Declarative Analysis of Binary Code

G. Gary Tan

School of EECS, Penn State Univ.

U of Wisconsin, Oct 23rd, 2023
Analyzing Binary Code

Unavailable source code
Malware analysis

Analyzing Binary Code ... is Challenging

- Less semantic info
- Mixed code & data
- Var-sized instrs
- Unstructured control flow

...
State of Binary Analysis

• Many techniques
  – Based on **heuristics** (e.g., code patterns)
  – Make heavy assumptions about compilation toolchains

• Sometimes partial coverage of code behavior
  – Dynamic symbolic execution; ML techniques
  – Great at bug finding and exploit generation; unsuitable for producing **defenses**

• Principled methods
  – Value-set analysis (VSA) [Balakrishnan and Reps 2004]
Towards Robust, Precise, and Scalable Binary Reverse Engineering

• **Declarative analysis rules**
  – Rules are mutually dependent
  – Solved by a high-performance solver

• **Block memory model**
  – Speed up pointer analysis by simplifying the reasoning of pointer arithmetics

• **Work in progress**
Mutual Dependency among Basic Binary Analyses
Basic Task: Instruction Discovery

- What instructions are in a binary?
  - Variable-sized instructions
  - Data embedded into the code section
Basic Task: Control Flow

• What are the targets of control-flow transfers?
  – Control-flow graph (CFG)
  – Not easy for indirect branches: indirect calls/jumps and returns

... call r12

What functions can it target?
Basic Task: **Pointer analysis**

- What objects may a pointer point to?
  - Important for understanding both control and data flow

```plaintext
... call r12

Control flow: what functions can r12 point to?
```

```plaintext
... r2 := mem(r10)

Data flow: from what objects can it read data?
```
Mutual Dependency among Basic Tasks

- **Instr discovery**
  - Without instructions, cannot talk about control flow

- **Control flow**
  - Without control flow, cannot follow control-flow targets to discover new instrs

- **Pointer analysis**
  - Without control flow, cannot perform precise pointer analysis
  - Without pointer analysis, cannot know the targets of indirect branches

Better together: run all basic tasks simultaneously!
A Rule-Centric Approach

• Same phenomenon for source-level analysis
  – E.g., “Exception analysis and points to analysis: Better together” [Smaragdakis et al.]

• Using mutually dependent Datalog rules to perform a joint analysis
Datalog

- A declarative logic programming language
  - A program is a set of declarative rules
- Similar to Prolog but restricts the use of cuts, negation, and recursion
  - E.g., stratification restriction on negation and recursion
  - Restrictions make execution efficient and terminate
  - Rule/parameter ordering does not matter for correctness
- Efficient solvers
Benefits of Declarative Rules

Modularity

• Individual analyses as separate modules
• Easy to replace one analysis with a different algorithm/implementation
• Solver auto computes a fixed point solution

High performance

• The declarative style enables parallel solvers
• Previous source-level research shows the Datalog approach is more scalable
  • E.g., enable a deeper context
The Block Memory Model
Memory Model of VSA

• Divide memory into a set of abstract locations (alocs)
  – An aloc holds a variable-like entity
• E.g., the global mem region

```
<table>
<thead>
<tr>
<th>aloc1</th>
<th>aloc2</th>
<th>aloc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>An array</td>
<td>A pointer</td>
<td>An int</td>
</tr>
</tbody>
</table>
```

• Offsets are tracked via the strided-interval domain
• p+T can point to any aloc
The Block Memory Model

- Memory divided into a set of memory blocks
- Pointer-arithmetic assumption
  - $p+o$ can point to only the same block as what $p$ points to
The Block Memory Model

• Assume: there is an **infinite amount of space** between two blocks

```
   blk3
   blk2
 p+o  'p'
   blk1
```

• Justification: analysis should capture legal behavior of code
  – Given correct block boundaries, illegal behavior if an out-of-bound “p+o” is used to access memory
The Block Memory Model

• First proposed in CompCert [Leroy et al.]
  – To specify the memory model of C-like languages and verify correctness of program transformations

• We use it to simplify pointer analysis
  – p plus any offset (even T) points to the same block
  – More scalable
BPA [NDSS 2021];
BinPointer [CC 2022]
BPA & BinPointer

• Formulated as a set of declarative Datalog rules

• Interprocedural binary pointer analyses based on the block memory model
  – Context insensitive
  – Flow insensitive on memory but are flow sensitive on registers (with static single assignment form)
Design Choices

• BPA: track what blocks a pointer might point to
  – No offset tracking
  – Pointers have unknown offsets within blocks: b[T]

• BinPointer
  – Distinguishes slots within blocks by tracking some offsets
  – 0-base abstraction: tracks only those offsets resulting from pointer arithmetics using b[0]
Workflow

Decoding; convert to an IR

Binary code → Input preprocessing → Program facts

CFG + Points-to info → Datalog

Reachable instr & control flow & pointer ana. & SSA → Block generation

Datalog

Decoding; convert to an IR
Datalog Rule Primer

• Rule in the form of

\[ c :\!\!\!: - a_1, a_2, \ldots, a_n. \]

If assumptions \( a_1 \) to \( a_n \) all hold, then conclusion \( c \) holds.

