Accelerating Symbolic Analysis for Android Apps

Mingyue Yang, David Lie, Nicolas Papernot
University of Toronto
myshirley.yang@mail.utoronto.ca,
{david.lie, nicolas.papernot}@utoronto.ca
Motivation

- Symbolic execution challenge: **path explosion**
- **Unsatisfiable** paths: no input can trigger
- Require constraint collection & solving to find out satisfiability (expensive!)
Idea

- **Predict** satisfiability => skip potentially unsatisfiable paths
- **Criteria:**
  - Satisfiable **recall**: miss fewer (potentially malicious) satisfiable paths
  - Satisfiable **precision**: better speedup
  - Security-related analysis => **satisfiable recall** more important than **satisfiable precision**

- **Use** program features instead of constraint features
  - More info about code functionality
  - Save constraint collection time
Analysis Platform

- TIRO: symbolic analysis on Android apps
- Overall, only 29.8% of satisfiable paths (258,510 / 868,474)
  - From 127 out of 200 popular Google Playstore apps

**TABLE I**

<table>
<thead>
<tr>
<th>Path Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean (ms)</td>
</tr>
<tr>
<td>Unsatisfiable Path Analysis</td>
</tr>
<tr>
<td>Overall Path Finding</td>
</tr>
</tbody>
</table>

path analysis cost >> path finding cost
Approach

- Statistical features (specific to Jimple IR)
- Simple models: logistic regression & random forest
  - more complex model as future work (require more data; time-consuming to collect)
Approach: Statistical Features Used

**Method-level features:** complex execution => satisfiability

- **Control Flow**
  - # of If / Goto / LookupSwitch / TableSwitch
  - # of loops, # of statements in loop
  - # of returns / void returns

- **Method Invocation**
  - # of method invocation, # of methods invoked

- **Operation**
  - # of identity statements / assign statements
  - # of cast expressions
  - # of arithmetic / logical / shift / cmp operations

- **Program Size / Structure**
  - # of nop statements
  - # of blocks, # of units

**Variables & Expressions**

- # of defs / uses / locals
- # of new arrays, # of new expressions
- # of array references, # of arrays referred
- # of field references, # of fields referred
- # of length expressions
- # of class constants / concrete value constants / nulls

**Others**

- # of enter monitors / exit monitors
- # of throw statements

**Path-level features:**

- # of methods in path
- entry method type (if common)
- target method type (if common)

Complex data reference => hard for solve

- certain entry easier to reach
- certain target
Approach: Feature Vector Construction

Method-level features

- feature vector 1: [1, 3, 0, ..., 4]
- feature vector 2: [1, 0, 0, ..., 2]
- feature vector n: [0, 1, 0, ..., 3]
- sum of method features: [7, 12, 0, ..., 11]

Get statistical features:
- # of Ifs
- # of blocks

Path:

- method 1
- method 2
- ...
- method n

Path-level features

- # of methods
- entry method type
- target method type

Path feature vector: [7, 12, 0, ..., 11, 5, ..., 23]

satisfiable?

Model
Evaluation: Google Playstore apps

- Dataset: 127 out of 200 most popular apps in Google Playstore (July 2019)

- **All Apps**: Randomly split all paths into 5 groups & do 5-fold cross validation
  - Paths in training & test set could be from *same* app

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Model</th>
<th>Satisfiable Precision</th>
<th>Satisfiable Recall</th>
<th>Unsatisfiable Precision</th>
<th>Unsatisfiable Recall</th>
<th>Average Accuracy</th>
<th>Balanced Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Apps (Section IV-B)</td>
<td>LogReg</td>
<td>0.820</td>
<td>0.883</td>
<td>0.939</td>
<td>0.902</td>
<td>0.896</td>
<td>0.893</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.913</td>
<td>0.947</td>
<td>0.973</td>
<td>0.955</td>
<td>0.952</td>
<td>0.951</td>
</tr>
<tr>
<td>Cross-App (Section IV-C)</td>
<td>LogReg</td>
<td>0.743</td>
<td>0.895</td>
<td>0.949</td>
<td>0.858</td>
<td>0.872</td>
<td>0.877</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.751</td>
<td>0.914</td>
<td>0.956</td>
<td>0.864</td>
<td>0.880</td>
<td>0.889</td>
</tr>
</tbody>
</table>

