Accelerating Symbolic Analysis for Android Apps

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

***Abstract*—While tools based on symbolic execution are commonly used to analyze mobile applications, these tools can suffer from path explosion when real-world applications have more paths than available computing resources can handle. However, many of the paths are *unsatisfiable*, that is, no input exists that can satisfy all the path constraints and cause the path to execute. Unfortunately, analysis tools cannot determine this without constraint collection and constraint solving, which are expensive to perform. As a result, analysis tools waste valuable computational resources on unsatisfiable paths.**

In this work, we demonstrate that machine learning classifiers can predict unsatisfiable paths, resulting in a savings of computational resources. Our classifiers take path-level statistical features as input, and model inference can run immediately after a path is found. This saves analysis time spent on both constraint collection and constraint solving for unsatisfiable paths. We enhance the TIRO Android application analysis tool to avoid paths that are predicted to be unsatisfiable and show that a Random Forest model can achieve 95% balanced predication accuracy in Android applications. We also show that modified TIRO is able to avoid analyzing 51% of paths as they are unsatisfiable, resulting in a savings of 14% of the analysis time.

I. Introduction

Symbolic execution is a powerful program analysis tool for mobile applications and has been demonstrated to be effective for detecting malware [9], deobfuscating applications code [10], privacy leakage analysis [13] and application auditing [11]. However, one challenge with symbolic execution is path explosion. Path explosion occurs because real-world applications have a large number of paths for symbolic execution to explore: analyzing all of them costs more time and computing power than allocated resources can handle.

We observe that in many cases, there exists no input that could satisfy the constraints of a given path. In other words, such program paths cannot be executed in reality, as no input can trigger it. Our measurements show that the vast majority of individual paths, roughly 70%, are themselves unsatisfiable. Unfortunately, trying to determine if a path is unsatisfiable requires constraint collection and solving collected constraint using a SAT-solver[1] [1], which are expensive to perform. In our experiments, it can take several seconds to find solution for a satisfying set of inputs on an average path. Collecting and solving constraints for unsatisfiable paths waste resources as they do not enable further symbolic exploration of the program, yet consume valuable analysis time.

In this work, we hypothesize that similarities between Android applications can be exploited to train a machine learning model that can predict with reasonable accuracy whether a path is satisfiable or not. Since a machine learning model can return predictions considerably faster than collecting and solving constraints for satisfiability, using the model to filter out paths will result in a speedup. While previous work has applied machine learning to speed up constraint solving [7], [8], [4], they predict over abstracted boolean constraints. In contrast, we predict over program features available earlier in the program analysis process, allowing greater time savings. We believe that program features may contain more information about code functionality than the derived constraints.

To evaluate our hypothesis, we conduct experiments using the TIRO [10] Android application analysis tool. We train several models and demonstrate that they can predict satisfiability both within applications, as well as, more importantly, across applications. Being able to predict across applications means that the models can predict accurately on applications they were not trained on, due to general similarities among Android applications. This allows our models to save analysis time for unseen applications. With only benign apps in the training set, the models are also able to generalize to malware, with high recall for satisfiable paths. This means paths that present malware behaviors are not likely to be missed.

II. Symbolic Analysis in Android

A. Analysis Platform

To study the cost of symbolic analysis on Android, we use TIRO [10], a deobfuscation tool that uses symbolic analysis to generate inputs that drive dynamic execution down paths that trigger code deobfuscation in Android applications. TIRO has a static analysis component that performs analysis on *call paths* leading to a target deobufscation location. A path is a sequence of method calls starting from an Android framework entry-point and ending in a target location. Such a call path may have constraints on its inputs that need to be satisfied to execute. In some cases, it may also depend on constraints that need to be satisfied by executing other *dependent paths*—for example, a heap variable may need to be set by executing another path first. In those cases, TIRO will also generate inputs for those dependent paths.

To execute call paths, TIRO also contains a dynamic analysis component that injects the inputs generated by the static component. If successful, the execution will trigger the appropriate code in the application to deobfuscate the obfuscated parts of the application. Additionally, for dynamic analysis, TIRO also instruments the analyzed application to collect deobfuscated information and discover more obfuscated code.

[1]We use SAT-solver, constraint-solver and theorem-prover interchangeably as they are all similar tools.

