# Optimal Conjunctive Normal Form Encoding for Symbolic Execution


1st Weiyu Pan*
*College of Computer*
*National University of Defense Technology*
Changsha, China
panweiyu@nudt.edu.cn

2nd Ziqi Shuai
*College of Computer*
*National University of Defense Technology*
Changsha, China
szq@nudt.edu.cn

3rd Ke Ma
*School of Computer Science*
*WuHan University*
Wuhan, China
2017302580024@whu.edu.cn

4th Luyao Liu
*College of Computer Science and Electronic Engineering*
*Hunan University*
Changsha, China
liuluyao@hnu.edu.cn



***Abstract*—Constraint solving is a key challenge in symbolic execution. Usually, symbolic execution uses the fixed-size bit-vector theory to precisely model the program's behavior and generates the bit-vector formula to query the SMT solver. To solve such bit-vector formula, SMT solvers usually adopt a bit-blasting and conjunctive normal form (CNF) conversion step, transforming the original formula into a equi-satisfiable CNF formula, and then check the formula's satisfiability. However, the different CNF conversions can significantly affect the efficiency of SAT solving. We observe that each CNF encoding algorithm has its suitable applications, while adopting a specific CNF conversion algorithm for all formulas is often not optimal. Therefore, we propose to intelligently select a suitable CNF encoding algorithm for each logical formula. We have integrated our selection algorithm into the symbolic execution framework based on KLEE and STP, which are the state-of-the-art symbolic execution engine for C programs and its default underlying constraint solver, respectively. The experimental results, based on extensive evaluation of 86 real-world C programs in Coreutils benchmark, indicate that our method can effectively improve the efficiency of symbolic execution. On average, our method increases the number of the explored paths by 27.2%.**



***Index Terms*—CNF, SAT, Machine learning, Symbolic execution**


## I. INTRODUCTION

Symbolic execution [8], [11] is a widely used program analysis technique to systematically explore the path space of programs. Its applications covers many fields of software engineering, including automated test generation, software verification and bug detection. Symbolic execution is processing on symbolic inputs instead of concrete inputs. Therefore, the operations in program are recorded as the computation between symbolic expressions. For each program path, symbolic execution maintains a path condition ($PC$) that is updated whenever a branch instruction is encountered. Only if current branch is reachable is the corresponding path condition updated. Otherwise, the branch is unreachable and symbolic execution terminates the exploration. Note that the feasibility of a program path is determined by the result of constraint solving, *i.e.*, determining whether the path condition which is a quantifier-free first-order logic formula [13] is satiable. In this way, symbolic execution can explore the path space of the program systematically and understand the program precisely. Due to these advantages, many successful symbolic execution engines emerge, such as KLEE [3], Pex [23], and SPF [18], to name a few.

Obviously, constraint solving is a critical component of symbolic execution, as it is used to check the feasibility of a path and generate the test to execute the corresponding path if feasible. However, there exists many obstacles for constraint solving, which further limits the development of symbolic execution. On the one hand, the number of paths to be explored grows exponentially with the increase of program size and some syntactic constructs like loops can even lead to infinite paths. Therefore, symbolic execution engines will issue a bulk of queries to the underlying solver for complex programs. On the other hand, because of complex features in real world programs, *i.e.*, non-linear arithmetic and array operation, symbolic execution engines will build complex queries which are quite hard to solve. In brief, constraint solving is the most time-consuming part and limits the scalability of symbolic execution.

In general, symbolic execution uses bit-vector arithmetic SMT theory combining with other SMT theories (*e.g.*, array theory) to precisely model the behavior of program. When solving the bit-vector formula, bit-blasting is a key step in most SMT solvers which reduces a bit-vector formula into a pure propositional SAT formula. Unfortunately, such SAT formula won't be solved by SAT solvers immediately. Modern SAT solvers [7] mainly take the input as a conjunctive normal form (CNF) formula in which the solver is able to apply highly efficient solving algorithms. Consequently, SMT solvers have to convert the SAT formula after bit-blasting into an equi-satisfiable CNF formula, which can be efficiently solved by


*Corresponding author


DOI reference number:10.18293 / SEKE2021-113

SAT solver.

Currently, there are two common CNF transformation algorithms: Tseitin algorithm [25] and the algorithm based on technology mapping [6]. The former adds a new variable to each logic gate of the original formula, and then constrains the variable with a new clause to form a new CNF. This algorithm has a lower complexity, but the generated CNF is often huge and difficult to solve. The latter divides AIG (And Inverter Graph) into logical nodes wherein there is no more than K inputs for each node, and extracts CNF for each node based on a look-up table. The algorithm has a higher complexity, but it can generate CNF that is more concise and easier to solve. We have the following key observation of CNF conversion in SMT solvers: almost every SMT solver of QF_BV logic always uses one of the specific CNF conversion algorithms above. However, each CNF conversion algorithm has its suitable applications, and the efficiency of using a specific CNF conversion algorithm as the solution of all formulas is often not optimal.

