Challenges and New Directions in Program Analysis

Monica Lam
Stanford University
Software Reliability

A pressing problem in computer science

Buffer overruns

- Cause of most security breaches
  - Blaster, Slammer, Code red, nimda, ..., Internet Worm, 1988
- Cannot audit programs for a simple and important property
Testing Costs

“50% of my company employees are testers,
and
the rest spends 50% of their time testing!”

Bill Gates,
in 1995
Program Analysis

Improves software reliability & productivity

- Optimizations
  - High-level programming
- Parallelization
  - Sequential programming
Can We Find Errors Automatically?

- What is the correct behavior?
  - Optimizations / parallelization generate equivalent program
  - Formal specifications do not exist

- Needs precision
Software Governed by Design Rules

- No null pointer dereferences
- Languages
  - No buffer overruns
  - No memory leaks
- Application-specific
  - Do not save unencrypted passwords
- Classes, methods, procedures
- Statements, variables
Design Rules

- Used by programmers
  - Correctness, security, behavior goals
  - Conventions to keep programming simple
- Not written down, not checked
- Many violations in existing code
- Premise: Improve software quality by
  - Capturing the design rules
  - Enforcing the design rules
Capture the Design Rules

- **High-level design rules**
  - More succinct than assertions, pre- and post-conditions, loop invariants
  - Linguistic support to express them

- **Low-level design rules**
  - Too tedious to write
  - Changes with the code
  - Automatically infer them from the code
  - Same rules obeyed over & over
  - Errors = inconsistencies, anomalies
Example: Memory Leaks

- A hard problem
  - Requires whole-program understanding
  - Previous tools find easy cases
  - Finding majority of leaks requires analyzing pointers store in the heap

- Tracking all pointers to objects is hard
  - for analysis or ... programmers

- Programmers use meta rules to keep sane
Meta Rules: Object Ownership

- Every object has one and only one owning pointer at any one time.
  
  \[
  p = \text{new Object;}
  \]
  
  \[
  q = p;
  \]
  
  \[
  \text{delete } q;
  \]

- Conservation of ownership
Object Design Rule

- A member field either **always** or **never** owns its object at public method boundaries

```cpp
class Employee {
    salary-info * s;  // owner
    dept * d;          // not an owner

    ~Employee() {
        delete s;
    }
}
```
Clouseau: Memory Leak Detector

- [Heine & Lam, PLDI 03]

- Automatically infers
  - Class field member ownerships
  - Class method ownership signatures

- Important to identify the source of errors
  - Satisfy high-confidence constraints first
  - Errors = Unsatisfied high-confidence constraints
A Practical Tool

- Case study:
  - 125K lines commercial program in C
  - 50 lines of specification to identify containers
  - Root-caused 82% of dynamic leaks
Provenance

- Where a value comes from?
- Example: format string vulnerability
  Does there exist an $s$ such that
  
  $s = \text{fgets}(...) :$
  
  $...$
  
  $\text{printf} (s);$  

- Requires tracking pointers carefully
Challenge #2: Precision in Analysis

Data
- Local
- Arrays
- Pointers

Control
- flow
- interprocedural
- path

Precision
- Optimizations
- Parallelization
- Verification
Context-sensitive, path-sensitive pointer alias analysis?

Coming soon ...
Programs Don't Grow Exponentially!

Size of Microsoft Windows

Year

Million Lines of Code


Size of Microsoft Windows
New Paradigm

- Old approach
  - Find cheapest way to solve each problem
  - Much time spent on heuristics

- New approach to solving complex problems
  - Take advantage of machine cycles
  - Build upon general abstractions
  - Leverage and build general tools
Examples

- Data dependence analysis
- Loop transformations
- Pointer alias analysis
Data Dependence Analysis

- Formulated by Kuck et al.[72], Lamport[74]
- Affine access functions and loop bounds
  - $A[I,I], A[2i+3j]$
- Two affine accesses have a dependence if there exist two instances that access the same location
- Equivalent to integer programming, an NP-complete problem
Solving Common Data Dependences

- Designed for common subsets

- **Cliff effect:**
  create new tests to handle new subsets
  - GCD [Banerjee 76]
  - Banerjee’s test [Banerjee 79]
  - I-Test [Kong et al. 90]
  - Lambda test [Li et al. 90]
  - Delta test [Goff et al. 91]
  - Power test [Wolfe et al. 92]
Data Dependence