- **Cross-App**: predict paths for *unseen* apps; more realistic scenario
  - Randomly split all apps into 5 groups for 5 runs
  - In each run, pick 1 group as test set & other groups as training set
- Random forest overfits for specific apps
- Satisfiable class: higher recall than precision (without tuning)
  - Higher satisfiable recall: miss fewer (potentially malicious) satisfiable paths
## Cross-app evaluation: different path types

Inspect paths used in previous cross-app validation (popular Google Playstore apps)

<table>
<thead>
<tr>
<th>Path Time</th>
<th>Num Paths</th>
<th>Max Save Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sat : Unsat</td>
<td>Total Paths</td>
<td>48 (0.00063%)</td>
</tr>
<tr>
<td></td>
<td>10ms-100ms</td>
<td>139032 (16.0%)</td>
<td>7092 (0.093%)</td>
</tr>
<tr>
<td>100ms-1s</td>
<td>334184 (38.5%)</td>
<td>122830 (1.61%)</td>
<td>130343 (1.71%)</td>
</tr>
<tr>
<td>1s-10s</td>
<td>241385 (27.8%)</td>
<td>368698 (4.84%)</td>
<td>888698 (11.7%)</td>
</tr>
<tr>
<td>10s-100s</td>
<td>130164 (15.0%)</td>
<td>270983 (3.56%)</td>
<td>3924772 (51.5%)</td>
</tr>
<tr>
<td>&gt;100s</td>
<td>11970 (1.4%)</td>
<td>279367 (3.67%)</td>
<td>2667137 (35.0%)</td>
</tr>
</tbody>
</table>
Cross-app evaluation: different path types

Inspect paths used in previous cross-app validation (popular Google Playstore apps)

For better satisfiable recall >10s: can increase overall confidence threshold

<table>
<thead>
<tr>
<th>Path Prediction Type</th>
<th>Sat Performance</th>
<th>Unsat Performance</th>
<th>Confidence</th>
<th>Total Paths</th>
<th>Saved Time</th>
<th>Max Save Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>Precision</td>
<td>Recall</td>
<td>Confidence</td>
<td></td>
</tr>
<tr>
<td>Path Time</td>
<td>Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;10ms</td>
<td>LogReg</td>
<td>0.463</td>
<td>1.000</td>
<td>0.999</td>
<td>0.239</td>
<td>0.826</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.543</td>
<td>0.994</td>
<td>0.991</td>
<td>0.452</td>
<td>0.828</td>
</tr>
<tr>
<td>10ms-100ms</td>
<td>LogReg</td>
<td>0.079</td>
<td>0.982</td>
<td>0.999</td>
<td>0.671</td>
<td>0.817</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.079</td>
<td>0.980</td>
<td>0.999</td>
<td>0.675</td>
<td>0.746</td>
</tr>
<tr>
<td>100ms-1s</td>
<td>LogReg</td>
<td>0.346</td>
<td>0.996</td>
<td>1.000</td>
<td>0.920</td>
<td>0.926</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.337</td>
<td>0.985</td>
<td>0.999</td>
<td>0.917</td>
<td>0.861</td>
</tr>
<tr>
<td>1s-10s</td>
<td>LogReg</td>
<td>0.958</td>
<td>0.933</td>
<td>0.949</td>
<td>0.968</td>
<td>0.921</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.950</td>
<td>0.956</td>
<td>0.966</td>
<td>0.961</td>
<td>0.889</td>
</tr>
<tr>
<td>10s-100s</td>
<td>LogReg</td>
<td>0.984</td>
<td>0.852</td>
<td>0.323</td>
<td>0.836</td>
<td>0.814</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.985</td>
<td>0.874</td>
<td>0.363</td>
<td>0.846</td>
<td>0.789</td>
</tr>
<tr>
<td>&gt;100s</td>
<td>LogReg</td>
<td>0.987</td>
<td>0.778</td>
<td>0.304</td>
<td>0.907</td>
<td>0.809</td>
</tr>
<tr>
<td></td>
<td>RandForest</td>
<td>0.977</td>
<td>0.786</td>
<td>0.291</td>
<td>0.825</td>
<td>0.745</td>
</tr>
</tbody>
</table>
Cross-application speedup

- Project speedup for Google Playstore apps

- Additional prediction overhead: feature extraction + model prediction
  - Method-level feature: save extracted features for encountered methods into hash-map; later retrieve (only retrieval overhead)