If the propositional formulas can be classified according to their suitable CNF conversion algorithm, then the solving efficiency of SMT solver can be improved distinctly. Therefore, an intuitive idea is to extract the features of propositional formulas, and divide propositional formulas into two different categories, the one with higher efficiency of SMT solving using Tseitin algorithm and the one using Technology mapping algorithm. Then using machine learning to train a model which well classify such two different categories. Machine learning [12], [29] is a branch of artificial intelligence that focuses on building models that learns from data and improve their accuracy over time. In machine learning, models are trained to find the law of large amounts of training data, so that the model's predictions on new data can be made as correct as possible. Machine learning algorithms can be divided into supervised learning, unsupervised learning and reinforcement learning according to learning methods. In real world, the application of machine learning is very extensive, such as: data mining and analysis, pattern recognition and many other fields. As for SMT solving, machine learning also has many combined applications [1], [21].

This paper proposes to select a suitable CNF encoding algorithm for each given formulas. Our key idea is to use the existing SMT formulas in the SMT-LIB benchmark repository [2] as training data to train a machine learning model offline, so as to automatically choose a more appropriate CNF encoding algorithm for the formula in the process of SMT solving, hoping to improve the efficiency of SMT solving. We have implemented our approach on KLEE and STP, which are the state-of-the-art symbolic execution engine for C programs and its default underlying constraint solver, respectively. The experimental results, based on extensive evaluation of 86 real-world C programs in Coreutils benchmark, indicate that our method can effectively improve the efficiency of symbolic execution. On average, our method increases the number of the explored paths by 27.2%

The remainder of the paper is organized as follow. Section 2 shows the related work. Section 3 illustrates our method in details. Section 4 gives the evaluation and Section 5 discuss the limitation. Finally we draw a conclusion of the paper.

## II. Related work

Our work is closely related to the constraint solving optimizaitons in symbolic execution and machine learning techniques in constraint solving. We will discuss them in detail.

### *A. Constraint Solving Optimizaitons in Symbolic Execution*

The ability of constraint solving is the main bottleneck for the scalability of symbolic execution. Therefore, lots of research focus on accelerating constraint solving in symbolic execution. A typical idea is optimizing constraint solving in the context of symbolic execution, which mainly focuses on the optimizations of symbolic expression and invokes the underlying solver in a black-box manner [3], [10]. CUTE [22] has implemented a mechanism of fast unsatisfiability check based on the syntactical contradiction of symbolic expression, which reduces invocations of constraint solver by 60-95%. KLEE [3] uses three kinds of optimizations to speed up constraint solving, including caching the counter-examples to avoid calling the underlying solver in certain situations, rewriting the constraint into simpler one, *e.g.* strength reduction and linear simplifications, just like what a compiler does, and splitting the constraint into disjoint sets of independent constraints for better reusing. Aiming at array constraint, KLEE-Array [20] proposes some novel optimizations based on repeated values in constant arrays to simplify the symbolic expressions. In addition, there exists some works which synthesize symbolic execution and constraint solving and then use the constraint solver in a white-box manner. For example, multiplex symbolic execution (MuSE) [31] collects all partial solutions generated by the underlying constraint solver in one time of solving and constructs multiple program inputs according to these solutions.

### *B. Machine Learning Techniques in Constraint Solving*

Recently, machine learning is a hot topic in academia and industry, with new methods invented all the time. Researchers in different research areas benefit from emerging machine learning techniques a lot. In constraint solving, some researchers try to improve the ability of constraint solver by combining machine learning techniques. Portfolio-based approach is an well-known way to improve the efficiency of constraint solver with machine learning methods, such as SATZilla [27], CPHydra [17] and MachSMT [21]. The basic idea is picking a solving algorithm from a set of solving algorithms, which is a typical classifier problem that machine learning method is good at. MLB [15] transforms the feasibility problem of the path condition in symbolic execution into optimization problem and employs an optimization solver which implements a machine learning guided sampling and validation method. FastSMT [1] is designed to generate a faster solving strategy for SMT solving. First, it uses a combining method of random search and neural network to learn a set of candidate solving strategies. Then it synthesizes a combined solving strategy with branches based on the candidates. Besides, Petr Somol et al. proposed a search principle for optimal feature subset selection using the Branch & Bound method [26], which can be used to improve performance of SAT solvers. Earlier research [14] accelerated the SMT solving by learning to select branching rules in DPLL algorithm.

