Enhancing Symbolic Execution by Machine Learning Based Solver Selection

Sheng-Han Wen*, Wei-Loon Mow*, Wei-Ning Chen*, Chien-Yuan Wang* and Hsu-Chun Hsiao*†
*Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
†Research Center for IT Innovation, Academia Sinica, Taiwan

Abstract—**Constraint solving creates a serious performance bottleneck in symbolic execution. Examining a plethora of SMT solvers with diverse capabilities, we address the following research questions: How can the performance of symbolic execution improve if it can pick a priori the best solver for a given path constraint? How can such a prediction oracle be practically implemented? In this work, we first define the solver selection problem in symbolic execution and its evaluation metrics, and perform a preliminary study to gauge potential performance improvement through solver selection. We then present the design and implementation of Path Constraint Classifier (PCC), a machine learning based meta-solver that aims to reduce overall constraint solving latency by dynamically selecting a solver per query. The use of using machine learning seems straightforward, yet surprisingly underexplored; one main technical challenge is how to avoid excessive overhead introduced by feature extraction. We address this challenge by taking advantage of the structural characteristics of symbolic execution. Our experiments confirm that the overall solver time can be reduced by 10.3% in the KLEE dataset and 46% in the benchmark dataset, while the solver prediction procedure only accounts for 2% to 10% of overall solving time.**

I. Introduction

Symbolic execution is an automated program analysis technique for software testing and vulnerability discovery. Symbolic execution systematically explores possible program execution paths, represents each path as a set of constraints, and sends the constraints to a *solver* for satisfiability checks and input generation.

Despite being a key enabler of symbolic execution, constraint solving is a computationally intractable problem in theory and remains a serious performance bottleneck in practice. Palikareva and Cadar [27] showed that solvers can use up to 99% of total time in symbolic execution. The performance may further deteriorate when advanced symbolic execution techniques, such as state merging, increase the constraint complexity. Kuznetsov et al. [21] showed that merging multiple states may end up increasing the overall solving time as the size of the symbolic path constraints increases.

Because state-of-the-art SMT solvers address intractability via various optimizations, they have different capabilities and solving times. No existing solver can consistently outperform the others, as shown in our preliminary study in Section III-A. To take advantage of multiple solvers with diverse capabilities, several symbolic execution engines support multiple solvers in parallel. For example, KLEE [10] and JDART [25] embed the pySMT framework [15] and the JConstraints interface, respectively, as a unified interface for solvers. In addition, because solver selection is a type of algorithm selection problem, several portfolio-solving techniques have been applied to select solvers [19], [34]. However, because the time spent on constraint processing and prediction was not included in their evaluation, it remains unclear how effective these solutions are when applied to improve the overall performance of symbolic execution.

In this work, we define the *solver selection problem in symbolic execution*: Given a set of solvers and a sequence of path constraints generated by a symbolic execution engine, we would like to have a prediction oracle that selects a solver for each path constraint, such that the overall solving and prediction time is minimized. An interesting difference between the generic solver selection problem and the one in symbolic execution is the path constraints collected by the symbolic execution engine have high similarity with each other. This is because when the symbolic execution engine explores to a certain state with path constraint $\{c_1, c_2, ..., c_n\}$, the engine can choose to explore the next state with constraint c_{n+1} or its negation c'_{n+1}. Choosing either will share a set of similar constraints except the last one.

To approximate this optimization problem, we propose Path Constraint Classifier (PCC), a system that automatically selects a solver by utilizing machine learning techniques. PCC predicts which solver can perform well when receiving a path constraint. The use of machine learning seems straightforward, yet is surprisingly underexplored, with one main technical challenge being how to avoid excessive overhead introduced by machine learning. We address this challenge by taking advantage of the domain knowledge of symbolic execution. Specifically, to accelerate feature extraction, we identify a set of features that can be quickly extracted from path constraints collected during symbolic execution, and propose using a Constraint Feature Cache Table (CFCT) to drastically reduce the feature extraction time by leveraging the structural characteristics of symbolic execution.

In the usage of CFCT, the feature extraction procedure time has 28x to 788x speedup in the SE dataset and 11x in the

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23xxx
www.ndss-symposium.org

benchmark dataset. Our system can perform better than any SMT solver in the KLEE and benchmark datasets, which has the least overall solving time including extra solver prediction and feature extraction overhead. Our system manage to achieve 1.12x and 1.85x speedup in the KLEE and benchmark datasets respectively, and has the most solve rate in the KLEE and benchmark datasets.