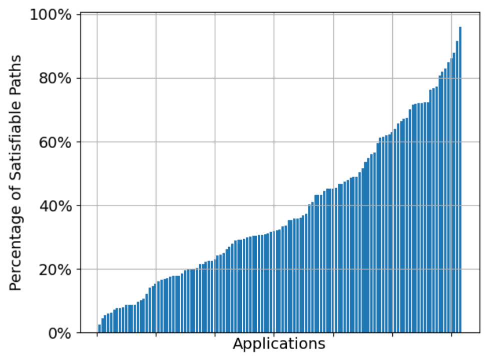

Fig. 1. Sorted Percentage of Satisfiable Paths per App

TIRO found that obfuscated code is widespread and present in the bulk of benign and malicious applications.

B. Application Path Properties

We collect 868,474 paths extracted from 127 popular play-store applications (all from the top 200 most popular apps in Google playstore 2019). Within these paths, TIRO finds only 258,510 (29.8%) satisfiable paths, while the remaining 609,964 paths are unsatisfiable. Although we find that on average it takes less time to analyze an unsatsifiable path, perhaps because the constraint solver can quickly identify a logical conflict, these unsatisfiable paths still consume 15.9% of analysis time on average for each application.

Figure 1 presents the percentage of satisfiable paths per app (for 123 apps each with at least 125 paths), sorted in ascending order. This percentage varies broadly across applications, ranging from 2.6% (android.apps.docs.editors.slides) to 95.5% (zxing.client.android). The distribution of the percentage of satisfiable paths across apps is roughly uniform, with the median percentage around 33%, indicating that there are more unsatisfiable paths in general.

C. Path Processing Costs

We measure the cost to process satisfiable paths and unsatisfiable paths. TIRO processes a path in two phases. First, a *path finding* phase traverses the app's call graph to identify paths to targeted locations where deobfuscation code might exist. Then a *path analysis* phase collects constraints and attempts to solve them using the Z3 [1] constraint solver. Table I shows the average and standard deviation of running times that TIRO took to find and analyze paths. We find that the analysis time for unsatisfiable paths is several magnitudes larger than path finding. Indeed, the constraint generation/solving process for TIRO is much more time-consuming compared to path finding.

Analysis time spent on unsatisfiable paths does not contribute to the goals of the application analysis since the paths can never execute in reality. However, as path-finding time is relatively cheap compared to analysis time, this suggests that if we may find a way to predict whether a path is satisfiable from information that can be collected after path finding, then we may save the expensive path analysis phase.

TABLE I
PATH PROCESSING TIME

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

III. APPROACH

For hybrid and dynamic analysis tools such as TIRO, it is often the case that there are far more paths in an application that can be realistically analyzed in a finite amount of time. As a result, it is beneficial to focus analysis time on paths that are satisfiable, that is, executable with some set of inputs.

To this end, we deploy machine learning on the program path information available immediately after TIRO's path finding phase, to predict whether TIRO's constraint analysis process can successfully generate satisfying inputs or not. This allows TIRO to discard paths that are likely to be unsatisfiable and focus its resources on call paths that are more likely to have inputs that can satisfy the path constraints. If the machine learning model is faster than TIRO's path analysis phase and has reasonable accuracy, it will improve TIRO's analysis speed and enable it to analyze more application paths within the same amount of time.

We conduct our experiments with simple statistical models, and leave the exploration of more complex machine learning models, such as those involving deep-learning for future work. Deep-learning generally requires larger amounts of labeled training data, and our data can only be labeled by running TIRO in full, which is time consuming. A simple model allows us to evaluate our hypothesis in a shorter amount of time. A positive result with a simple model suggests that refinement with a more complex model with access to control- and data-flow should yield even greater analysis time improvements. As a result, we currently use a logistic regression model and a random forest model to classify samples.

We note that in TIRO, path constraints are not collected until the path analysis phase. Thus the model only has available to it basic program analysis path information. Since TIRO relies on the Soot analysis framework [6], which uses the Jimple IR, we use Jimple-specific path features as predictors for path satisfiability. For each path, we first collect method-level feature vectors for every method in the path. We then add up all method-level feature vectors to be a path-level feature vector. Path-level features (# of methods in path, entry method, target method) are then appended at the end of the path feature vector for the path sample. Below lists details of all features:

Control Flow Statement: Control flow statements include If, Goto, LookupSwitch and TableSwitch. These statements add diverging paths in code: the complexity of code is thus increased and it is more likely to result in conflicting constraints that make the path unsatisfiable

Loops and Statements in Loops: Loops also add complexity by significantly increasing the number of possible execution paths. We also include the number of statements in the loop as a feature to indicate the loop size.

