Slammer (950 million)
  - Code red (2.6 billion productivity loss)
  - LoveLetter (8.8 billion)
  - Klez virus (9.0 billion)
  - Other famous bugs
Approaches to finding bugs

- Approaches
  - strengthening the type system
  - static analysis to detect bug patterns
  - automated theorem proving
  - dynamic analysis
    - catch errors before they occur
    - find the cause for failures after the fact

- Evaluating the different approaches
  - how many false positives?
  - how many false negatives?
  - extend of user intervention or ease of use
  - efficiency of approach

Example bugs

- null dereference
  ```java
  if (p == null) {
      p->open();
  }
  ```

- array bounds error
  ```java
  int a[20];
  a[20] = ...;
  ```

- untrusted access
  - format string vulnerability
    ```c
    fgets(buffer, size, file);
    printf(buffer);
    ```
Type qualifiers [Shankar et al. 2001]

- **Idea**
  - Add tainted and untainted types to library function signatures
    
    ```c
    fgets(\texttt{tainted} char *buffer, int size, FILE *f);
    printf(\texttt{untainted} char *format, ...);
    ```
  - Use type constraint solver to find errors
    - errors are type mismatches

- **Issues**
  - What is the type of `strdup()`?
  - What happens when the values of strings change?

Static analysis examples

- **FindBugs**
  - project at University of Maryland for finding bugs in Java
  - they observe that bugs found in student programs are also found in production code
  - implementation approach
    1. Think of the simplest technique that would find occurrences of the bug
    2. Implement it
    3. Apply it to real software. Hopefully find some real bugs. Will probably produce some false warnings
    4. Add heuristics to reduce percentage of false warnings.
  
  Their experience: new detectors can usually be implemented quickly (somewhere between a few minutes and a few days). Often, detectors find more bugs than you would expect
Analysis for bug detectors in FindBugs

- Kind of analysis in implementing detectors
  - Examination of method names, signatures, class hierarchy
  - Linear scan of bytecode instructions (Java) using a state machine
  - Method control flow graphs, dataflow analysis
  - No interprocedural flow analysis or sophisticated heap analysis

How FindBugs handles the example bugs

- Null pointer dereferences
  - found 37 in rt.jar 1.5-b59, 55 in eclipse-3.0

- Array bounds checking
  - not an issue in Java

- Untrusted code
  - Can static fields (or the objects they refer to) be modified by untrusted code?
    - public, non-final static fields
    - public static fields pointing to an array
  - Warnings: 254 in rt.jar 1.5-b59, 967 in eclipse-3.0
Real-world bug finding through static analysis

- Introduction to static analysis for assurance (John Rushby)
- Example tools:
  - C/C++ -- Clang static analyzer
  - Java: ESC/Java, Find Bugs, PMD, QJ Pro,

Automated theorem proving

- What is a theorem prover?
  - A computer program that can generate and check mathematical theorems
  - Theorems are expressed in some mathematical logic, such as propositional logic, predicate logic, first-order logic, ...
    - Extra: A nice summary of different logics:
      [http://www.personal.psu.edu/t20/notes/logic.pdf](http://www.personal.psu.edu/t20/notes/logic.pdf)

- Why theorem proving?
  - If you can prove all the lemmas, then you can be sure that the program meets its specification
  - There is no chance that a bug-finding tool’s heuristics just weren’t smart enough to find a bug

- Many different provers available: Isabelle/HOL, TWELF, Coq, Metamath, Nuprl
Theorem prover overview

- Provided by user (not to scale)
  - Math (in some logic)
- Provided by developer (not to scale)
  - Theorem Prover

Differences between provers

- The logic the prover uses
  - Isabelle/HOL: Higher Order Logic (HOL)
  - TWELF: Logical Framework (LF)
  - Coq: Calculus of (Co-)Inductive Constructions (CiC)
  - etc.

- Some logics are more powerful (can express and prove more theorems) than others, e.g.,
  - Propositional Logic is usually the weakest
  - CiC is more powerful than HOL

- More powerful logics can be harder to use
Differences between provers

- The task a prover handles
  - All provers can check theorems in their logic
  - Automated proof generation is much harder

- Different provers have different tradeoffs between degree of automation and the power of the logic they handle
  - The more powerful the logic, the less automatic generation of proofs
  - Many automated theorem provers are really more automated theorem checkers.

Proof checking vs. proof generation

- A formal proof is a list of formulas each of which is justified by an axiom or an inference rule applied to earlier formulas

<table>
<thead>
<tr>
<th>Formulas</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Axiom</td>
</tr>
<tr>
<td>F2</td>
<td>Rule 3 and F1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Theorem</td>
<td></td>
</tr>
</tbody>
</table>

- Formal proofs are very easy to check mechanically
  - Just make sure the justifications are applied correctly

- However, proof generation is harder – have to generate a list of formulas, each of which has a valid justification, and where the last formula is the desired theorem
Differences between provers

- Provers are software packages, so...
- Some provers have
  - more support if there is trouble
  - bigger user base
  - more frequent new versions
  - tool support
  - better library support (e.g., built-in definitions of the real numbers, etc.)
- Can have bugs!

What does the user provide?