Because `PCC` can be considered as a meta-solver, it can be integrated with other optimization techniques (e.g., constraint simplification, query reduction, and improved solvers) to further improve the scalability of constraint solving in symbolic execution.

The contributions of this paper are listed below:

- We compare the performance of different solvers over three constraint datasets and show the amount of time reduction one might gain by employing solver selection.
- We define the solver selection problem in symbolic execution and identify the fundamental distinctions between it and the generic solver selection problem.
- We design and implement a system called Path Constraint Classifier based on machine learning techniques, with domain-specific optimizations to accelerate the extraction of path constraint features.
- We discuss several future directions to further improve PCC, including the use of backup solvers, logic selection, delayed solving, and dynamic timeout.

II. Background and Related Work

Symbolic execution is a well-known software testing technique. Given a program, a symbolic execution engine systematically explores it and generates *path constraints* that represent program states using symbolic variables. During exploration, the symbolic execution engine heavily relies on *Satisfiability Modulo Theories (SMT) solvers*, such as Z3, for two important tasks: (1) checking the satisfiability of a path constraint; (2) obtaining concrete input values that can be used to reach the corresponding state of a path constraint.

Able to generate concrete inputs for software crashes, symbolic execution is an effective method of software testing and has been applied to small-scale industry applications and for academic usage recently. However, when it comes to large-scale applications, many scalability issues arise, limiting the practicability of symbolic execution. One of the performance bottlenecks is constraint solving. An experiment [27] about the time spent constraint solving shows that the constraint solving time without any optimization could account for up to 99% of the total time of symbolic execution.

Several methods have been proposed to speed up constraint solving by reducing the number of constraints [10], [17], reusing proofs [1], [2], [10], [20], [32], simplifying complex constraints (e.g., nonlinear constraints) [16], [29], or searching for approximate solutions to complex constraints [7], [13], [22], [31]. In addition, specialized constraint solvers are proposed to better support specific operations or library functions. For example, Z3-str [35] treats strings as a primitive type to efficiently solve strings-related theories. CORAL [31] utilizes meta-heuristic search algorithms to support theories with transcendential functions such as trigonometric and logarithmic functions. Our solution can be integrated with these optimization techniques to further improve the scalability of constraint solving in symbolic execution.

As solvers have diverse capabilities, several proposals explore the idea of querying different solvers for different types of constraints. KLEE [10] supports a multi-solver framework using pySMT. However, querying all the supported solvers in parallel is resource-consuming. Hence, our work aims to predict the best solver for a given path constraint and only queries this solver for improved performance. Pasareanu et al. [28] split a path constraint into simple and complex parts. The solutions of the simple part will be reused to help simplify the complex one. This technique has been improved by Hybrid-KLEE [23], which proposes to divide a path constraint into linear and nonlinear parts. The solutions of the linear parts will be treated as initial seeds of a local search algorithm to solve the complex parts. Instead of having pre-defined classifications, our work proposes an automated constraint classification procedure based on machine learning.

SATzilla [34] considers SAT solver selection as an algorithm selection problem and adopts existing portfolio-solving techniques. Healy et al. [19] apply similar techniques to the SMT solving domain. Our work focuses on how to apply portfolio-solving in the symbolic execution domain, and proposes domain-specific optimizations to accelerate the extraction of path constraint features.

III. Solver Selection Problem in Symbolic Execution

A. Preliminary Study

We first conduct an experiment to show that no solver can consistently outperform the others and simple classification based on constraint logic is insufficient. This motivates the development of a more sophisticated solver selection method in symbolic execution.

a) Data: We use the SMT-LIB standard [5], [6] as a common format to express constraints. SMT-LIB is well documented and is compatible with most of the current symbolic execution engines and SMT solvers. We obtain constraints with various logic types from the SMT-COMP benchmark [3]. A logic in SMT-LIB can be viewed as a depiction of the expression language, including the theory (e.g., linear or nonlinear arithmetic), the primitive types (e.g.., integers, reals or bitvector) and corresponding operations, and also the limitations between the conjunction of primitives. We also use constraint data collected by running two symbolic execution engines, KLEE and angr (the SE dataset). See Section V-A for details.

b) Solvers: We choose five popular SMT solvers from SMT-COMP, which are Z3 [12], MathSAT [11], CVC4 [4], Yices [14], and Boolector [8]. For each constraint, we query each of the five solvers and identify the best solver that uses the least solving time.

c) Results: TABLE I and TABLE II show that even though `Boolector` seems to be the best solver for the SE dataset, `Yices` requires the least solving time to solve all the data. We can conclude that `Boolector` might struggle to solve particular constraints, which increases the total solving time. On the other hand, though `Yices` is the fastest solver for 5%-37% of constraints, it is stable (i.e., few timeouts and errors) and uses the least total solving time.

