Review, Summary and Future Studies

Wei Le

Iowa State University

April 25, 2018
Outline

Important Concepts in Program Analysis
Program Representation
Control Flow Analysis
Program Dependence and Slicing
Dataflow Analysis
Abstract Interpretation
Points-To and Alias Analysis
Symbolic Execution
Dynamic Analysis
General Applications of Program Analysis
Implementation and Experimentation
Program Analysis and Program Properties

Program Analysis

- Get information from the code or execution, understand the code, predict program behaviors
- Determine properties for a program: what conditions hold for some or all executions

Safety: something never happens (given finite steps of an execution, we can determine the violation of the property), e.g., buffer overflow

Liveness: something eventually will happen, e.g., memory leak
Program Analysis and Program Properties

Program Analysis

- Get information from the code or execution, understand the code, predict program behaviors
- Determine **properties** for a program: what conditions hold for some or all executions
  - **Safety**: something never happens (given finite steps of an execution, we can determine the violation of the property), e.g., buffer overflow
  - **Liveness**: something eventually will happen, e.g., memory leak
Theoretical Complexity of Program Analysis [16, 22, 7]

**Problem:**
Points-to analysis: given a language (define the types of variables and statements), determine at each program point, which set of memory locations a pointer points to

**Factors that Impact Complexity:**
- Restrictions on the languages: variables (scalar or structure), dynamic memory (allocate memory at run-time) — infinite configurations of the space, procedures (single or multiple), level of pointer dereferences, typing
- Precision and types of the program analysis

**Soundness and completeness**
Input: program
Algorithm: program analysis
Output: set of program facts that satisfy the properties
- Sound: solution set $\supseteq$ correct set
- Complete: every answer in the solution set is the member of the correct set
Dimension of Precision for Program Analysis

- **Inter-, intra-procedural program analysis**: do we collect information across the procedure?
- **Path-sensitive, path-insensitive**: do we merge information from different paths?
- **Flow-sensitive, flow-insensitive**: do we consider the order of the program statements?
- **Context-sensitive, context-insensitive**: do we analyze a procedure by distinguishing its calling context?
- **Object-sensitive, object-insensitive**: do we distinguish which object a variable binds?
Program Representation - for Source Code [3]

- Abstract syntax tree (AST)
- Control flow graph (CFG)
- Call graph
- Interprocedural control flow graph (ICFG)
- Program dependency graph (PDG)
- System dependence graph (SDG)
- Points-to graph
- Single static assignment (SSA)
Program Representation - for Dynamic Information

- **Control flow trace**
- Dynamic dependence graph (DDG)
- Whole execution trace (WET)
Control Flow Analysis [3]

- Construct control flow graphs: determine the execution order of the statements and represent them in a graph
- Explore parallelism: determine the order of the statements
- Determine dominators and post-dominators
  - Dominators: all the paths between the entry and the target node will go through its dominators
  - Post-dominators: all the paths between the target node and the exit will go through its post-dominators
- Perform loop analysis
  - Loop nest level, loop head
  - Trip count
  - Loop invariant
  - Loop induction variable
Dependence between two statements (1981):

- **Control dependence:**

  \[
  \text{Definition} \quad \text{Let } G \text{ be a control flow graph. Let } X \text{ and } Y \text{ be nodes in } G. \ Y \text{ is control dependent on } X \text{ iff}
  \]
  \[
  (1) \text{ there exists a directed path } P \text{ from } X \text{ to } Y \text{ with any } Z \text{ in } P \text{ (excluding } X \text{ and } Y) \text{ post-dominated by } Y \text{ and}
  \]
  \[
  (2) \ X \text{ is not post-dominated by } Y.
  \]

  If \( Y \) is control dependent on \( X \) then \( X \) must have two exits. Following one of the exits from \( X \) always results in \( Y \) being executed, while taking the other exit may result in \( Y \) not being executed. Condition 1 can be satisfied by a path consisting

- **Data dependence:** data dependence between two statement nodes exists if a definition of a variable at one statement might reach the usage of the same variable at another statement.
Program Slicing and Chopping [28, 13, 23]

- Slicing criteria: variable v, statement s
- **Program slicing:**
  - Slice of v at s is **the set of statements** involved in computing vs value at s
  - For statement s and variable v, the slice of program P with respect to the slicing criterion \( s, v \) includes only those statements of P needed to capture the behavior of v at s.
  - Forward slice: a set of statements that can be affected by v at s
- Computing program slice: using the PDG, one computes a program slice via a backwards transitive closure or reachability operation on the PDG.
- **Program chopping:**
  - Given source s and sink t, what program points transmit effects from s to t
  - Intersect backward slicing from t and forward slicing from s
Static and Dynamic Slicing