## III. The Proposed Approach

This section presents the details of our intelligent selection method. The framework will be introduced first. Then, the extraction of formula features and the CNF encoding selection are explained in the next two sub-sections.

### A. Framework

Algorithm 1 shows details of our intelligent selection method of CNF encoding. The inputs are a logical formula $formula$ represented in the SMT-LIB format [2]. The algorithm first employs AST to translate input $formula$ to Abstract Syntax Tree representation, $T$ (Line 1). Then, we apply Merge to merge the leaf nodes of $T$ (Line 2) which represent same variables or constants. Merge returns a directed acyclic graph (DAG) $D$. Next, the algorithm carries out Extract (*c.f.* Algorithm 2) on $D$. Extract returns the corresponding feature $F$. Finally, we use an intelligent selection method on $F$ to select the most effective CNF encoding for given logical formula.

**Algorithm 1** ISCE($formula$)

**Input:** The SMT formula $formula$.
**Output:** The CNF encoding method $Result$.
1: $T = \text{AST}(formula)$
2: $D = \text{Merge}(T)$
3: $F = \text{Extract}(D)$
4: $Result = \text{Select}(F)$
5: **return** $Result$

### B. Feature Extraction

Algorithm 2 gives the details of feature extraction from the original formula. The input is a DAG which represents a logic formula compactly, the output is the representation in bag of word model [30].

Specifically, The algorithm considers nodes in DAG as words, and uses the type of nodes to distinguish them, and count the number of nodes in different types. Consider the following example,

$$x_1 \implies (x_2 \wedge x_3) \tag{1}$$

There are three kinds of node types, *i.e.* $variable$, $\implies$ and $\wedge$. The corresponding BoW representation is,

$$\{variable : 3, \implies : 1, \wedge : 1\} \tag{2}$$

which keys are node types and values are the number of nodes in different types.

**Algorithm 2** Extract($D$)

**Input:** The DAG of formula $D$.
**Output:** The BoW representation $BoW$.
$N = \text{Nodes}(D)$
**for** $node \in N$ **do**
    $BoW[node] \leftarrow BoW[node] + 1$
**end for**
**return** $BoW$

### C. Intelligent Selection

The Algorithm 1 uses Select to get the most suitable CNF encoding algorithm, which improves the solving efficiency of SMT solver apparently. The input is feature of a logical formula, which is generated by III-B. The output is Tseitin algorithm or Technology mapping algorithm which improves the solving efficiency most of SMT solver.

We employ an offline trained learning model to predict a CNF encoding algorithm for an logical formula. To train the model, we generate the training data from the existing SMT formula in the SMT-LIB benchmark repository [2]. Each element in the training data is a tuple $(\mathcal{E}(\varphi), t)$ consisting of four parts: $\mathcal{E}(\varphi)$ is the embedding feature of the current formula $\varphi$, $t$ is the specific CNF encoding algorithm which improves the speed of SMT solving more than other (*c.f.* $t = 0$ means Technology mapping is better and $t = 1$ means Tseitin algorithm). Since we are interesting in analyzing computer programs, we choose the formulas in QF_BV and QF_ABV logic, and generate the corresponding embedding feature by III-B. For $t$ of each element in the training data, we use STP [7] as SMT solver under Technology mapping and Tseitin algorithm simultaneously, then set $t$ to the algorithm that spending less time when solve formula $\varphi$.

### D. Symbolic Execution Framework

This sub-section depicts how our intelligent selection method can be integrated into the symbolic execution framework. Algorithm 3 gives the symbolic execution framework. The input is the program under symbolic execution. Our framework adopts a state-based symbolic execution [11] and employs a worklist based implementation. In the beginning, there is only initial state $s_i$ in the worklist (*c.f.* Line 1).

The main loop is a worklist based procedure. When exploring the state space, the symbolic executor selects a state from the worklist to explore the state space (Line 5). During symbolic execution, logical formula of corresponding path condition is generated (Line 6). Then we use our intelligent selection method to decide which CNF encoding algorithm should be used so that SMT solver may be speed up (Line 8). Finally, the CNF encoding algorithm $En_{cnf}$ is applying to speedup the SMT solver and the symbolic executor would append new states into worklist (Line 18).