SE dataset	Yices	CVC4	Z3	Boolector	MathSAT
angr	8.56%	0.00%	0.31%	56.37%	34.76%
KLEE	36.77%	0.00%	0.00%	62.48%	0.75%

TABLE I: The ratio of being the best solver in the SE dataset

SE dataset	Yices	CVC4	Z3	Boolector	MathSAT
angr	227s	1304s	1721s	579s	337s
KLEE	338s	2888s	1092s	477s	813s

TABLE II: Total time to solve all path constraints in the SE dataset

TABLE III shows the ratio the best solvers and the breakdowns based on logic type of the SMT-COMP benchmark dataset. The result confirms that no existing solver can consistently outperform the others, and thus it is possible to improve overall performance via careful solver selection. In addition, while logic type is an important factor affecting solvers' performance, there is still no clear winner in the logic types commonly seen in symbolic execution (e.g., QF_ABV) [9]. Hence, instead of relying on a single solver, we are motivated to combine the strength of multiple solvers.

B. Problem Definition

We now define the solver selection problem in symbolic execution. Compared to the generic solver selection problem, this formulation takes into account the time overhead of the selection procedure and the sequence of path constraints is generated by a symbolic execution engine.

Solver Selection Problem in Symbolic Execution. Given a sequence of M path constraints $PC = \{pc_1, pc_2, ..., pc_M\}$ and a set of N constraint solvers $S = \{s_1, s_2, ..., s_N\}$, we formulate the solver selection problem in symbolic execution as finding $b_i \in S$ such that $T(S, PC)$, which is the total solving time including solver selection time and solver solving time, is minimized.

$$T(S, PC) = \sum_{i=1}^{M}(T_{solve}(b_i, pc_i) + T_{determine}(pc_i)) \quad (1)$$

where T_{solve} stands for the time spent on solving path constraint pc using solver s, and $T_{determine}$ stands for the time spent on determining the best SMT solver for a given path constraint.

Note that this formulation does not include one-time costs such as initialization or training time in ML.

Theory Logic	Yices	CVC4	Z3	Boolector	MathSAT
ALIA	0.00%	0.00%	100.00%	0.00%	0.00%
AUFLIRA	0.00%	0.00%	100.00%	0.00%	0.00%
AUFNIRA	0.00%	9.71%	90.29%	0.00%	0.00%
BV	21.43%	71.43%	7.14%	0.00%	0.00%
LIA	100.00%	0.00%	0.00%	0.00%	0.00%
LRA	42.11%	14.04%	43.86%	0.00%	0.00%
NIA	0.00%	0.00%	100.00%	0.00%	0.00%
NRA	0.00%	1.11%	98.89%	0.00%	0.00%
QF_ABV	22.72%	0.95%	10.57%	43.73%	22.03%
QF_ALIA	40.00%	0.00%	20.00%	0.00%	40.00%
QF_AUFBV	33.33%	0.00%	0.00%	33.33%	33.33%
QF_AUFLIA	47.66%	19.63%	19.63%	0.00%	13.08%
QF_AX	100.00%	0.00%	0.00%	0.00%	0.00%
QF_BV	44.83%	3.45%	31.03%	13.79%	6.90%
QF_IDL	89.29%	0.00%	3.57%	0.00%	7.14%
QF_LIA	66.67%	1.26%	14.38%	0.00%	17.69%
QF_LRA	52.76%	0.00%	4.91%	0.00%	42.33%
QF_NIA	88.32%	0.99%	10.50%	0.00%	0.20%
QF_NRA	47.73%	6.38%	45.89%	0.00%	0.00%
QF_RDL	92.59%	0.00%	7.41%	0.00%	0.00%
QF_UF	99.86%	0.14%	0.00%	0.00%	0.00%
QF_UFBV	33.33%	0.00%	33.33%	33.33%	0.00%
QF_UFIDL	67.86%	0.00%	28.57%	0.00%	3.57%
QF_UFLIA	71.19%	0.00%	20.34%	0.00%	8.47%
QF_UFLRA	76.03%	0.00%	12.40%	0.00%	11.57%
QF_UFNRA	42.86%	0.00%	57.14%	0.00%	0.00%
UF	0.00%	100.00%	0.00%	0.00%	0.00%
UFBV	0.00%	0.00%	100.00%	0.00%	0.00%
UFIDL	0.00%	33.33%	66.67%	0.00%	0.00%
UFLIA	0.00%	39.17%	60.83%	0.00%	0.00%
UFLRA	0.00%	100.00%	0.00%	0.00%	0.00%
Overall	31.82%	8.53%	37.43%	6.46%	5.31%