- Depends on the prover
- All provers: statement of theorem expressed in the logic of the system
- For some provers, this is enough – all you do is give the desired theorem and push “Go”
- Fully automatic provers can’t prove nearly as many theorems as “semiautomatic” provers
  - User must provide some kind of hints that help the prover (often provided in the same file as the theorem)
  - Least useful hint: “A proof exists – search forever until you find it”
  - Most useful hint: “Here is the proof: ...”
Intermediate hints

- Most provers take a middle path and require hints between the two extremes
  - Statement of key lemmas (useful intermediate results)
  - Proof outline (how the lemmas connect)
  - Key idea in proof ("prove by induction on n")
  - Proof script (list of medium-sized steps in the proof)

- One advantage of providing hints is that if the theorem is not provable, the prover can provide better reporting as to why the proof failed
  - Most error reporting is still difficult (much worse than a typical compiler error report)

Theorem prover overview

- Provided by user (not to scale)
- Provided by developer (not to scale)
Libraries

- What do provers have to prove (each time)?

- Many theorems share commonly used definitions and lemmas
  - Natural numbers
    - definition (zero & successor)
    - facts about naturals \((a + b = b + a)\) and their proofs
  - Integers
    - definition (naturals & negative naturals & zero)
    - facts about integers (e.g., \(a + (-a) = 0\)) and their proofs
  - etc.

Theorem prover overview

- Provided by user (not to scale)
  - Statement of Theorem
    - Hints
- Provided by developer (not to scale)
  - Theorem Library
  - Theorem Prover
Why are theorem provers used?

- Very high assurance due to mechanical checking
  - Checkers are very thorough: don’t get tired (don’t need coffee), don’t get bored, don’t make mistakes (unless buggy, of course)
  - If anything, the problem may be the opposite – trying to convince a checker that a true thing is really true can be frustrating
  - Used in areas where correctness is critical: aerospace, defense, etc.

- When possible, automatic proof generation can significantly improve program development
  - Earlier detection of bugs
  - Better code/design coverage than testing
  - Frequently tools can locate errors faster than human debugging

When to use provers

- Better at some kinds of tasks than others

- Best: proving behavior of real programs
  - Static analysis tools: buffer overrun analysis, safety property analysis
  - Safety of web applications
  - Type inference & checkers (e.g., ML, Java, C#)

- Bad:
  - Cryptography: often we rely on guesses (P = NP?)
  - Pure math: Tools and libraries not practical yet
  - Design: how to prove that one UI is better than another?
  - Performance
Disadvantages of automated theorem proving

- For proof generation:
  - Only useful for certain kinds of “simple” problems
  - Tools are frequently very difficult to develop
  - Often can have very bad worst-case running time (e.g., Hindley-Milner type inference is $O(2^n)$); sometimes average running time is much better

- For proof checking:
  - Developing the hints/proof by hand can be very labor-intensive; learning curve to use provers can be steep
  - It can be very difficult to formalize correctness
    - “correct” OS?
    - “correct” web browser?
    - “correct” compiler (an early example: McCarthy and Painter, Correctness of a Compiler for Arithmetic Expressions, 1967)
  - Programs are only verified when all surrounding elements are verified

To end on a positive note: Advantages

- Provers are fun to use! A bit like writing software in a scripting language
  "Building such scripts is surprisingly addictive, in a videogame kind of way..." - Xavier Leroy

- Never having to worry about bugs in the finished product
Example: SAL at Microsoft
- Standard Annotation Language for interface pre- and post-conditions
- focus is on buffer overruns and pointer usage
- SALinfer is a tool that determines specifications automatically

SAL example

// Requirements of foo’s callers: must pass a buffer that is len elements long:
void foo(pre_elementCount(len) int *buf, int len) {
  // Assumption made by foo: buf is count elements long

  ...Local checker: do the assumptions imply the requirements?

  // Requirement on foo: argument buf is len*4 bytes long
  memset(buf, 0, len*sizeof(int));
}

// Requirement on memset’s callers: must pass a buffer that is len bytes long
int *memset(pre_byteCount(len) void *dest, int c, size_t len);
Dynamic analysis

  - adds run-time checks to C programs for catching memory safety errors
  - requires user annotations
  - the only thing that happens statically is figuring out what special type a pointer should be, want fastest possible type that still can catch any possible dynamic errors
  - around 15-50 times faster than Purify

How Ccured handles the example bugs

- New pointer types
  - SAFE pointer: on use does a null pointer check
  - SEQ pointer: on use does a null pointer check and an array bounds check
  - DYN pointer: on use does a null pointer check, a bounds check, and a type check (checks type casts)

- Null pointer dereference
  - use SAFE pointer

- Array bounds
  - use SEQ pointer

- Untrusted access
  - doesn’t handle this, focus on handling memory issues
Remaining issues

- Evaluation of new techniques is tedious
  - must have a human determine if problem reported is an actual bug
  - getting developers to fix the bug is another battle
  - how can we determine if one bug detection system is better than another?
    - might analyze different languages
    - experiments performed on different benchmarks (different versions of the software make a different benchmark)
    - approach: people are starting to put together bug benchmarks

- Static analysis
  - whole program vs partial program analysis
  - quality of alias analysis affects quality number of false positives

Concepts

- Approaches to bug detection
  - augmenting the type system
  - static analysis
  - automated theorem proving
  - dynamic analysis

- Comparing bug detection techniques is tricky
  - what is considered a real bug?
  - how can we compare false positives with false negatives? how can we determine all?