A LOOK BACK ON
A FUNCTION
IDENTIFICATION
PROBLEM

Hyungjoon Koo, Soyeon Park,
and Taesoo Kim
Function Identification Problem

Problem definition
- Discover a set of function boundaries in a binary
- No symbol or debugging information readily available

A binary function is
- Defined by a developer from source code (e.g., user-defined function)
- Generated by a compiler (e.g., stack canary check)
- Inserted by a linker (e.g., CRT function)

Why important?
- Serve as a basis for reversing executable binaries
- Many applications: binary transformation, malware analysis, call graph reconstruction, etc.
- Almost every binary analysis tool includes a feature of function recognition
Common Challenges

Code optimization often blurs a clear function signature
  e.g., removing function prologue and epilogue, function inlining

Compiler-generated code or compiler-specific heuristics

Mixed code and data
  e.g., jump table

Non-returning functions
  e.g., ending with a call (tail call)

Code from a manually written assembly
Existing Approaches

Rule-based approach
- Disassemble code
  - Linear disassembly: Linearly disassemble all code (e.g., objdump)
  - Recursive traversal: Starting from an entry point and following a direct control flow transfer
- Seek function signature matching (e.g., function prologue)
- Downsides
  - Cannot identify functions in case of missing signatures (patterns)
  - Disassembly failure: code/data intermixed or indirectly reachable (or unreachable) functions cannot be recognized

Machine learning oriented approach
- Conditional random field (CRF)
- Weighted prefix tree
- Recurrent neural network (RNN)
## Summary of Prior Works

<table>
<thead>
<tr>
<th>Work</th>
<th>Year</th>
<th>Dataset</th>
<th>F1</th>
<th>Arch</th>
<th>Binaries</th>
<th>Compared To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus</td>
<td>2017</td>
<td>SPEC2006, nginx, lighttpd, opensshd, vsftp, exim</td>
<td>0.97</td>
<td>x86/x64</td>
<td>476</td>
<td>Dyninst, ByteWeight, IDA</td>
</tr>
<tr>
<td>Qiao et al.</td>
<td>2017</td>
<td>GNU Utils, SPEC2006, Glibc</td>
<td>0.985</td>
<td>x86/x64</td>
<td>2,488</td>
<td>ByteWeight, Shin:RNN</td>
</tr>
<tr>
<td>Jima</td>
<td>2019</td>
<td>GNU Utils, SPEC2017, Chrome</td>
<td>0.997</td>
<td>x86/x64</td>
<td>2,860</td>
<td>ByteWeight, Shin:RNN, IDA Free, Ghidra, Nucleus</td>
</tr>
<tr>
<td>ByteWeight</td>
<td>2014</td>
<td>GNU Utils</td>
<td>0.929</td>
<td>x86/x64</td>
<td>2,200</td>
<td>Dyninst, BAP, IDA</td>
</tr>
<tr>
<td>Shin:RNN</td>
<td>2015</td>
<td>GNU Utils</td>
<td>0.983</td>
<td>x86/x64</td>
<td>2,200</td>
<td>ByteWeight</td>
</tr>
<tr>
<td>FID</td>
<td>2017</td>
<td>GNU coreutils</td>
<td>0.930</td>
<td>x86/x64</td>
<td>4,240</td>
<td>IDA, ByteWeight</td>
</tr>
</tbody>
</table>
A Look Back on Function Identification

(Q1) Is the previous dataset appropriate for evaluation?
(Q2) Have prior evaluations been properly interpreted?
(Q3) Is the current metric (i.e., precision, recall, F1) fair enough?
(Q4) Are recent advances with an ML-centered approaches superior to rule-based ones?
(Q5) Is there a tool’s own characteristic or behavior?
Our Focus

Is NOT about

• Raising a question on the reproducibility or correctness of prior work
• Ranking the existing approaches (i.e., which one is the best and the worst?)