TABLE III: The best solver count percentage in different theory logic

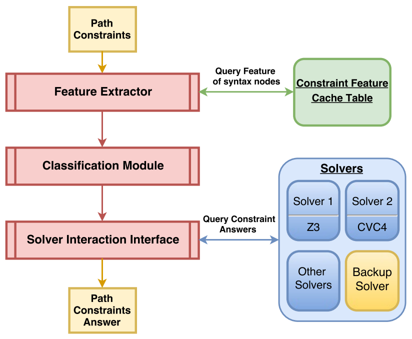


Fig. 1: The architecture of Path Constraint Classifier

IV. PCC Design and Implementation

To address the solver selection problem in symbolic execution, we propose a new symbolic execution component: *Path Constraint Classifier (PCC)*. PCC leverages machine learning techniques to predict the performance of each solver when solving a given path constraint, and selects the solver with the best predicted performance.

We implement PCC with an embedded solver interaction interface so that the selected solver can be queried directly according to the classification result. Therefore, one can also view PCC as a special kind of SMT solver that can be flexibly integrated with other symbolic execution engines.

Figure 1 presents an overview of PCC's internal architecture and a flowchart of how a path constraint is classified to a solver. PCC is composed of three major components: Feature Extractor, Classification Module and Solver Interaction Interface. We also propose a special component called Constraint Feature Cache Table in Feature Extractor to further improve the performance of PCC. The main classification workflow is as follows:

- **Feature Extractor:** Given a new incoming path constraint, the Feature Extractor will generate features for the path constraint in cooperation with Constraint Feature Cache Table, which stores all the generated features in a table in order to speed up the generation of identical features, and pass the features to Classification Module.
- **Classification Module:** The classification module will perform a classification algorithm to determine the most suitable solver for a given path constraint, and then query the chosen solver through the Solver Interaction Interface.
- **Solver Interaction Interface:** When the chosen solver finishes its task, the interaction interface will help transfer and output the result to the upper layer component.

In the following subsection, we describe the design and implementation of Path Constraint Classifier in details, including the features, extraction method and optimization, and finally, the classification algorithm.

A. Feature Extraction

Recall that in order to evaluate the performance of different SMT solvers, we use the SMT-LIB standard as a common expression of a path constraint. Therefore, to extract the features of a given path constraint, we first express it in the SMT-LIB form. This conversion is supported by most symbolic execution engines. Next, the path constraint is parsed into a syntax tree, with every internal node representing an operator and every leaf representing an operand (i.e., symbol or constant). An example of a syntax tree is shown in Figure 2.

B. Feature Selection

To capture the characteristics of a path constraint, we select 119 features for solver prediction. These features can be roughly classified into four categories:

- **Syntax Node Statistics Features** are the statistics of syntax node properties, including the appearance of arithmetic operators such as *add*, *minus*, *multiply* and *divide*; bitvector operators such as *bv-add*, *bv-minus*, *bv-and* and *bv-or*; quantifiers *forall* and *exists*. There are about 62 features in this category.
- **Tree Structure Features** contain the structure properties of a tree, including the depth of a tree, the number of leaves and nodes.

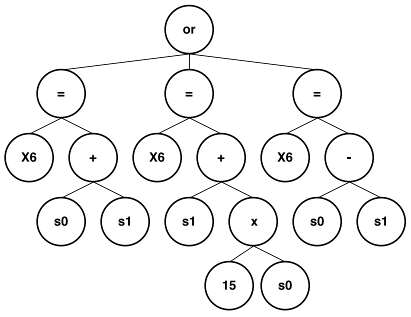


Fig. 2: Syntax tree for constraint: (assert (or (= X6 (+ S0 S1)) (= X6 (+ (* 15 S0) S1)) (= X6 (- S0 S1)))

- **Variable Features** are the features related to variables, such as the number of variables and unique variables, and also the number of variable clauses such as *AND* and *OR*.
- **Logic Features** is a 50-dimension vector composed of only Boolean values, which indicates the types of logic the path constraint belongs to.

The majority of the chosen features are related to **constraint size**, as a larger constraint tends to cause higher burden on a solver. In addition, these features can be easily extracted, reducing overhead imposed on the solver selection procedure. An example of an easily extracted feature is the *depth* feature, as all we have to do is traverse all the syntax sub-trees and find the one with the biggest depth value, where the time complexity is $O(\#ofsubtrees)$.

