"Systemized" Static Analysis

Harry Xu

University of California, Los Angeles
Overview of My Work

What Kind of Mindset Do You Have?

Practicality  Elegancy

Systems  PL
Static Analysis: Has the Problem Been Solved?

Academia

- Hundreds of papers published in the past decade
- Algorithms become increasingly sophisticated

Industry

- Less than a dozen commercial analysis tools
- Use very simple algorithms
- Software becomes increasingly large and dynamic

More elegant

More practical
The Ever-increasing Gap

Scalability

Difficulty in implementation

Lost in multiple languages
Attempts from the PL Community

- Poor scalability
  - Trading off precision for scalability
  - Minimizing generated information

- Complicated implementations
  - Further complicates the implementation
  + Using declarative models such as Datalog
  - Fundamentally limited by a Datalog engine

- Lost in multiple languages
  - Nothing has been done
The Outside World

- The FB graph had **721M vertices (users)**, **68.7B edges (friendships)** in May 2011

- Google Maps had **20 petabytes of data** in 2015
Our “Large” Programs

- The Linux kernel, 16M lines of code; a fully inlined version has about 1B edges
- HBase, 1.37M lines of code; 128M edges in a fully inlined version
- Hadoop, 546K lines of code; 44M edges in a fully inlined version

FB Graph: 68.7B edges
Time for a Mindset Shift?

It is not because our programs are too large, but because we haven’t thought about how to develop *scalable systems*
“Big Data” Thinking

Solution =

(1) Large Dataset + (2) Simple Computation +

System Design

Don’t complicate the algorithm

Don’t worry about too much (intermediate) data

Don’t stop at the interface between app and system

Leave the algorithm simple

Leverage modern computing resources

Design and implement customized systems
What We Did

Built single-machine, disk-based systems specifically for the static analysis workload

- Graspan: a graph system for CFL-reachability computation [ASPLOS’17]
- Grapple: a graph system for finite-state property checking [In Submission]
Why Systemized Static Analyses Work

• Poor scalability
  No longer worry about memory blowup as we have disk-support

• Complicated implementations
  Analysis developers only implement a few interfaces; No longer worry about performance

• Lost in multiple languages
  Components in different languages are turned into graphs of the same format and analyzed together
Graspan: Context-Free Language (CFL) Reachability

- A program graph $P$

- A context-free Grammar $G$ with balanced parentheses properties

$K \rightarrow l_1 l_2$

c is $K$-reachable from $a$

Reps, Program analysis via graph reachability, IST, 1998
A Wide Range of Applications

- Pointer/alias analysis

- Dataflow analysis, pushdown systems, set-constraint problems can all be converted to context-free-language reachability problems

Sridharan and Bodik, Refinement-based context-sensitive pointsto analysis for Java, *PLDI*, 2006
A Wide Range of Applications (Cont.)

- Pointer/alias analysis

\[ b = \& a; \ // \text{Address-of} \]
\[ c = b; \ // \text{Dereference} \]
\[ d = \*c; \ // \text{Dereference} \]

- \textit{Address-of \& / dereference*} are the open/close parentheses

Sridharan and Bodik, Refinement-based context-sensitive pointsto analysis for Java, \textit{PLDI}, 2006
“Big Data” Thinking

Solution =

(1) Large Dataset + (2) Simple Computation +

System Design
Turning Code Analysis into Data Analytics

• Key insights:
  – The input is a fully inlined program graph
  – Adding transitive edges explicitly – satisfying (1)
  – Core computation is adding edges – satisfying (2)
  – Leveraging disk support for memory blowup

• Can existing graph systems be directly used?
  – No, none of them support dynamic addition of a lot of edges
    (1) Online edge duplicate check and (2) dynamic graph repartitioning
Graspan [Wang-ASPLOS’17]

• Scalable
  – Disk-based processing on the developer's work machine

• Parallel
  – Edge-pair centric computation

• Easy to implement a static analysis
  – Implement a few interfaces

4 students + 1 postdoc, 1.5 years of development; implemented in both Java and C++
https://github.com/Graspan/
How It Works?

GRAMMAR RULES
Granspan Design

Preprocessing

Edge-Pair Centric Computation

Postprocessing
Computation Occurs in Supersteps

Preprocessing

Edge-Pair Centric Computation

Postprocessing
Each Superstep Loads Two Partitions

Grammar: $C := AB$  $D := BC$  $B := AD$  $A := CD$

Preprocessing  Edge-Pair Centric Computation  Postprocessing
Each Superstep Loads Two Partitions

Grammar: \[ C = AB, \quad D = BC, \quad E = AB, \quad A := CD \]

We keep iterating until delta is 0.
Post-Processing

- Repartition oversized partitions to maintain balanced load on memory
- Save partitions to disk
- Scheduler favors in-memory partitions and those with higher matching degrees
What We Have Analyzed

<table>
<thead>
<tr>
<th>Program</th>
<th>#LOC</th>
<th>#Inlines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 4.4.0-rc5</td>
<td>16M</td>
<td>31.7M</td>
</tr>
<tr>
<td>PostgreSQL 8.3.9</td>
<td>700K</td>
<td>290K</td>
</tr>
<tr>
<td>Apache httpd 2.2.18</td>
<td>300K</td>
<td>58K</td>
</tr>
</tbody>
</table>