Is about

• Filling the void of what has been overlooked or misinterpreted
• Revisiting the previous datasets, metrics, and evaluations
• Bringing up the question of “Has the function identification problem been fully addressed?”
Q1. Appropriateness of Dataset

GNU utilities (129)

- **ByteWeight** released 16 **binutils**, 104 **coreutils**, and 9 **findutils**
- Most subsequent works utilize them for their evaluations
- **coreutils** has a static library (**libcoreutils.a**) in common → many redundant functions

Normalization for ML approaches

- Pre-processing step to reduce the number of instructions to feed a model
- 17.6K / 146K (12.1%) remain unique (ByteWeight)
- 91.4% in a test set has been discovered in a training set

<table>
<thead>
<tr>
<th>Group</th>
<th>Files</th>
<th>Funcs</th>
<th>Set</th>
<th>Group</th>
<th>Files</th>
<th>Funcs</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>57</td>
<td>19,996</td>
<td>train</td>
<td>Group 6</td>
<td>49</td>
<td>12,236</td>
<td>train</td>
</tr>
<tr>
<td>Group 2</td>
<td>55</td>
<td>9,475</td>
<td>train</td>
<td>Group 7</td>
<td>48</td>
<td>12,197</td>
<td>train</td>
</tr>
<tr>
<td>Group 3</td>
<td>51</td>
<td>18,442</td>
<td>train</td>
<td>Group 8</td>
<td>46</td>
<td>12,324</td>
<td>train</td>
</tr>
<tr>
<td>Group 4</td>
<td>57</td>
<td>13,779</td>
<td>train</td>
<td>Group 9</td>
<td>46</td>
<td>20,680</td>
<td>test</td>
</tr>
<tr>
<td>Group 5</td>
<td>55</td>
<td>13,481</td>
<td>train</td>
<td>Group 10</td>
<td>52</td>
<td>13,519</td>
<td>train</td>
</tr>
</tbody>
</table>
Q2. Re-interpretation of Prior Evaluations

Noticeable reports
- ByteWeight: F1 of 98.8 for ELF x64
- Shin’s RNN: F1 of 98.3
- LEMNA (Shin’s RNN re-implementation): 99.99% accuracy

Are we there yet?
- Re-experimentation with a different dataset (e.g., SPEC2017, other utilities of our choice)
- Retraining the ByteWeight model with our dataset: F1 of 78.0
- LEMNA’s accuracy comes from the number of decisions per byte (i.e., large # of true negatives)
- The LEMNA results with our dataset: precision of 94.5, recall of 86.1
Q3. Rethinking of Current Metrics

(Case 1) Non-continuous functions
Q4. Effectiveness of ML Techniques

Comparison of the number of true functions

- RNN VS Rule-based approaches

- Non-returning function detection (ending with call, jump, or __exit)

<table>
<thead>
<tr>
<th>Tool</th>
<th># of Missing</th>
<th>Total</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDA Pro</td>
<td>0</td>
<td>9,409</td>
<td>0.00%</td>
</tr>
<tr>
<td>Ghidra</td>
<td>54</td>
<td>9,409</td>
<td>0.57%</td>
</tr>
<tr>
<td>Nucleus</td>
<td>1,186</td>
<td>9,409</td>
<td>12.60%</td>
</tr>
<tr>
<td>Byteweight</td>
<td>4,615</td>
<td>9,409</td>
<td>49.05%</td>
</tr>
<tr>
<td>Byteweight*</td>
<td>2,024</td>
<td>5,125</td>
<td>39.49%</td>
</tr>
<tr>
<td>Shin:RNN</td>
<td>24</td>
<td>250</td>
<td>9.60%</td>
</tr>
</tbody>
</table>
Q5. Faithfulness of Tools

[Case study – IDA Pro] Under Reporting
- Disassembly with recursive traversal
- Intentionally does not seek unused functions (e.g., from an object file at link time)

[Case study – Ghidra] Over Reporting
- `objdump` or `nm` read function symbols merely from a symbol table
- Ghidra discovers more functions with a frame description entry (FDE) by parsing debugging sections; e.g., 13,380 such cases from `cpugcc_r-amd64-clang-O1`

- Recall Q3 → This may distort current metrics!
Insights and Conclusion

Insights

• State-of-the-art function detection tools work well for binaries without optimizations
• Not a single tool dominates all the others
• Difficult to claim an ML-centric approach surpasses rule-based ones
• The current metrics may not be reasonable in corner cases
• The capability of function identification relies on a tool’s strategic choice

Conclusion

• A function detection problem has yet been fully resolved, necessitating better dataset and metrics
Q&A