C. Constraint Feature Cache Table

In symbolic execution, every path constraint of a state and its child states shares a large proportion of identical constraints. More precisely, the only difference will be the last constraint of its child state. Due to this, we are likely to waste excessive time extracting the same features and thus slow down the performance of symbolic execution.

To accelerate the feature extraction procedure, we design a special component called **Constraint Feature Cache Table (CFCT)**. The idea is based on hash-consing [18]. Before extracting features from a syntax tree, we first query whether the features of a given syntax tree have already been stored in CFCT. If so, we directly take the stored features. Otherwise, we extract the required properties from its root and apply the same procedure to all the syntax sub-trees, then store it in CFCT for future use. To show the performance of CFCT, we conduct an experiment to compare extraction times with and without the usage of CFCT, as shown in Section V-B.

D. Classification Model

We use **Deep Neural Network (DNN)** with nine dense layers as our classification model, which is a lightweight machine learning model whose classification procedure takes negligible time. Given the features of a path constraint, the model will predict the probability for every SMT solver being the most suitable solver to solve it. This probability indicates which solver is least likely to result in losses among all the other solvers. For example, if the probability of the `z3` solver is 1.0, then `z3` will result in the least amount of loss when solving a given path constraint. The loss function can be flexibly defined depending on the optimization objective.

We design two versions of DNN model with different loss functions and heuristics in mind.

- **DNN-Alpha**: In the first version of the model, our heuristic is very simple: if we can accurately predict the fastest solver, then the total solving time could be minimized. Hence, in this model, we only use the fastest solver as the answer to each problem and train the model with *categorical cross entropy loss*.
- **DNN-Beta**: In the second version of the model, we try to take solvability and solver performance into consideration. After all, in symbolic execution, choosing a solver that fails to solve a path constraint is worse than choosing a slow one that successfully solves it. Hence, we give penalty to those that fail to solve due to timeout or error, and assign a loss value based on the time of a successful solve:

$$Loss(Solver) = \begin{cases} A * Norm(T_{Solver}), & \text{if } \textit{solved} \\ B, & \text{if } \textit{timeout} \\ C, & \text{if } \textit{error} \end{cases}$$

$$Norm(T) = \frac{T - T_{min}}{T_{max} - T_{min}}$$

$Norm(.)$ is a function that normalizes the solving time T_{Solver} of $Solver$ according to T_{max} and T_{min}, which are the maximal and minimal values of the solving time collected from all the solvers that successfully solve this problem. A, B and C are adjustable parameters to evaluate the outcome of a solving procedure, which can be determined by one's preference for each solving outcome. For example, suppose our preference order for solving outcomes are $\{$*Successfully Solve*$\}$ $>$ $\{$*Timeout*$\}$ $>$ $\{$*Error*$\}$, which means "it is better for a solver to *Successfully Solves* a problem than to *Timeout*". Therefore, according to our preference, a possible assignment for A, B and C could be 50, 100 and 200.

V. Evaluation

A. Data Collection

We use two datasets for experiments: *SE* and *benchmark*. The SE dataset is collected from two symbolic execution engines: angr [30] and KLEE [10]. Specifically, we perform symbolic execution on eight target binaries from `GNU Coreutils 8.29` and dump all the path constraint queries into a `.smt2` file. To increase diversity, we also collect the *benchmark dataset* from SMT-COMP [3], which contains constraints of several different logics type stored in the `.smt2` file format.

The benchmark dataset and the SE dataset contain $8,199$ and $59,073$ files, respectively. The SE dataset contains $39,776$ files generated by KLEE and $19,297$ files generated by angr.

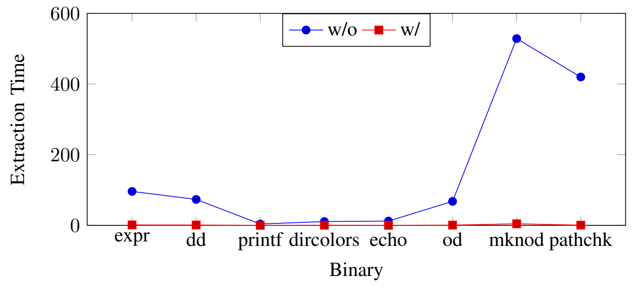


Fig. 3: Feature extraction time of different binaries with or without Feature Cache

B. Constraint Feature Cache

To evaluate the effectiveness of using the constraint feature cache table, we compare the extraction times between PCC with and without the usage of CFCT.