- Presented as proportion of total path analysis time
- Added overhead **negligible**

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction Time</th>
<th>Added Prediction Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogReg</td>
<td></td>
<td>0.021%</td>
</tr>
<tr>
<td>RandForest</td>
<td></td>
<td>0.022%</td>
</tr>
</tbody>
</table>
## Cross-application speedup

- Saved analysis time close to max achievable
- Some time-consuming paths are missed
- Adjust confidence threshold
  - Find points that balance missed path rate & saved analysis time

### Unsatisfiable Path Savings

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Model</th>
<th>Saved Analysis Time</th>
<th>Max Achievable Analysis Time</th>
<th>Saved Paths</th>
<th>Max Achievable Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>LogReg</td>
<td>13.9%</td>
<td></td>
<td>51.7%</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td></td>
<td>13.2%</td>
<td></td>
<td>47.3%</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td></td>
<td>12.1%</td>
<td></td>
<td>40.5%</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>RandForest</td>
<td>13.9%</td>
<td></td>
<td>51.2%</td>
<td>61.4%</td>
</tr>
<tr>
<td>0.6</td>
<td></td>
<td>13.0%</td>
<td></td>
<td>47.5%</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td></td>
<td>11.7%</td>
<td></td>
<td>43.0%</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td></td>
<td>7.4%</td>
<td></td>
<td>26.9%</td>
<td></td>
</tr>
</tbody>
</table>

### Satisfiable Path Savings

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Model</th>
<th>Missed Analysis Time</th>
<th>Missed Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>LogReg</td>
<td>13.8%</td>
<td>3.6%</td>
</tr>
<tr>
<td>0.7</td>
<td></td>
<td>6.8%</td>
<td>1.6%</td>
</tr>
<tr>
<td>0.9</td>
<td></td>
<td>1.9%</td>
<td>0.49%</td>
</tr>
<tr>
<td>0.5</td>
<td>RandForest</td>
<td>12.4%</td>
<td>3.1%</td>
</tr>
<tr>
<td>0.6</td>
<td></td>
<td>7.18%</td>
<td>1.8%</td>
</tr>
<tr>
<td>0.7</td>
<td></td>
<td>3.8%</td>
<td>0.97%</td>
</tr>
<tr>
<td>0.9</td>
<td></td>
<td>0.53%</td>
<td>0.17%</td>
</tr>
</tbody>
</table>
Cross-application speedup

Fig. 2. Sorted Percentage of Missed Paths per App (Logistic Regression)
Cross-app evaluation on malware

- Train on benign apps & test on malware (Genome dataset)
- Malware: important **not to miss paths** (potentially malicious) => require **high satisfiable recall**
  - 61% of malware paths are satisfiable

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Model</th>
<th>Satisfiable Precision</th>
<th>Satisfiable Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>LogReg RandForest</td>
<td>0.765</td>
<td>0.934</td>
</tr>
<tr>
<td>0.7</td>
<td>LogReg RandForest</td>
<td>0.714</td>
<td>0.976</td>
</tr>
<tr>
<td>0.9</td>
<td>LogReg RandForest</td>
<td>0.657</td>
<td>0.995</td>
</tr>
</tbody>
</table>
Conclusion

- **Reduce path explosion**: predict unsatisfiable paths with ML classifiers
- Use statistical features with **path-level** program analysis info

- Evaluation: TIRO deobfuscation tool for Android
- Able to generalize patterns about satisfiability to unseen apps & malware
- Saved analysis time close to max achievable save time
- Miss a small number of time-consuming satisfiable paths
  - Adjust confidence threshold: trade-off small amount of saved analysis time => reduce missed satisfiable paths
Thank you!
Related Work

- Use **constraint features** to predict best constraint solver / satisfiability: [DeepSolver, Path Constraint Classifier (PCC), SMTimer]
  - Still need to run constraint collection
  - We use **program features** before constraint collection: more info about code functionality
- Predict satisfying values for constraints that are difficult to solve: [NeuEx, MLB]
- Android program analysis tools: [IntelliDroid, AppIntent, AppAudit]
  - find paths statically and then dynamically execute them
  - also path explosion: can apply our technique
- Reduce path explosion: [Statsym, Fitnex, Mutation-based Validation Paradigm (MVP)]
  - Instead of filter out unsatisfiable ones, use path search algorithm to select paths matching specific objectives
References


References