Return Statement: Return statements interrupt the control flow by exiting the current method. Returning early in method body may result in simpler paths that are more likely to be satsifiable. However, a large number of returns in a method may result in complex, unsatisfiable constraints. We track both return statements with and without return values.

Method Invocation: Method invocation indicates more code will be executed other than the statements in the method body itself. This increases the probability that there will be greater complexity in the code. We currently do not include the features of called methods. Instead, the number of invocations and the number of methods invoked are tracked separately. This distinguishes the case when a method is invoked several times from the case when several methods are invoked.

Identity Statement: In Jimple, Identity statements are used to assign "this" references and parameters to locals. More Identity statements means there are more variables used as input / output for the current method. This may also add complexity to the code and thus the generated constraints.

Assign Statement: The number of assign statements may be related to data flow complexity and are thus included.

Cast Expression: Cast expressions indicate existence of different types in code, and may reveal information about code style/functionality along with other features.

Arithmetic / Logical / Shift / Comparison Operation: These operations adds computation in the method, which may be included as part of a constraint.

Nop Statement: While nop statements themselves have no effect, they could be correlated to other statements and may imply some information like code alignment specific to the application/path.

Block and Unit: Blocks represent number of basic blocks in a method. Number of blocks and units would reflect both method size and complexity of control flow.

Def, Use, and Local: We also keep track of the number of defs, uses and locals in method body, as they may complicate data dependencies in code.

New Array/Expression: New arrays / expressions are special types of objects that can be more complex than ordinary variables and may complicate the resulting constraint.

Array/Field Reference: As an array can contain multiple elements, reference to the same array may or may not refer to the same element. This adds more loads to the constraint solver. A field in class may be updated by methods that do not exist in current path and result in more dependencies that need to be satisfied. The number of references and arrays/fields referred are used as separate features, to take into account cases when a single array/field is used multiple times.

Length Expression: The length expression for arrays could be variable, and may not be obvious to determine. This could result in complication in code and increase loads to the constraint solver.

Class Constant, Concrete Value Constant, and Null: Usage of constants may denote special usages and functionality in code. We specifically separate class type constant and null from other constant values.

Enter/Exit Monitor: Enter/exit monitor could be correlated. Existence of only enter/exit may imply dependency paths or invalid path.

Throw Statement: Existence of throw statements denote possibility of error in code. The conditions that result in errors may not be easily captured.

Number of Methods in Path: More methods mean more constraints need to be satisfied in order to reach a deep call.

Common Entry/Target Method: As different entry methods and target methods vary by functionality, certain entry methods are more/less likely to reach certain target methods. For these features, we find common entry / target methods that are used more than 500 times as a entry / target method among the 868,474 paths we collected. This left us with a list of 81 entry methods and 89 target methods as features. Recorded entry / target methods are indexed starting from 1. For entry / target methods that are not in the list, we use the value 0 to denote it is not a common entry / target method.

IV. Evaluation

In this section, we evaluate how our proposed models perform on real-world applications. We use both cross-validation on the same set of apps and evaluation across different apps. The recall on malware samples is also checked to ensure malicious paths are not incorrectly predicted as unsatisfiable, which would cause TIRO to skip analyzing them.

A. Dataset Collection

To collect path samples for evaluation, we run TIRO on 127 apps among the 200 most popular apps in Google Playstore from July 2019. Entrypoint preprocessing is already done as a separate job before the analysis. For the analysis run, we set a timeout to be 24 hours for TIRO. For each app, we use 4 threads and 240G of memory in total. While some apps run out of memory during analysis, we still keep paths from all analysis runs in our dataset to get more samples. In total, we collected 868,474 paths from the 127 applications. On average, there are 31.6 methods for each path.

B. Performance on All Applications

In this section, we evaluate the performance of the statistical models on the dataset we collected. We use a 5-fold cross-validation by randomly splitting paths from all applications into 5 groups. In this experiment, the training set and test set may contain paths from the same application. As shown in Table II, the models are able to learn patterns that predict satisfiability. The simple logistic regression model yields considerable accuracy for both classes, and the more complex random forest has better performance than logistic regression.

TABLE II
PERFORMANCE ON GOOGLE PLAYSTORE APPLICATIONS

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

C. Cross-Application Evaluation

More realistically, our machine learning models need to make predictions for paths from unseen applications. Thus, we randomly split all applications into 5 groups for 5 different runs. For each run, we choose one corresponding group as the test set and the other groups as the training set: the paths in training set and test set are thus from different applications. In the training set, for paths with duplicate feature vectors and same label (likely duplicate paths), only one such sample is kept to avoid overfitting. In the test set, however, we keep the original data to model realistic scenarios for model usage: paths with the duplicate feature vectors are kept as separate samples. The averaged results for 5 runs are shown in Table II.