For each extraction procedure, we set the time limit to 20 seconds and record the time spent on extraction as well as the number of extracted syntax nodes. The result is shown in Figure 3 (more details in Table IV Appendix A). The result in the `RANDOM` column is an average of 10 iterations of 500 randomly sampled `.smt2` files from the benchmark dataset.

In this experiment, we use the SE dataset collected from KLEE. According to he experiment results, using CFCT greatly speeds up the extraction procedure for both types of data. Additionally, the SE dataset enjoys a higher speedup ratio because the SE dataset contains a higher percentage of duplicate constraints than the benchmark dataset.

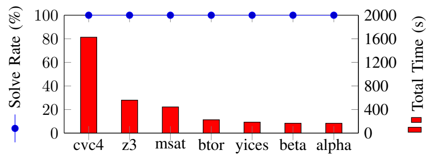
At 788x faster, the speedup factor for binary pathchk is much higher than other binaries. A possible explanation for this could be that some of the path constraints in `pathchk` contain a huge number of constraints, but most are identical. Consequently, a large amount of time can be saved by using CFCT, demonstrating yet again that CFCT can effectively improve the efficiency of feature extraction.

For the benchmark dataset, where all data have relatively low dependency compared to the SE dataset, the usage of CFCT results in 11x speedup. This shows that CFCT can greatly speed up the feature extraction procedure for all kinds of datasets.

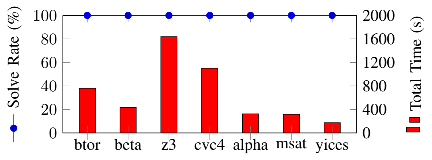
C. Solving Performance

Training Data Generation. We use five SMT solvers to solve all the constraints with a given timeout (100 seconds in the current experiment) and record the corresponding outcome, including the solving time, accuracy of the answer, and whether an error or timeout event occurs. As for model training, we split our data into training and testing datasets as follows:

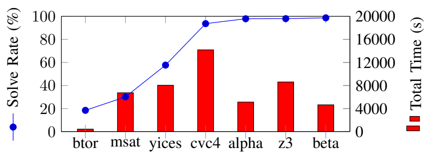
- **Benchmark dataset**: We randomly choose 60% of them to be our training dataset, and the remainder is our testing dataset.
- **SE dataset**: We divide the binaries into two groups, one for generating training data and the other reserved for testing.



(a) KLEE



(b) angr



(c) benchmark

Fig. 4: Solving performance of every solver in different datasets

The reason for splitting the SE dataset in this manner is to reduce the chance of information leaks during the training phase. Since a path constraint could share a large proportion of identical constraints with another path constraint in the same binary, randomly splitting the SE dataset could mean that the training dataset contains knowledge about the testing dataset, thus making prediction results unreliable.

Experiment Settings. We train our models and compare the prediction results of them directly using an SMT solver. Five solvers are evaluated: cvc4, z3, msat, btor, and yices. In addition, the `best` solver represents the ideal selection that always correctly predicts the fastest solver, with zero prediction overhead, for solving a given path constraint.

Experiment Results. The experiment results are shown in Figure 4 (more details in TABLE V in Appendix B). Based on the results, the performance of our DNN models is the closest to the `best` solver with respect to both the solve rate and the total time in the KLEE and benchmark datasets.

In the KLEE dataset, almost all of the SMT solvers have a 100% solve rate. Our DNN models perform better than other SMT solvers in terms of total time, even with additional feature extraction and prediction overhead.

In the benchmark dataset, the DNN-beta model has the highest solve rate compared to other solvers. Despite the fact that `btor` requires the least total time, it can only solve about 20% of the data, which may be intolerable as this could prevent the symbolic execution engine from exploring further paths if most of the constraints are unsolved. `z3` seems to be the best solver among the five SMT solvers, because `z3` can solve almost 98% of data within a reasonable time. However, our DNN-beta results in about 2x speedup and can solve almost 99% of data.

In the angr dataset, our DNN models do not perform well but the performance is still tolerable compared with other solvers like `btor`, `cvc4` and `z3`, which result in 1.8x to 5x performance speedup. We believe that increasing the size of the angr dataset can improve the performance of our DNN model.

Notice that in the SE dataset, the performance of DNN-alpha is better than DNN-beta. This is because all the solvers we tested can solve all symbolic data, and the factors of `timeout` and `error` do not matter much. Thus, choosing the fastest solver will be the best decision. On the other hand, in the benchmark dataset, the number of `timeouts` and `errors` becomes an important factor for solver selection. DNN-beta can make better decisions in choosing the *fastest and solvable* solver than choosing the *fastest but unsolvable* solver.

