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

Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path

Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path

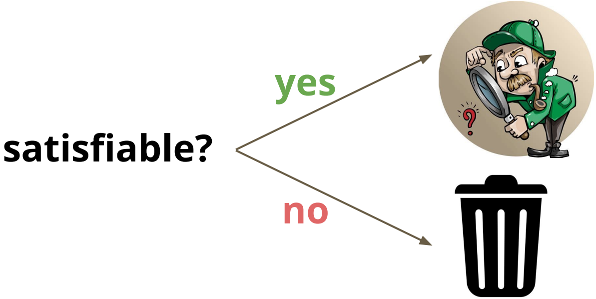
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path
Unsatisfiable path

No time to analyze :(

Satisfiable path

- 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

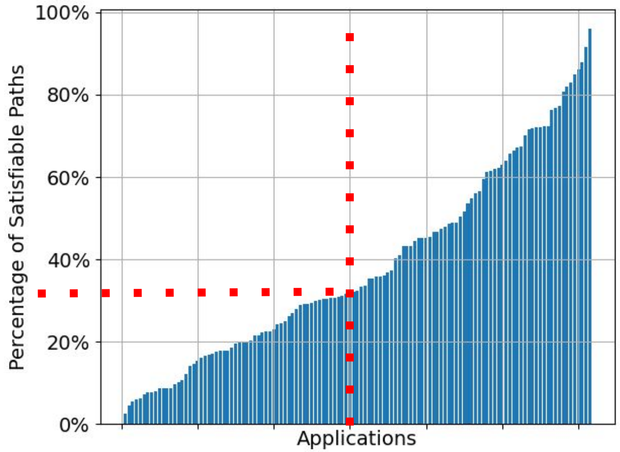
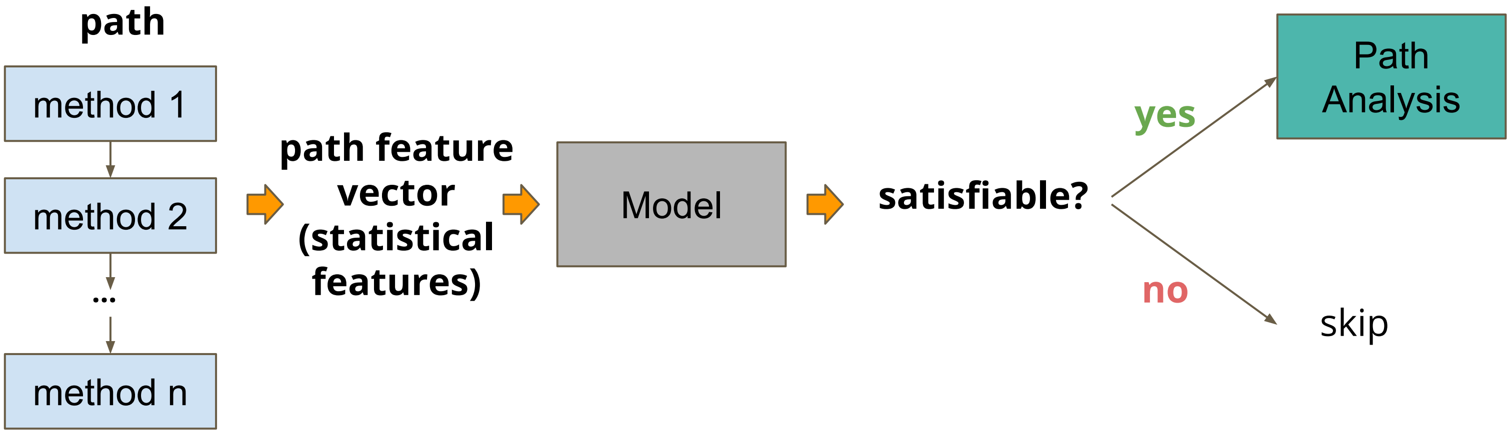


TABLE I
PATH PROCESSING TIME

	mean (ms)	std (ms)
Unsatisfiable Path Analysis	1720	15026
Overall Path Finding	32	107

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

Complex data reference => hard for solve

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 ← **program style**
# 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