Overall, the results from the two models are similar. The models achieve considerable balanced accuracy for both classes despite of class imbalance, showing ability to generalize to unseen applications. The recall for the satisfiable class is around 90%, indicating that the likelihood to miss satisfiable paths is reasonable. Moreover, without any tuning, we found that both models naturally have higher recall rather than precision in the satisfiable class. This is beneficial, as higher recall ensures fewer satisfiable paths are missed. While the precision for the satisfiable class is not as high, the model is still able to eliminate a number of unsatisfiable paths and improve performance.

For cross-application evaluation, the more complex random forest model has only a small advantage over the simple logistic regression model. This is different from the case when paths from the same application exist in both training set and test set, in which the more complex random forest model have a large advantage over simple logistic regression. This suggests that the random forest model overfits for specific applications, with little improvement for generalized patterns across applications.

D. Cross-Application Evaluation for Different Path Types

We further inspect how the models perform on paths with different analysis time across applications. This allows us to get a brief overview of how the models act differently on different types of paths and how much path analysis time may be saved. We group all path samples from the test sets in Section IV-C based on their analysis time, and evaluate the performance in each group using the same models trained in Section IV-C. The results are shown in Table III.

The number of and the types of paths in each group varies. The unsatisfiable paths mostly concentrate in the 10ms-100ms group and the 100ms-1s group, while the satisfiable paths are mostly in the 1s-10s group and the 10s-100s group. In each group, there exists a different class imbalance for the satisfiable and unsatisfiable paths. Along with the types of paths, the total analysis time TIRO spends on each group is also different. The analysis time is mainly spent on paths that take more than 1s. The 15% of paths in the 10s-100s group takes up more than half of the total analysis time. The 1.4% of paths that take more than 100s uses 35% of the total analysis time. While the satisfiability and analysis time is different in each group, most savings come from paths that have analysis time >1s. As the recall for unsatisfiable class increases, the saved time also approaches the theoretical maximum.

For paths that take less than 1s to analyze, the models have high recall on the satisfiable class, missing few satisfiable ones. Although the precision for satisfiable paths is low, this would not affect the speed of the analysis much, as these groups in total only cost less than 2% of total analysis time.

As paths become more expensive to analyze, the recall for the satisfiable class degrades, while the precision for satisfiable paths increases. For paths that take more than 100s to finish, the recall for satisfiable paths is around 0.78. This results in some time-consuming satisfiable paths being missed. However, we generally would want to bias the predictor to avoid skipping satisfiable paths as this may miss behavior the analysis is intended to detect. As the confidence for the long running paths is not as high, we configure TIRO to skip paths only when the predicted unsatisfiable path is above some confidence threshold. By adjusting the threshold, one may achieve considerable saved analysis time, while reducing the number of missed satisfiable paths.

E. Cross-Application Model Speedup

To show the effect of our models on TIRO's analysis time and accuracy, we estimate the cost of the feature extraction and model inference, the saved analysis time for unsatisfiable paths and percentage of missed satisfiable paths.

As the original TIRO tool does not extract features required for model prediction, extra computation is required to extract the feature vectors: this consists of feature extraction time and model prediction time. For feature extraction, we measure the cost for each method separately. The feature extraction time is different for every method, ranging from 0.12ms to 796ms. On average, the feature extraction time per method is 3.5ms. The feature extraction cost for a path is the sum of cost for all methods in the path. While not currently implemented, the feature extraction cost can be reduced by caching the features of each method in a hash map, so that extraction is only performed once per method. We estimate the cost of insertion into the hash map as 0.03ms and the cost to lookup and retrieve a method's features from the hash map as 0.025ms.