VI. Conclusion and Future Work

In this work, we present Path Constraint Classifier (`PCC`), a new component in symbolic execution engines, which aims to improve the efficiency of constraint solving by predicting the fastest solver for a given path constraint. We first conduct a preliminary study on modern SMT solvers to demonstrate the need for solver selection. Next, we define the solver selection problem in symbolic execution and propose a machine learning based solution with several optimizations to tackle this problem. `PCC` transforms path constraints into a syntax tree to extract features, and uses Deep Neural Network (DNN) with two self-designed loss functions to predict the best solver in different scenarios. We also propose *constraint feature cache* to greatly reduce the overhead in feature extraction. Finally, we evaluate the solving performance of `PCC` and demonstrate that it can achieve better performance than individual SMT solvers, thus improving the efficiency of constraint solving.

Several interesting research directions remain for future exploration. As our solution is based on machine learning, one future work is to perform feature engineering and model tuning (e.g., a better loss function) to increase the robustness of prediction results while reducing the overhead cost of the determination procedure. To improve the model, we would also like to collect more symbolic data (e.g., path constraints in the depth of a program) for training to fully recognize the ability of SMT solvers.

Another interesting direction is examining how to combine `PCC` with other solver optimization techniques, including constraint simplification, query reduction, and improved solvers, and evaluating the performance. It is also important to note that when using PCC, the symbolic execution engines may frequently switch between different solvers. This might cause frequent context switching that affects the overall performance of symbolic execution.

Finally, existing symbolic execution engines mostly treat solvers as a blackbox. It would be interesting to see whether and how a symbolic execution engine can be more tightly integrated with solvers. For example, the symbolic execution engine could dynamically determine how to encode a path condition [21] such that the constraints can be solved quickly by solvers.

References

[1] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 305–315. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771802

[2] A. Aquino, G. Denaro, and M. Pezzè, "Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 427–437. [Online]. Available: https://doi.org/10.1109/ICSE.2017.46

[3] Barrett, C. de Moura, L. Stump, and Aaron, "Smt-comp: Satisfiability modulo theories competition," in *Computer Aided Verification*, Etessami, K. Rajamani, and S. K., Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 20–23.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032319

[5] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2016. [Online]. Available: www.SMT-LIB.org

[6] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[7] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu, "Symbolic execution with interval solving and meta-heuristic search," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 111–120. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.91

[8] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, ser. TACAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 174–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00768-2_16

[9] S. Bucur, "Encoding Symbolic Expressions as Efficient Solver Queries," 2015. [Online]. Available: http://dslab.epfl.ch/blog/2015/07/26/encoding-symbolic-expressions.html

[10] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[11] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 93–107. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_7

[12] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[13] P. Dinges and G. Agha, "Solving complex path conditions through heuristic search on induced polytopes," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 425–436. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635889

[14] B. Dutertre, "Yicesä2.2," in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[15] M. Gario, A. Micheli, and F. B. Kessler, "Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms," in *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories SMT*, 2015.

[16] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[17] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2094081

[18] E. Goto, "Monocopy and associative algorithms in an extended lisp," 1974.

[19] A. Healy, R. Monahan, and J. F. Power, "Predicting SMT solver performance for software verification," in *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016.*, 2016, pp. 20–37. [Online]. Available: https://doi.org/10.4204/EPTCS.240.2

[20] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 177–187. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771806

[21] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254088

[22] 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 *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 554–559. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970364

[23] M. Lin, X. Hou, R. Liu, and L. Ge, "Enhancing constraint based test generation by local search," in *Proceedings of the 6th International Conference on Software and Computer Applications*, ser. ICSCA '17. New York, NY, USA: ACM, 2017, pp. 154–158. [Online]. Available: http://doi.acm.org/10.1145/3056662.3056672

[24] H. Liu, E. R. Dougherty, J. G. Dy, K. Torkkola, E. Tuv, H. Peng, C. Ding, F. Long, M. Berens, H. Liu, L. Parsons, Z. Zhao, L. Yu, and G. Forman, "Evolving feature selection," *IEEE Intelligent Systems*, vol. 20, no. 6, pp. 64–76, Nov 2005.