• Example:

\[ \text{parent}(x, y) :\!\!\!: - \text{mother}(x, y). \]

• The assumptions/conclusion are expressed with relations
  – parent(x,y); mother(x,y)


## Example Relations for Binary Analysis

**Input relation**: perform decoding at every code address

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{InstrAt}(i, addr)$</td>
<td>Instruction $i$ is at address $addr$</td>
</tr>
<tr>
<td>$\text{Reachable}(addr)$</td>
<td>Address $addr$ is reachable from the program entry</td>
</tr>
<tr>
<td>$\text{CallEdge}(callsite, f)$</td>
<td>The call instruction at $callsite$ can target function $f$</td>
</tr>
<tr>
<td>$\text{PointsTo}(r, f)$</td>
<td>Register $r$ points to the beginning address of function $f$</td>
</tr>
</tbody>
</table>

**Output relations**

- **Instr discovery**
- **Control flow**
- **Pointer ana.**
Example (Simplified) Rules

\begin{align*}
\text{Reachable}(addr) &\, :\, - \\
&\quad \text{CallEdge}(callsite, f), \text{InFun}(f, addr).
\end{align*}

\begin{align*}
\text{CallEdge}(callsite, f) &\, :\, - \\
&\quad \text{Reachable}(callsite), \\
&\quad \text{InstrAt}(call \, r, \, callsite), \\
&\quad \text{PointsTo}(r, f).
\end{align*}

\begin{align*}
\text{PointsTo}(r, f) &\, :\, - \\
&\quad \text{Reachable}(addr), \\
&\quad \text{InstrAt}(r=\text{mem}(r' + o), \, addr), \\
&\quad \text{PointsTo}(r', b), \, \text{PointsTo}(b, f)
\end{align*}

Instr discovery depends on control flow.

Control flow depends on instr discovery and pointer analysis.

Pointer analysis depends on instr discovery.
Modularity of Datalog Rules

• Can change rules for one component without affecting other components
  – E.g., change the rules for pointer analysis
• A Datalog solver performs a fixed-point calculation
  – Without user involvement to resolve mutual dependency
Workflow

Binary code → Input preprocessing → Program facts

Datalog

Datalog

CFG + Points-to info

Reachable instr & control flow & pointer ana. & SSA

Block generation
Memory Block Generation

• Partition a memory region into abstract blocks
  – Need to be conservative
  – Cannot split a compound data structure such as an array

• Coarse-grained heap blocks
  – Use the allocation-site ID to identify the abstract block of memory allocated at that site
Mem Block Generation: Global Region

- Use symbol tables to identify boundaries
- Without symbol tables

Step 1: identify global addresses used in pointer arithmetics (addr1+r, addr2+r*scale, ...)
Step 2: estimate the range of addresses accessed based on a step-1 address
Step 3: eliminate addresses in the middle of a step-2 range
Evaluation: Benchmarks

• x86 binaries compiled from
  – SPEC CPU 2006 benchmarks
  – And security-critical applications: thttpd, Memcached, lighttpd, exim, nginx
  – Multiple compilers and O0-O3 optimization levels
Performance Evaluation

<table>
<thead>
<tr>
<th></th>
<th>hmmer</th>
<th>h264ref</th>
<th>nginx</th>
<th>gobmk</th>
<th>perlbench</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td># of instrs</td>
<td>60k</td>
<td>100k</td>
<td>153k</td>
<td>157k</td>
<td>226k</td>
<td>647k</td>
</tr>
<tr>
<td>BPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (s)</td>
<td>79</td>
<td>379</td>
<td>2,793</td>
<td>1,933</td>
<td>4,006</td>
<td>27,619</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>0.6</td>
<td>3.6</td>
<td>24</td>
<td>28</td>
<td>57</td>
<td>352</td>
</tr>
<tr>
<td>BinPointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (s)</td>
<td>510</td>
<td>23,020</td>
<td>7,692</td>
<td>Inf</td>
<td>Inf</td>
<td>Inf</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>1.8</td>
<td>8.8</td>
<td>41</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

- BPA achieves reasonable performance on large binaries
  - Largest benchmark gcc: terminated within 10 hours with 352GB
- BAP-VSA (CAV’11) didn’t terminate within 10 hours
  - Another paper (BDA, OOPSLA’19) showed similar result
Soundness and Precision Evaluation

• Challenge: **no ground truth** for CFG and points-to information
  – Our strategy: use dynamic profiling results to perform validation
  – Collected dynamic traces by running benchmarks with reference inputs (with Intel’s Pin tool)
CFG Soundness Validation

• Dynamic traces include runtime targets of indirect branches

• Indirect-branch **recall rate**
  – Percentage of dynamic indirect-branch targets covered by the static CFG computed by binary analysis

• Results: recall rate 100%
Pointer Analysis Soundness Validation

• Pointer analysis results: b[o]
• During trace collection, need to convert flat mem addresses to block memory addresses