TABLE III
CROSS-APPLICATION PERFORMANCE ON DIFFERENT TYPES OF PATHS

Path Prediction Type		Sat Performance		Unsat Performance		Confidence	Num Paths		Analysis Time (sec)		
Path Time	Model	Precision	Recall	Precision	Recall		Sat : Unsat	Total Paths	Saved Time	Max Save Time	Total Time
<10ms	LogReg	0.463	1.000	0.999	0.239	0.826	4651:7088 (0.66)	11739 (1.4%)	13.6 (0.00018%)	48 (0.00063%)	67 (0.00088%)
	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	3857:135175 (0.029)	139032 (16.0%)	5110 (0.067%)	7092 (0.093%)	7231 (0.095%)
	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	13670:320514 (0.043)	334184 (38.5%)	115711 (1.52%)	122830 (1.61%)	130343 (1.71%)
	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	105522:135863 (0.78)	241385 (27.8%)	356483 (4.68%)	368698 (4.84%)	888698 (11.7%)
	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	119996:10168 (11.8)	130164 (15.0%)	209206 (2.75%)	270983 (3.56%)	3924772 (51.5%)
	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	10814:1156 (9.4)	11970 (1.4%)	240624 (3.16%)	279367 (3.67%)	2667137 (35.0%)
	RandForest	0.977	0.786	0.291	0.825	0.745			221619 (2.91%)		

TABLE IV
PERCENTAGE OF AVERAGE PREDICTION SPEEDUP ACROSS APPLICATIONS

Threshold	Model	Prediction Time	Unsatisfiable Path Savings				Satisfiable Path Savings	
		Added Prediction Overhead	Saved Analysis Time	Max Achievable Analysis Time	Saved Paths	Max Achievable Paths	Missed Analysis Time	Missed Paths
0.5	LogReg	0.021%	**13.9%**	15.9%	**51.7%**	61.4%	**13.8%**	**3.6%**
0.7			13.2%		47.3%		6.8%	1.6%
0.9			**12.1%**		**40.5%**		**1.9%**	**0.49%**
0.5	RandForest	0.022%	**13.9%**		**51.2%**		**12.4%**	**3.1%**
0.6			13.0%		47.5%		7.18%	1.8%
0.7			**11.7%**		**43.0%**		**3.8%**	**0.97%**
0.9			7.4%		26.9%		0.53%	0.17%

The model prediction time per path instance is 0.013ms for logistic regression and 0.078ms for random forest. The cost of feature extraction, insertion, retrieval and model prediction are summed into an overall *prediction time* to represent the additional cost of our approach.

In order to reduce missed satisfiable paths, we also set a confidence threshold as mentioned in Section IV-D: TIRO will only skip a path when it is both predicted as unsatisfiable and the confidence of prediction is greater than the selected threshold. We calculate saved/missed analysis time as a percentage of the overall analysis time per app. The saved/missed paths are also calculated as a percentage of all paths in the app. To ensure results are not dominated by a few apps with large number of paths or analysis time, we calculate the result per app and average the result for all used apps. To avoid result being affected by apps with very few paths, we only keep 123 apps in Section IV-A that have at least 125 paths.

Table IV shows the projected speedup at different thresholds. With a default threshold of 0.5, the models save around 13.9% of analysis time, close to the max achievable analysis savings: 15.9%. A small number of missed paths (3.6% / 3.1%) take up 13.8% / 12.4% analysis time respectively. This is consistent with results in Table III, as the model performance degrades for the small groups of time-consuming paths.

With higher confidence threshold, the reduction in missed paths outweighs the reduction in saved analysis time. As shown in Table IV, the logistic regression model has a 3.6% to 0.49% reduction in the number of missed paths Vs 13.9% to 12.1% in saved analysis time. This shows that conservatively using a high threshold allows TIRO to greatly reduce the number of missed paths while still saving a considerable amount of analysis time. The random forest model permits lower confidence thresholds than logistic regression as its average confidence values are lower for most types of paths.

The percentage of missed paths also varies across apps as shown in Figure 2. The missed percentage of paths is generally low for most apps except for a few outliers at the right. As the confidence threshold increases, the missed percentage of paths consistently decreases for all applications.

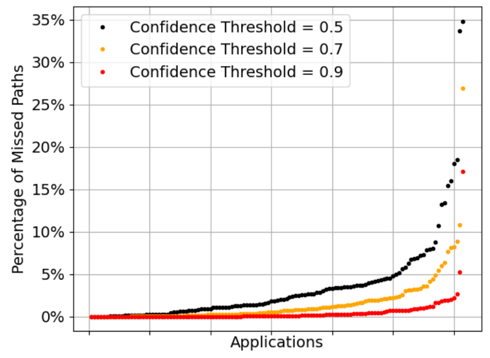

Fig. 2. Sorted Percentage of Missed Paths per App (Logistic Regression)

F. Malware Performance

In this section, we evaluate the use case of cross-application evaluation on malware. Paths in malwares have only 4.1 methods on average. In the malware dataset, 61% of the paths are satisfiable, much more than the proportion in ordinary apps. However, with malware, it is important not to miss paths as they may cause TIRO to miss malicious behavior, which as a result, motivates good recall for satisfiable paths.