- **Static slicing**: along all the execution paths, which set of the statements may affect the value of \( v \) at \( s \)

- **Dynamic slicing**: for a particular execution, which set of the statements affect the value of \( v \) at \( s \)

- The relation of two types of slicing: informally, merging dynamic slices for all the inputs, we have the static slice; the static slice does not distinguish the paths

- Slicing algorithms and tools
Dataflow Analysis [3]

- A set of dataflow problems:
  - Reaching definitions (backward, may): for all the definitions, we determine which program points a definition reaches
  - Available expressions (backward, must): whether an expression is available at a program point
  - Live variable analysis (forward, may): which variables are alive at a program point

- Dataflow equations:
  - $\text{In}(B), \text{Out}(B), \text{Gen}(B), \text{Kill}(B)$
  - forward problem: how to compute $\text{Out}(B)$ using $\text{In}(B)$; backward problem: how to compute $\text{In}(B)$ using $\text{Out}(B)$

- Dataflow algorithms (works for a category of dataflow problem): worklist, demand-driven, bit-vector (at each program point, we will use a bit-vector to represent a set, an intermediate solution for the dataflow problem)

- Using lattice to prove the termination of dataflow algorithms
Abstract Interpretation [1]

A theoretical framework to formalize approximation
A collection of abstract values for a lattice, called an abstract domain

**Problem:** Compute a sound approximation $S^\#$ of $S$

$S \subseteq S^\#$

**Solution:** Galois connections
An Abstraction

- Define an abstract semantics that computes only the sign of the result.

\[ \sigma: \text{Exp} \rightarrow \{+,-,0\} \]

\[
\sigma(i) = \begin{cases} 
+ & \text{if } i > 0 \\
0 & \text{if } i = 0 \\
- & \text{if } i < 0 
\end{cases}
\]

\[
\sigma(e_1 \times e_2) = \sigma(e_1) \times \sigma(e_2)
\]
Abstract Interpretation - An Example

Soundness

• We can show that this abstraction is correct in the sense that it correctly predicts the sign of an expression.
• Proof is by structural induction on $e$.

\[
\mu(e) > 0 \iff \sigma(e) = + \\
\mu(e) = 0 \iff \sigma(e) = 0 \\
\mu(e) < 0 \iff \sigma(e) = -
\]
Points-To and Alias Analysis

- Undecidable problems
- **Pointer analysis** aims to determine what memory location a pointer points to?
- **Alias analysis** aims to determine when the two variables refer to the same memory/storage location
- **Shape analysis**: static analysis that discovers and verifies properties of linked, dynamically allocated data structures in (usually imperative) computer programs, e.g., discriminating between cyclic and acyclic lists and proving that two data structures cannot access the same piece of memory.
Importance of Alias Analysis

Compiler optimization:

1. \( X = 5 \)
2. \( *p = ... \)  \( // p \) may or may not point to \( X \)
3. \( ... = X \)

Constant propagation: assume \( p \) does point to \( X \) (i.e., in statement 3, \( X \) cannot be replaced by 5).

Dead Code Elimination: assume \( p \) does not point to \( X \) (i.e., statement 1 cannot be deleted).

Bug detection: we need to precisely track the relations, values and ranges of variables
Alias Analysis Algorithms

- Andersen-Style Pointer Analysis [4]
- Steensgaard-Style Pointer Analysis [27]
Andersen-Style Pointer Analysis [4]

- Flow-insensitive, context-insensitive analysis, first for C programs (1994) later for Java
- View pointer assignments as subset constraints, also called inclusion based algorithms
- Use constraints to propagate points-to information

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>a = &amp;b</td>
<td>a ⊇ {b}</td>
<td>loc(b) ∈ pts(a)</td>
</tr>
<tr>
<td>Simple</td>
<td>a = b</td>
<td>a ⊇ b</td>
<td>pts(a) ⊇ pts(b)</td>
</tr>
<tr>
<td>Complex</td>
<td>a = *b</td>
<td>a ⊇ *b</td>
<td>∀v ∈ pts(b). pts(a) ⊇ pts(v)</td>
</tr>
<tr>
<td>Complex</td>
<td>*a = b</td>
<td>*a ⊇ b</td>
<td>∀v ∈ pts(a). pts(v) ⊇ pts(b)</td>
</tr>
</tbody>
</table>
Steensgaard-Style Pointer Analysis [27]