- Results: recall rate 100%

![Diagram of memory blocks with addresses](image)
CFG Precision

- AICT: the average # of indirect call targets in the CFG
  - The lower, the more precise
- 34.5% higher precision than TypeArmor’s arity approach

<table>
<thead>
<tr>
<th></th>
<th>Arity</th>
<th>BPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>*456.hmmer</td>
<td>22.0</td>
<td>2.8</td>
</tr>
<tr>
<td>*464.h264ref</td>
<td>28.9</td>
<td>26.4</td>
</tr>
<tr>
<td>*445.gobmk</td>
<td>1413.3</td>
<td>1297.2</td>
</tr>
<tr>
<td>*400.perlbench</td>
<td>536.6</td>
<td>363.7</td>
</tr>
<tr>
<td>*403.gcc</td>
<td>581.8</td>
<td>427.8</td>
</tr>
<tr>
<td>Exim</td>
<td>38.0</td>
<td>30.6</td>
</tr>
<tr>
<td>Nginx</td>
<td>432.4</td>
<td>511.0</td>
</tr>
</tbody>
</table>

AICT table (binaries compiled by GCC-9.2, -O2)
Pointer Analysis Precision Metric

• A metric based on dynamic trace data and the block memory model

• Intuition: for a memory access to block b

<table>
<thead>
<tr>
<th>Trace data</th>
<th>Pointer analysis</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0], b[1]</td>
<td>b[0], b[1], b[2]</td>
<td>2/3</td>
</tr>
<tr>
<td>b[0], b[1]</td>
<td>b[T]</td>
<td>2/sizeof(b)</td>
</tr>
</tbody>
</table>

• The overall precision is then the average precision across all memory accesses and all blocks
## Points-to Precision Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Details</th>
<th>Runtime-based precision results (%)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Stack</td>
<td>Global</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPA</td>
<td>BinP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td>O0</td>
<td>3.3K</td>
<td>18.9</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>2.4K</td>
<td>26.3</td>
</tr>
<tr>
<td>lbm</td>
<td>O0</td>
<td>6.5K</td>
<td>17.9</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>2.2K</td>
<td>22.3</td>
</tr>
<tr>
<td>libquantum</td>
<td>O0</td>
<td>10K</td>
<td>65.6</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>9.6K</td>
<td>47.9</td>
</tr>
<tr>
<td>bzip2</td>
<td>O0</td>
<td>21K</td>
<td>38.8</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>11K</td>
<td>16.9</td>
</tr>
<tr>
<td>sjeng</td>
<td>O0</td>
<td>32K</td>
<td>24.8</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>22K</td>
<td>32.7</td>
</tr>
<tr>
<td>milc</td>
<td>O0</td>
<td>31K</td>
<td>40.0</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>23K</td>
<td>49.4</td>
</tr>
<tr>
<td>hmer</td>
<td>O0</td>
<td>88K</td>
<td>30.0</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>60K</td>
<td>38.0</td>
</tr>
<tr>
<td>h264ref</td>
<td>O0</td>
<td>161K</td>
<td>28.9</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>100K</td>
<td>35.3</td>
</tr>
</tbody>
</table>
Limitations & Future Work

• Extend analysis coverage to binaries
  – That are of other ISAs (e.g., ARM)

  **IR-level analysis**

  – That are compiled from C++ code

  **Object sensitivity**
Future Work: Applications

• Binary hardening
  – Control-flow integrity (CFI)
  – Data-flow integrity (DFI)
  – Binary-level debloating and privilege separation
    • Program-dependence graph (PDG)
    • Previous work at the source-code level: PDG-based privilege separation (PtrSplit [CCS ’17], Program mandering [CCS ’19])
Future Work: Applications

• Binary applications that require only approximately-correct analysis results
  – Clone detection; authorship attribution; provenance analysis; software evolution; ...

• Our pointer analysis can enhance ML models with features on memory objects
Future Work: Performance Improvement of Datalog Solvers

• Datalog does not support lattices
  – The solver only adds new facts, but **old facts cannot be updated**

• This greatly impacts runtime performance and memory consumption
  – Analysis of gcc: took 10 hours with 352GB memory
Future Work: Improving Datalog Solvers

• “From Datalog to Flix: A declarative language for fixed points on lattices” [PLDI ‘16]
  – But didn’t focus on performance; no parallelization

• Preliminary work
  – Extended Souffle’s interpreter to support lattices; 2x-10x speed up

• Plan: integrate lattice support into Souffle’s compiler
  – Expect a 100x speed up
  – Related work: Bring Your Own Data Structures to Datalog [OOPSLA ‘23]
Conclusions

• “There's Plenty of Room at the Bottom ...”
• Lots of interests from the security community and funding agencies
• Declarative analysis rules are
  – Easy to read, easy to write, easy to debug, easy to modularize
• Need more research on binary-level abstractions
Acknowledgements

• Sponsors
  
  ![NSF Logo]  
  ![Office of Naval Research Logo]

• Thanks to students and collaborators

  ![Sun Hyoung Kim]  
  ![Dongrui Zeng]  
  ![Cong Sun]

• BPA open sourced
  
  – [https://bitbucket.org/psu_soslab/bpa](https://bitbucket.org/psu_soslab/bpa)