- With
  - A fully context-sensitive pointer/alias analysis
  - A fully context-sensitive dataflow analysis

- On a Dell Desktop Computer with 8GB memory and 1TB SSD
Evaluation

• Can the interprocedural analyses improve D. Englers’ checkers?
  – Found 85 new NULL pointer bugs and 1127 unnecessary NULL tests in Linux 4.4.0-rc5

• How well does Graspan perform?
  – Computations took 11 mins – 12 hrs

• How does Graspan compare to other systems?
  – GraphChi crashed in 133 seconds
  – Traditional implementations of these algorithms ran out of memory in most cases
  – Datalog (SociaLite) –based implementation ran out of memory in most cases

• Will try a differential dataflow system like Naiad
Grapple: A Finite-State Property Checker

• Many bugs in large-scale systems have finite-state properties
  – Many OS bugs studied in Chou et al. in 2001 are finite state property bugs: misplaced locks, use-after-free, etc.
  – Most distributed system bugs studied in Gunawi et al. in 2014 are finite state property bugs: socket leaks, task state problems, mishandled exceptions, etc.

Gunawi et al., What bugs live in the cloud? a study of 3000+ issues in cloud systems, SoCC, 2014
Chou et al., An empirical study of operating systems errors, SOSP, 2001
Analyses Under the Hood

• What we need for the checker
  – Extract sequences of method calls on each object of interest
  – Check them against the FSM specification

• What analyses we need
  – Alias analysis
  – Dataflow analysis
  – Context sensitivity and path sensitivity
Grapple

• Phases
  – A fully path-sensitive, context-sensitive alias analysis
  – A fully path-sensitive, context-sensitive dataflow analysis
  – Extract event sequences

• Computation Model
  – Edge-pair-centric model
  – Challenge: how to represent and solve path constraints during graph processing
Grapple Computation Model

- A program graph $P$
- A context-free grammar $G$ with balanced parentheses properties
- $C = c_1 \land c_2$ is satisfiable

$c$ is $K$-reachable from $a$

$K \rightarrow l_1 \, l_2$
Path Constraint Representation

- Challenges
  - Each edge carries only fixed-size data
  - The size is often smaller than 4 bytes
- Using *interprocedural control flow execution tree* (ICFET) as an index engine
- Each edge contains a path encoding, which is used to query for a path constraint based upon ICFET
Control Flow Execution Tree (CFET)

A simple numbering algorithm: T child -> ID * 2; F child -> ID * 2 + 1

Built before the graph computation starts
Path Representation

- An intraprocedural CFET path can be uniquely encoded as a pair \([ID_{\text{start}}, ID_{\text{end}}]\)
- Decoding can be done efficiently *online*
- Loops are unrolled a certain number of times

Example: \([0, 6]\) uniquely identifies the right most path

Decoding can be done by right shifts

Symbolic execution used to compute conditions
void foo (int x) {
    int y = x + 1;
    if (x > 0) { y = bar (2 * x); //f_2 }
    if (y < 0) {...} return;
}

int bar (int a) {
    if (a < 0) {return a + 1;}
    return a - 1;
}

Connecting callers with callees using call and return edges, annotated with call site IDs and symbolic equations
Interprocedural Path Representation

- A sequence of intervals
  - [2, 0], 25, [2, 0]
  - Bounded by the call stack depth
- A constraint can be computed by extracting constraints for path fragments and combining them into a conjunctive form
Computation

• Use Graspan’s edge-pair-centric computation model
• Z3 is used for constraint solving
• Each partition is much easier to become imbalanced
  – Eager repartitioning during the computation
## Evaluation Subjects

<table>
<thead>
<tr>
<th>Program</th>
<th>#LoC</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ZooKeeper</td>
<td>206K</td>
<td>3.5.0</td>
</tr>
<tr>
<td>Apache Hadoop</td>
<td>568K</td>
<td>2.7.5</td>
</tr>
<tr>
<td>HDFS</td>
<td>546K</td>
<td>2.0.3</td>
</tr>
<tr>
<td>Apache HBase</td>
<td>1.37M</td>
<td>1.1.6</td>
</tr>
</tbody>
</table>
Checkers Implemented

- IO checker
- Socket checker
- Exception handling checker
- Lock usage checker

- Checkers: 3.2K lines of Java code
- Grapple: 13K lines of C++ code, with about 1.5K lines reused from Graspan
- 1 postdoc + 5 students, 1 year of effort
# Bugs Found

<table>
<thead>
<tr>
<th>Checker</th>
<th>I/O</th>
<th>lock</th>
<th>except.</th>
<th>socket</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RE</td>
<td>FP</td>
<td>RE</td>
<td>FP</td>
<td>RE</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>Hadoop</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>HDFS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>43</td>
</tr>
<tr>
<td>HBase</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>176</td>
</tr>
</tbody>
</table>