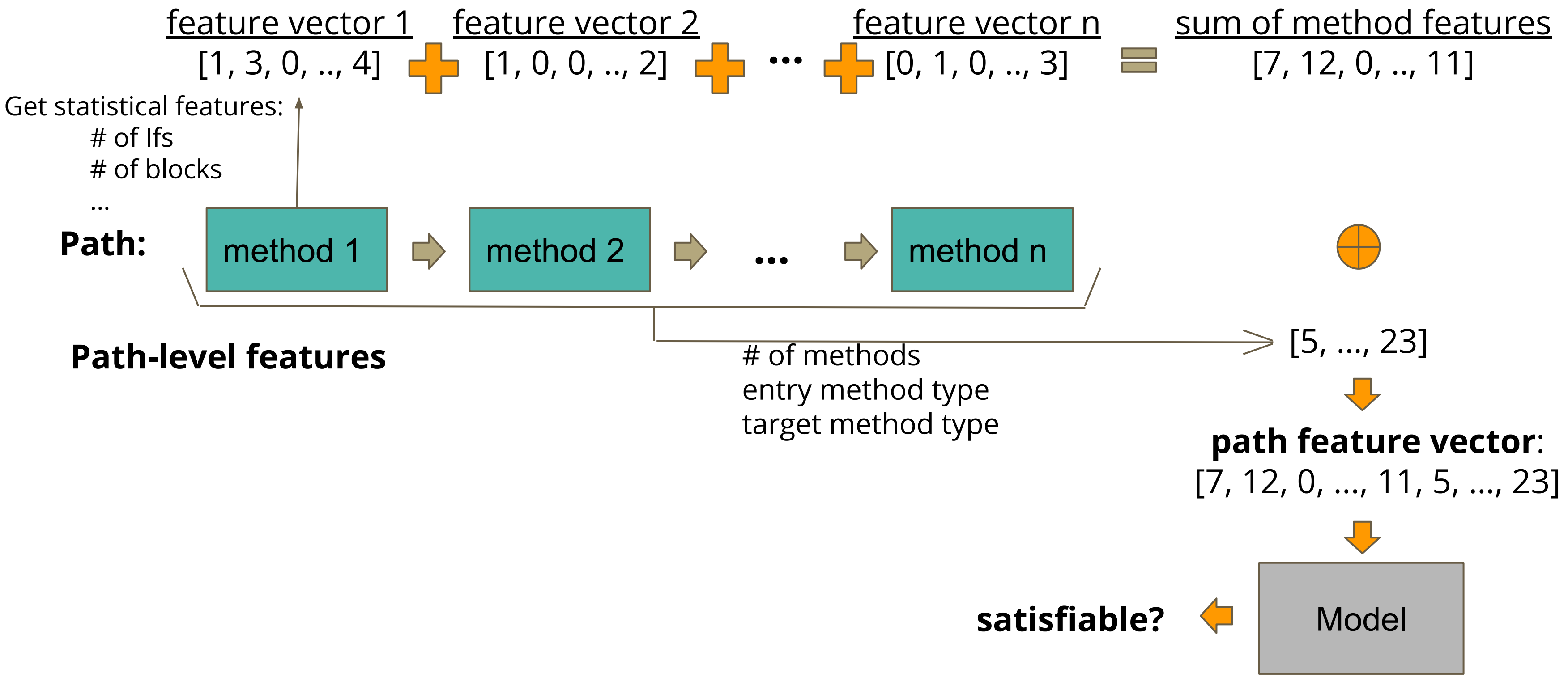
Path-level features:

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

certain entry easier to reach certain target

Approach: Feature Vector Construction

Method-level features



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

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

- **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: <u>miss fewer (potentially malicious</u>) satisfiable paths

Cross-app evaluation: different path types

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

Path Time	Num Paths		Max Save Time	Total Time
	Sat : Unsat	Total Paths		
<10ms	4651:7088 (0.66)	11739 (1.4%)	48 (0.00063%)	67 (0.00088%)
10ms-100ms	3857:135175 (0.029)	139032 (16.0%)	7092 (0.093%)	7231 (0.095%)
100ms-1s	13670:320514 (0.043)	334184 (38.5%)	122830 (1.61%)	130343 (1.71%)
1s-10s	105522:135863 (0.78)	241385 (27.8%)	368698 (4.84%)	888698 (11.7%)
10s-100s	119996:10168 (11.8)	130164 (15.0%)	270983 (3.56%)	3924772 (51.5%)
>100s	10814:1156 (9.4)	11970 (1.4%)	279367 (3.67%)	2667137 (35.0%)

Cross-app evaluation: different path types

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

Path Prediction Type		Sat Performance		Unsat Performance		Confidence			
Path Time	Model	Precision	Recall	Precision	Recall		Total Paths	Saved Time	Max Save Time
<10ms	LogReg	0.463	1.000	0.999	0.239	0.826	11739 (1.4%)	13.6 (0.00018%)	48 (0.00063%)
	RandForest	0.543	0.994	0.991	0.452	0.828		24 (0.00031%)	
10ms-100ms	LogReg	0.079	0.982	0.999	0.671	0.817	139032 (16.0%)	5110 (0.067%)	7092 (0.093%)
	RandForest	0.079	0.980	0.999	0.675	0.746		5148 (0.068%)	
100ms-1s	LogReg	0.346	0.996	1.000	0.920	0.926	334184 (38.5%)	115711 (1.52%)	122830 (1.61%)
	RandForest	0.337	0.985	0.999	0.917	0.861		115245 (1.51%)	
1s-10s	LogReg	0.958	0.933	0.949	0.968	0.921	241385 (27.8%)	356483 (4.68%)	368698 (4.84%)
	RandForest	0.950	0.956	0.966	0.961	0.889		352983 (4.63%)	
10s-100s	LogReg	0.984	0.852	0.323	0.836	0.814	130164 (15.0%)	209206 (2.75%)	270983 (3.56%)
	RandForest	0.985	0.874	0.363	0.846	0.789		213817 (2.81%)	
>100s	LogReg	0.987	0.778	0.304	0.907	0.809	11970 (1.4%)	240624 (3.16%)	279367 (3.67%)
	RandForest	0.977	0.786	0.291	0.825	0.745		221619 (2.91%)	