[25] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman, "Jdart: A dynamic symbolic analysis framework," in *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 442–459. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49674-9_26

[26] I. Mierswa and K. Morik, "Automatic feature extraction for classifying audio data," *Mach. Learn.*, vol. 58, no. 2-3, pp. 127–149, Feb. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10994-005-5824-7

[27] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Proceedings of the 25th International Conference on Computer Aided Verification*, ser. CAV'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 53–68. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_3

[28] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 34–44. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001425

[29] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[31] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "Coral: Solving complex constraints for symbolic pathfinder," in *Proceedings*

of the Third International Conference on NASA Formal Methods, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 359–374. [Online]. Available: http://dl.acm.org/citation.cfm?id=1986308.1986337

[32] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393665

[33] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl, "Automatic feature selection in neuroevolution," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1225–1232. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068210

[34] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for sat," *J. Artif. Int. Res.*, vol. 32, no. 1, pp. 565–606, Jun. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622673.1622687

[35] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 114–124. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491456

Appendix A
Feature Cache Evaluation

Refer to TABLE IV.

Appendix B
Performance of SMT solvers and PCC model

Refer to TABLE V.

Target Binary	**expr**	**dd**	**printf**	**dircolors**	**echo**	**od**	**mknod**	**pathchk**	**RANDOM**
Num. of SMT files	3881	1013	122	135	433	1512	5226	703	500
Total syntax nodes	1.1×10^6	902361	47178	131403	149765	823947	6.5×10^6	2.4×10^{39}	4.2×10^{62}
					Without Feature Cache				
Extracted nodes	1.1×10^6	902361	47178	131403	149765	823947	6.5×10^6	2.5×10^7	1.2×10^6
Extraction time (s)	96.04	73.41	3.79	10.68	12.06	67.84	528.55	419.83	999.83
Timeout	0	0	0	0	0	0	0	20	43
					With Feature Cache				
Extracted nodes	10104	5119	447	3479	724	5549	34438	5592	6.5×10^5
Extraction time (s)	1.11	0.90	0.04	0.37	0.07	0.64	4.40	0.53	93.32
Timeout	0	0	0	0	0	0	0	0	0.6
Cache size (KB)	786.71	196.88	49.43	196.88	49.43	786.71	3146.00	786.71	50331.92
Speedup factor	86.22	81.13	83.54	28.73	152.98	105.73	120.01	788.41	11.77

TABLE IV: Feature extracting time with and without the usage of feature cache

Solver	**Solve Rate**	**Number of Timeout**	**Number of Error**	**Solving Time**	**Timeout Time**	**Error Time**	**Feature Extraction Time**	**Predict Time**	**Total Time**
					KLEE				
best	100.00%	0	0	137.70	0.00	0.00	-	-	137.70
DNN-alpha	100.00%	0	0	150.09	0.00	0.00	14.54	2.28	166.91
DNN-beta	100.00%	0	0	150.61	0.00	0.00	14.54	2.23	167.38
yices	100.00%	0	0	186.18	0.00	0.00	-	-	186.18
btor	100.00%	0	0	226.26	0.00	0.00	-	-	226.26
msat	100.00%	0	0	444.24	0.00	0.00	-	-	444.24
z3	100.00%	0	0	559.31	0.00	0.00	-	-	559.31
cvc4	99.99%	1	0	1526.81	100.01	0.00	-	-	1626.82
					ANGR				
best	100.00%	0	0	117.17	0.00	0.00	-	-	117.17
yices	100.00%	0	0	174.37	0.00	0.00	-	-	174.37
msat	100.00%	0	0	318.17	0.00	0.00	-	-	318.17
DNN-alpha	100.00%	0	0	318.47	0.00	0.00	5.10	1.14	324.71
DNN-beta	99.99%	1	0	324.96	102.02	0.00	5.10	1.11	433.19
btor	99.98%	2	0	557.09	204.04	0.00	-	-	761.13
cvc4	100.00%	0	0	1101.88	0.00	0.00	-	-	1101.88
z3	100.00%	0	0	1638.07	0.00	0.00	-	-	1638.07
					BENCHMARK				
btor	18.49%	1	2600	335.23	102.41	1.54	-	-	439.18
best	100.00%	0	0	883.04	0.00	0.00	-	-	883.04
DNN-beta	98.65%	30	12	1445.6	3102.72	0.21	101.88	0.36	4650.77
DNN-alpha	97.87%	34	34	1505.95	3442.35	80.10	101.88	0.36	5130.64
msat	30.15%	34	2195	3265.41	3470.19	4.98	-	-	6740.58
yices	57.69%	68	1282	1241.03	6801.32	6.23	-	-	8048.58
z3	97.96%	62	3	2146.23	6386.44	79.81	-	-	8612.48
cvc4	93.61%	98	106	4364.52	9811.21	3.22	-	-	14178.95

TABLE V: The performance of different SMT solvers and our PCC models at benchmark, angr and KLEE dataset