- Uses equality constraints instead of subset constraints (1996)
- Unification based approach: assignment unifies the graph nodes, e.g., \( x = y \) (unified \( x \) and \( y \) in the same node), also called union-find algorithm, exclusion-based approaches, nearly linear complexity
- Less precise than Andersen-style, thus more scalable

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>( a = &amp;b )</td>
<td>( a \supset {b} )</td>
<td>( \text{loc}(b) \in \text{pts}(a) )</td>
</tr>
<tr>
<td>Simple</td>
<td>( a = b )</td>
<td>( a = b )</td>
<td>( \text{pts}(a) = \text{pts}(b) )</td>
</tr>
<tr>
<td>Complex</td>
<td>( a = *b )</td>
<td>( a = *b )</td>
<td>( \forall v \in \text{pts}(b). \text{pts}(a) = \text{pts}(v) )</td>
</tr>
<tr>
<td>Complex</td>
<td>( *a = b )</td>
<td>( *a = b )</td>
<td>( \forall v \in \text{pts}(a). \text{pts}(v) = \text{pts}(b) )</td>
</tr>
</tbody>
</table>
Symbolic Execution [6, 17, 11]

```c
int foo(int i) {
    int j = 2 * i;
    i = i++;
    i = i * j;
    if (i < 1) {
        i = -i;
    }
    return i;
}
```

Given $i_{\text{input}}$:

- $i = i_{\text{input}}$, $j = 2 \times i_{\text{input}}$
- $i = i_{\text{input}} + 1$, $j = 2 \times i_{\text{input}}$
- $i = 2 \times i_{\text{input}}^2 + 2 \times i_{\text{input}}$

OR

- $i = -2 \times i_{\text{input}}^2 - 2 \times i_{\text{input}}$
- $2 \times i_{\text{input}}^2 + 2 \times i_{\text{input}} < 1$
- $i = 2 \times i_{\text{input}}^2 + 2 \times i_{\text{input}}$
- $2 \times i_{\text{input}}^2 + 2 \times i_{\text{input}} \geq 1$
Symbolic Execution Tree

```c
int a = α, b = β, c = γ;
    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
} else { // if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
} else { assert(x+y+z!=3)
```
Detecting Infeasible Paths

Suppose we require $\alpha = \beta$

```c
int a = alpha, b = beta, c = gamma;
// symbolic
int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
} else if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
} else {
    assert(x+y+z!=3)
}
```

Path condition:

$\alpha \land (\beta < 5)$

Infeasible:

$\neg \alpha \land (\beta \geq 5)$
Test Input Generation

```
int a = α, b = β, c = γ;
   // symbolic
int x = 0, y = 0, z = 0;
if (a) {
   x = -2;
}
if (b < 5) {
   if (!a && c) { y = 1; }
   z = 2;
}
assert(x+y+z!=3)
```

Path 1: $\alpha = 1, \beta = 1$
Path 2: $\alpha = 1, \beta = 6$
Path 3 ...
Bug Finding

```c
int foo(int i){
    int j = 2*i;
    i = i++;
    i = i * j;
    if (i < 1)
        i = -i;
    i = j/i;
    return i;
}
```

```c
i_{input}  i_{input} = -1 Trigger the bug
```

**True branch:**
```
2* i_{input}^2 + 2* i_{input} < 1
i = - 2* i_{input}^2 - 2* i_{input}
i == 0
```

**False Branch:** always safe
```
2* i_{input}^2 + 2* i_{input} >= 1
i = 2* i_{input}^2 + 2* i_{input}
i == 0
```
Internal of Symbolic Executors: KLEE

C code → LLVM → LLVM bytecode → KLEE → Constraint Solver (STP)

x ≥ 0
x ≠ 1234

x = 3
x = 1234
x = -2
Three Challenges

- Path explosion
- Modeling program statements and environment
- Constraint solving
Solutions to the Challenges

- Path explosion: DFS, BFS, random, coverage-guided, combined, probabilistic-based ...
- Modeling program statements and environment: concretization, abstract models built manually or automatically
- Constraint solving (mainly use SMT solvers):
  - Handle well linear constraints, integer constraints
  - Handle bit-vector constraints
  - Developing string theories
  - No theories for float point computation
Important Symbolic Execution Tools

KLEE, SAGE, Java Pathfinder, Cloud9
Dynamic Analysis

- Learn from executions
- Three basic steps:
  - Run the program
  - Instrumentation and collect the data
  - Data analysis
- Experiment type of approaches
Instrumentation

- Static vs. dynamic instrumentation
- Source vs. binary instrumentation
- PIN, Valgrind
Dynamic Analysis

- Dynamic slicing [2]
- Dynamically generated program invariants [10]
Data Analysis