direction of increase

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

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**

Model	Prediction Time
	Added Prediction Overhead
LogReg	0.021%
RandForest	0.022%

Cross-application speedup

Threshold	Model	Unsatisfiable Path Savings			
		Saved Analysis Time	Max Achievable Analysis Time	Saved Paths	Max Achievable Paths
0.5	LogReg	**13.9%**	15.9%	**51.7%**	61.4%
0.7		13.2%		47.3%	
0.9		**12.1%**		**40.5%**	
0.5	RandForest	**13.9%**		**51.2%**	
0.6		13.0%		47.5%	
0.7		**11.7%**		**43.0%**	
0.9		7.4%		26.9%	

Threshold	Model	Satisfiable Path Savings	
		Missed Analysis Time	Missed Paths
0.5	LogReg	**13.8%**	**3.6%**
0.7		6.8%	1.6%
0.9		**1.9%**	**0.49%**
0.5	RandForest	**12.4%**	**3.1%**
0.6		7.18%	1.8%
0.7		**3.8%**	**0.97%**
0.9		0.53%	0.17%

- 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

Cross-application speedup

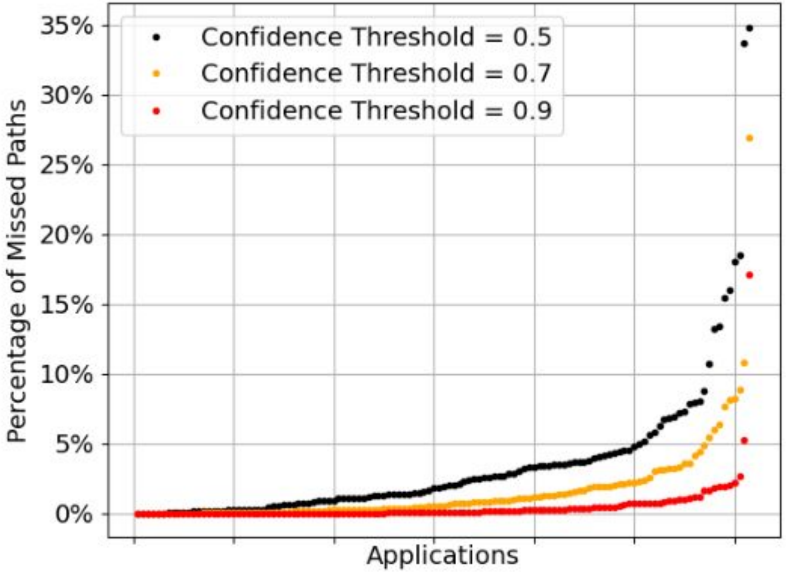


Fig. 2. Sorted Percentage of Missed Paths per App (Logistic Regression)

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 V
CROSS-APPLICATION PERFORMANCE ON MALWARE

Threshold	Model	Satisfiable Precision	Satisfiable Recall
0.5	LogReg	0.765	0.934
	RandForest	0.723	0.967
0.7	LogReg	0.714	0.976
	RandForest	0.647	0.998
0.9	LogReg	0.657	0.995
	RandForest	0.621	1

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

[1] Wong, Michelle Y., and David Lie. "Tackling runtime-based obfuscation in android with {TIRO}." *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018.

[2] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." *2012 IEEE symposium on security and privacy*. IEEE, 2012.

[3] Wen, Junye, et al. "Constraint Solving with Deep Learning for Symbolic Execution." *arXiv preprint arXiv:2003.08350* (2020).

[4] Wen, Sheng-Han, et al. "Enhancing symbolic execution by machine learning based solver selection." *Proceedings of the NDSS Workshop on Binary Analysis Research*. 2019.

[5] Luo, Sicheng, et al. "Boosting symbolic execution via constraint solving time prediction (experience paper)." *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021.

[6] Shen, Shiqi, et al. "Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints." *NDSS*. 2019.

[7] Li, Xin, et al. "Symbolic execution of complex program driven by machine learning based constraint solving." *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016.

References

[8] Wong, Michelle Y., and David Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." *NDSS*. Vol. 16. 2016.

[9] Yang, Zhemin, et al. "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013.

[10] Xia, Mingyuan, et al. "Effective real-time android application auditing." *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[11] Yao, Fan, et al. "Statsym: vulnerable path discovery through statistics-guided symbolic execution." *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017.

[12] Xie, Tao, et al. "Fitness-guided path exploration in dynamic symbolic execution." *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009.

[13] Kolchin, Alexander. "A novel algorithm for attacking path explosion in model-based test generation for data flow coverage." *2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC)*. IEEE, 2018.