- as integer programming
- SUIF compiler [Maydan et al. 91]
- Uses Fourier-Motzkin elimination
- Optimize for simple integer programs
Data Dependence

- as Presburger arithmetic
- Omega [Pugh et al. 92, 94]
- First-order logic of natural numbers with addition
- Lower bound complexity: $O(2^n)$
- Lasting contribution
  - Communication optimization
  - Code generation
  - Many non-compiler applications
Loop Transforms

- Improve data locality and parallelism
- Useful transformations for loops with affine accesses
  - Fusion, fission
  - Interchange, skewing, reversal
  - Scaling, re-indexing
  - Statement reordering
  - Blocking
Many Proposed Heuristics

- **Abstractions:**
  - Distance vectors, direction vectors
- **Pairwise loop transforms**
- **Unimodular transforms for perfect loops**
Affine Partitioning

- [Lim & Lam, POPL 97]
- Parallelization problem is extremely simple if we start with first principles
- Arbitrary loop nesting
- No intermediate conservative abstractions: dependence vectors
Synchronization-Free Parallelism

∀ \( i_1, i_2 \) \( B_1 i_1 \geq 0, \ B_2 i_2 \geq 0 \)

\[ F_1 (i_1) = F_2 (i_k) \rightarrow C_1 (i_1) = C_2 (i_2) \]

- maximize rank of \( C_1, C_2 \)
Pipelined Parallelism

∀ i₁, i₂ \ B₁i₁ ≥ 0, B₂i₂ ≥ 0 \ i₁ ≤ i₂

F₁(i₁) = F₂(iₖ) → T₁(i₁) ≤ T₂(i₂)

maximize rank of T₁, T₂
Algorithm

- Solution provided by Farkas’s Lemma (1901)
- Simple mathematical algorithm

- Canonical form: coarest-granularity of parallelism
- Enables array contraction & blocking across arbitrary loop nests.
Solve complex problems
by building general abstractions
Many complex context-sensitive, flow-sensitive analysis

- K-limiting [Chase et al]
- Assumed aliases [Ryder et al]
- Invocation graph [Hendren et al]
- Partial transfer functions [Wilson & Lam]
Context-Insensitive Pointers

- Equality-Based, 75K LOC [Steensgaard 96]
- Inclusion-Based
  - Complexity $O(n^3)$ [Andersen 94]
  - Partial online cycle elimination, 60K LOC [Fahndrich et al. 98]
  - Pre-transitive form, 1.3M LOC [Heintze et al. 00]
  - BDD formulation [Berndl et al. 03]
Binary Decision Diagrams

- Developed for model checking
- Represent boolean functions compactly
- Highly optimized
- Pointer alias analysis using BDD is extremely simple!
  - No application-specific optimizations!
BDD Points-to Analysis

Represent program as relations in BDD

- [new] $P_0$: $v = \text{new} \ cl$

- [copy] $C$: $v_d = v_s$

- [store] $S$: $v_d.f = v_s$

- [load] $L$: $v_d = v_s.f$

points-to includes
Inference Rules

\[ P = P_0 \cup \mathcal{C} \otimes_{v_s} \rho_{v_s/v_d}(P) \cup \mathcal{L} \otimes_{v_s} \rho_{v_s/v_d}(P) \otimes h_d F \]

\[ F = P \otimes_{v_d} S \otimes_{v_s} \rho_{v_s/v_d, h_s/h_d}(P) \]
Context-Sensitive Pointer Analysis

- Whaley & Lam
- Using BDDs
Number of Contexts

- Context: call-string
  (recursive functions simplified)
Context-Sensitive Points-to Set

![Graph showing the number of contexts versus minutes. The x-axis represents minutes ranging from 0.1 to 10, while the y-axis represents the number of contexts ranging from $10^1$ to $10^{16}$. The graph displays a trend where the number of contexts increases as the time increases, with a few outliers.](image-url)
Promising Results

- Build context-sensitive analysis on top of BDD abstractions
- Closer to answering user queries
Conclusions

- Use program analysis to find bugs automatically
- Specs = High-level design rules + Low-level inference
- Deep program analysis
  - Leverage Moore’s Law and simple, general abstractions