Grapple reported a total of **359 true bugs** and **17 false warnings**

4.7% false warning rate
# Grapple Performance

<table>
<thead>
<tr>
<th>Subject</th>
<th>#V (K)</th>
<th>#EB (K)</th>
<th>#EA (K)</th>
<th>PT</th>
<th>CT</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>6,737</td>
<td>12,907</td>
<td>20,953</td>
<td>1m55s</td>
<td>2h27m3s</td>
<td>2h28m58s</td>
</tr>
<tr>
<td>Hadoop</td>
<td>21,883</td>
<td>77,165</td>
<td>93,150</td>
<td>4m40s</td>
<td>7h24m41s</td>
<td>7h29m21s</td>
</tr>
<tr>
<td>HDFS</td>
<td>7,610</td>
<td>17,977</td>
<td>29,354</td>
<td>2m35s</td>
<td>3h37m43s</td>
<td>3h40m19s</td>
</tr>
<tr>
<td>HBase</td>
<td>44,754</td>
<td>128,180</td>
<td>202,045</td>
<td>19m36s</td>
<td>18h42m42s</td>
<td>19h02m18s</td>
</tr>
</tbody>
</table>

The execution time ranges from 2.5 hours to 19 hours
Performance Breakdown

I/O > Constraint lookup ≈ SMT solving > Edge computation

![Performance Breakdown Diagram](image)
Conclusion

• Develop systems to solve PL problems
• Try them out
  – https://github.com/graspan
  – https://github.com/grapple-system
Acknowledgements

• My (current and former) students and postdocs
  – Zhiqiang Zuo (postdoc 2015 – 2018, currently an Ass. Prof. at Nanjing University)
  – Kai Wang (Ph.D. student)
  – John Thorpe (Ph.D. student)
  – Aftab Hussain (M.S. student)
PL for Systems

I/O, Network, Computation Model, ...

Memory management, compilation, hybrid memories, ...

Systems

Language Runtime

Existing Work
Apache Hama

My Work
Big Data

Asterix DB
Spark
Systems for PL

PL Problems

SAT Solver, Program Analysis, Model Checking, ...

System Solutions

Big Data Systems

Scalable Results

Existing Work

Our Work
Evaluation II

• Is Graspan efficient and scalable?
  – Computations took 11 mins – 12 hrs

<table>
<thead>
<tr>
<th>Prog</th>
<th>IS=(E,V)</th>
<th>PS=(E,V)</th>
<th>PT</th>
<th>SS</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>(249.5M,52.9M)</td>
<td>(1.1B,52.9M)</td>
<td>91</td>
<td>27</td>
<td>1.7 hrs</td>
</tr>
<tr>
<td>PSQL</td>
<td>(25.0M,5.2M)</td>
<td>(862.2M,5.2M)</td>
<td>10</td>
<td>16</td>
<td>6.0 hrs</td>
</tr>
<tr>
<td>httpd</td>
<td>(8.2M, 1.7M)</td>
<td>(904.3M, 1.7M)</td>
<td>3</td>
<td>13</td>
<td>8.4 hrs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prog</th>
<th>IS=(E,V)</th>
<th>PS=(E,V)</th>
<th>PT</th>
<th>SS</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>(69.4M, 63.0M)</td>
<td>(211.3M, 63.0M)</td>
<td>65</td>
<td>33</td>
<td>11.9 hrs</td>
</tr>
<tr>
<td>PSQL</td>
<td>(34.8M,29.0M)</td>
<td>(56.1M, 29.0M)</td>
<td>35</td>
<td>16</td>
<td>2.4 hrs</td>
</tr>
<tr>
<td>httpd</td>
<td>(10.0M, 5.3M)</td>
<td>(19.3M, 5.3M)</td>
<td>9</td>
<td>16</td>
<td>11.4 mins</td>
</tr>
</tbody>
</table>
Evaluation III

- Graspan v/s other engines?
  - GraphChi crashed in 133 secs

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Graspan</th>
<th>ODA [101]</th>
<th>SociaLite [45]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CT</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>Linux-P</td>
<td>99.7 mins</td>
<td>46.6 secs</td>
<td>OOM</td>
</tr>
<tr>
<td>Linux-D</td>
<td>713.8 mins</td>
<td>4.2 mins</td>
<td>-</td>
</tr>
<tr>
<td>PostgreSQL-P</td>
<td>353.1 mins</td>
<td>4.5 mins</td>
<td>&gt; 1 day</td>
</tr>
<tr>
<td>PostgreSQL-D</td>
<td>143.8 mins</td>
<td>57.1 secs</td>
<td>-</td>
</tr>
<tr>
<td>httpd-P</td>
<td>497.9 mins</td>
<td>7.6 mins</td>
<td>&gt; 1 day</td>
</tr>
<tr>
<td>httpd-D</td>
<td>11.3 mins</td>
<td>3.3 secs</td>
<td>4 hrs</td>
</tr>
</tbody>
</table>

x = parse(args[0]); y = x;

FilterWriter out = null, o = null;

if(x > 0) {
    out = new FilterWriter();
    o = out;
    y--;
}
else { y++; }

if(y > 0) {out.write(...); o.close();}

return;