The techniques are originally used in the data mining and database field

- **Association rules mining**
  - Items, transactions, databases
  - Goal: learn implications between two itemsets
  - Mining algorithms: Apriori
  - Frequent itemset mining: http://fimi.ua.ac.be/

- **Sequential pattern mining**
  - Sequence, subsequence
  - Given a set of sequences, find the complete set of frequent subsequences
  - Algorithms:
    - Concept introduction and an initial Apriori-like algorithm: Agrawal & Srikant. Mining sequential patterns, ICDE95
    - Apriori-based method: GSP (Generalized Sequential Patterns: Srikant & Agrawal @ EDBT96)
    - Pattern-growth methods: FreeSpan & PrefixSpan (Han et al.@KDD00; Pei, et al.@ICDE01)
    - Vertical format-based mining: SPADE (Zaki@Machine Learning00)
    - Constraint-based sequential pattern mining (SPIRIT: Garofalakis, Rastogi, Shim@VLDB99; Pei, Han, Wang @ CIKM02)
    - Mining closed sequential patterns: CloSpan (Yan, Han & Afshar @SDM03)
C++ Implementations of Apriori, Eclat, FP-growth and several other algorithms are available on http://www.adrem.ua.ac.be/goethals/software/ and on http://fimi.cs.helsinki.fi/
Applications of Program Analysis

- Compiler optimizations
- Detecting infeasible paths
- Detecting program invariants
- Bug detection
- Test input generation
- Debugging [24]
- Program repair [21, 18, 20]
- Program synthesis [26]
Implementation and Experimentation

- Experimentation: to show theory is non-refutable; reject NULL-hypothesis
- Reproducible
- **Threats to Validity**: properties of scientific studies
  - Construct validity: to which degree a test measures what it claims to be measured
  - Internal validity: to which a causal relation established is warranted
  - External validity: to which extent, the result can be generalized to other objects beyond experimental studies
- Good example of threats to validity is given in the paper [5]
Platforms

- Soot
- LLVM
- Phoenix
Benchmarks

- Buffer Overflow Benchmarks [29]
- Bugbench [19]
- SIR Benchmarks
- Verisec
- BegBunch: benchmarks for C bug detection tools [8]
- RADBench: a Concurrency Bug Benchmark Suite [14]
- Spec: http://www.spec.org/
- Benchmarks for evaluating automatic program repairing techniques
- iBugs [9]
- MARMOSET [25]
- Siemens Suite [12]
- CoREBench [5]
S. Abramsky and C. Hankin.  

Hiralal Agrawal and Joseph R. Horgan.  
Dynamic program slicing.  

A.V. Aho.  
*Compilers: Principles, Techniques and Tools (for Anna University)*, 2/e.  
Pearson Education India, 2006.

Lars Ole Andersen.  
Program analysis and specialization for the C programming language.  

Marcel Böhme and Abhik Roychoudhury.  
Corebench: Studying complexity of regression errors.  
Cristian Cadar, Daniel Dunbar, and Dawson Engler.
Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.

Venkatesan T. Chakaravarthy.
New results on the computability and complexity of points–to analysis.

Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz.
Begbunch: Benchmarking for c bug detection tools.
Valentin Dallmeier and Thomas Zimmermann.
Extraction of bug localization benchmarks from history.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin.
Dynamically discovering likely program invariants to support program evolution.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin.
Grammar-based whitebox fuzzing.

Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand.
Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria.
Daniel Jackson and Eugene J. Rollins.
A new model of program dependences for reverse engineering.

Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen.
Radbench: A concurrency bug benchmark suite.

D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe.
Dependence graphs and compiler optimizations.

William Landi.
Undecidability of static analysis.
Wei Le and Mary Lou Soffa.  
Marple: A demand-driven path-sensitive buffer overflow detector.  

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer.  
Genprog: A generic method for automatic software repair.  

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou.  
Bugbench: Benchmarks for evaluating bug detection tools.  

Na Meng, Miryung Kim, and Kathryn S. McKinley.  
Systematic editing: Generating program transformations from an example.
Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra.
Semfix: Program repair via semantic analysis.

G. Ramalingam.
The undecidability of aliasing.

Thomas Reps and Genevieve Rosay.
Precise interprocedural chopping.

Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve.

Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh.
Software repository mining with marmoset: An automated programming project snapshot and testing system.

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster.
Path-based inductive synthesis for program inversion.

Bjarne Steensgaard.
Points-to analysis in almost linear time.
Mark Weiser.
Program slicing.

Testing static analysis tools using exploitable buffer overflows from open source code.