We collect 470,471 paths from 1,260 malware samples in the Android malware Genome dataset [15]. We use 8 threads and 240G of memory for analysis. The timeout is set to 6 hours, while most analysis jobs finish before timeout. We classify these paths using models trained with all paths from benign apps in Section IV-A—no malware paths exist in the training set.

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

Table V demonstrates the ability of our models to generalize to malware. With a default confidence threshold of 0.5, the logistic regression model and the random forest model achieve 93.4% and 96.7% recall for the satisfiable class respectively. This demonstrates that our models are able to achieve cross-application recall on malware that is comparable to that on benign applications. With higher confidence values, the satisfiable recall further increases to be nearly perfect.

V. RELATED WORK

Tools such as [7], [8], [4] use machine learning to predict time for constraint solving or constraint satisfiability in symbolic execution. They may be used to discard infeasible symbolic states or select best constraint solver. The main difference is that these tools predict over constraints instead of path features. Thus, they examine statistical constraint features like the number of nodes, constraint structures and variables or convert constraints to matrix representations, while our work uses basic program-analysis information from program paths. In addition, previous works require constraints to be collected, while our models are able to predict without constraints and thus can save constraint collection time.

There also exists work [5], [3] that uses machine learning to predict satisfying values for constraints that are difficult for constraint solver to solve. These models attempt to solve constraints when symbolic execution gets stuck, instead of predicting whether constraints may be satisfied.

There are existing Android program analysis tools [9], [13], [11] that find sensitive program paths statically and execute the paths dynamically to verify them. These tools are used for various purposes such as malware detection, privacy leakage detection, app auditing. [9] and [13] statically finds for input for the path and dynamically injects the input to execute it. As [11] uses dynamic taint analysis and approximated execution, it does not need to solve for input statically. Although these analysis tools are different, they also have invalid program paths and may experience path explosion. In future, our proposed approach may also be experimented on these tools.

To reduce path explosion, existing work [14], [12], [2] improves search algorithms that construct paths. As opposed to generically filtering infeasible paths, they take an analysis-specific approach of guiding the symbolic execution only towards paths that are more likely to match their objectives.

VI. CONCLUSION

To reduce path explosion, we use machine learning classifiers to predict unsatisfiable paths that do not yield meaningful results. Our models take statistical features with basic program analysis information at path level. We evaluate them on the Android program analysis tool TIRO. This technique is able to capture patterns for satisfiable paths and generalize to unseen applications including both benign apps and malware. On average, the saved analysis time we achieve is close to max achievable save time if all unsatisfiable paths are known beforehand. Model inference adds negligible time overhead and only misses a small percentage of time-consuming satisfiable paths. By adjusting the confidence threshold for prediction, one can also tradeoff a small amount of saved analysis time to reduce missed long-running satisfiable paths.

REFERENCES

[1] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[2] A. Kolchin, "A novel algorithm for attacking path explosion in model-based test generation for data flow coverage," in *2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC)*. IEEE, 2018, pp. 1–5.

[3] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, "Symbolic execution of complex program driven by machine learning based constraint solving," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 554–559.

[4] S. Luo, H. Xu, Y. Bi, X. Wang, and Y. Zhou, "Boosting symbolic execution via constraint solving time prediction (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 336–347.

[5] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints." in *NDSS*, 2019.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*. IBM, 1999, p. 13.

[7] J. Wen, M. Khan, M. Che, Y. Yan, and G. Yang, "Constraint solving with deep learning for symbolic execution," *arXiv preprint arXiv:2003.08350*, 2020.

[8] S.-H. Wen, W.-L. Mow, W.-N. Chen, C.-Y. Wang, and H.-C. Hsiao, "Enhancing symbolic execution by machine learning based solver selection," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[9] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware." in *NDSS*, vol. 16, 2016, pp. 21–24.

[10] ——, "Tackling runtime-based obfuscation in Android with TIRO," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)*. USENIX Association, 2018, pp. 1247–1262.

[11] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time Android application auditing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015, pp. 899–914.

[12] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 359–368.

[13] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013, p. 1043–1054.

[14] F. Yao, Y. Li, Y. Chen, H. Xue, T. Lan, and G. Venkataramani, "Statsym: vulnerable path discovery through statistics-guided symbolic execution," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 109–120.

[15] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.