The intelligent selection needs to balance the effectiveness and selection overhead. In principle, we can have a trained learning model that can recommend the best CNF encoding algorithm for each logical formula in validation set. However,

**Algorithm 3** SE($P$)

**Input:** A program $P$.

1: $worklist = \{s_i\}$
2: $T = 0$
3: $Save_{en} = default$
4: **while** $worklist \neq \emptyset$ **do**
5:   $s = Choose(worklist)$
6:   $C = GenConstraints(P, s)$
7:   **if** $T < K$ **then**
8:     $En_{cnf} = ISCE(C)$
9:     **if** $En_{cnf}\ is\ Save_{en}$ **then**
10:       $T = T + 1$
11:     **else**
12:       $T = 1$
13:       $Save_{en} = En_{cnf}$
14:     **end if**
15:   **else**
16:     $En_{cnf} = Save_{en}$
17:   **end if**
18:   $worklist \leftarrow worklist \cup Execute(s, En_{cnf})$
19: **end while**

the selection introduces more overhead which consist of feature extraction and learning model prediction. This balance is controlled by a variable $K$. A variable $T$ is initialize to 0. We use $T$ to count the times our method continuously predicts the same CNF encoding algorithm. We use $Save_{en}$ to save the previous prediction. When $T$ grows to $K$, we no longer use our selection algorithm but use the $Save_{en}$ to reduce overhead. In our experiments, we set $K$ to 100.

## IV. Experiments

We have implemented our method on KLEE [3] (*i.e.* a state-of-the-art engine for C programs). KLEE's version is 2.3-pre. We use STP as the backend solver and bit-vector SMT theory for encoding the path constraints. STP's version is 2.3.3. We train the intelligent selection model by XGBoost [4]. We implement the AST translation and Bag of Word embedding based on jSMTLIB [5].

We have conducted extensive experiments to answer the following two research questions:

- **RQ1**: what is the performance impact of the XGBoost intelligent selection algorithm?
- **RQ2**: how effective is our intelligent selection algorithm? Here, effectiveness means exploring more paths during symbolic execution.

### A. Experimental Setup

To evaluate the effectiveness of our method, we use Coreutils as the benchmark. Coreutils is the mainstream benchmark for the symbolic execution researches whose implementations are based on KLEE. The used Coreutils's version is 6.11. There are 89 programs (46746 SLOCs) in total.

We train the XGBoost model for intelligent selection as follows. We use the QF_BV, QF_ABV SMT-LIB2 benchmarks [2] for generating the data set. We filter the formulas whose ASTs contain more than 50,000 nodes. We use the bag of words (BoW) model [30] and the one-hot encoding [9] as the embedding feature of the logical formulas and the CNF encoding algorithm, respectively. We use STP [7] under Tseitin and Technology mapping algorithm to find the most suitable CNF encoding for every formula in our benchmarks. The timeout threshold is set to 30 seconds. If timeout occurred both Tseitin and Technology mapping algorithm, we would filter the corresponding formula.

We compare our method (which implements based on XGBoost) with the one employing Multi-layer Perceptron classifier from sklearn [19], to show what is the performance impact of the XGBoost algorithm. We have 18,782 formulas after filtered in above way. We select 50% for training dataset and the others for validation sets. XGBoost uses default settings. For MLP in sklearn, we use $adam$ as solver, the hidden layer sizes is $(30, 60, 30, 10)$ and the activation function is $logistic$.

We compare our symbolic execution framework with intelligent selection integrating, with baseline KLEE under two search heuristics, *i.e.*, DFS and BFS. We analyze each Coreutils program in 30 minutes. We set the end condition of intelligent selection (*c.f.* Algorithm 3 Line 8) as intelligent selection generating same continuous results more then $K$ times. $K$ is a threshold that we set it to 100 in our experiments. We used the same options as KLEE mentions in [3]. But we close three optimizations, *i.e.*, constraint independence, counterexample cache and branch cache, to generate more queries to smt solver.

All the experiments were carried out on a Server with 64GB memory and 16 3.1 GHz cores. The operating system is Ubuntu 14.04.

### B. Experimental Results

**Answer to RQ1.** To answer the first question, we evaluate our XGBoost based intelligent selection by comparing with MLP (Multi-layer Perceptron classifier) classifier based version in three aspects: accuracy, recall and confusion matrix [24].

TABLE I
Accuracy & Recall.

| Model | accuracy | recall |
|---|---|---|
| XGBoost | 91% | 89% |
| MLP | 91% | 83% |

Table I shows the accuracy and recall of different machine learning model. XGBoost has the same accuracy as MLP but higher recall. Our dataset consists of 3,025 formulas that is suitable for Technology mapping algorithm and 15,757 formulas for Tseitin algorithm. As our data is imbalance, where there are different number of samples in each class, the recall is more important than accuracy.

Table II and III are confusion matrix of XGBoost and MLP, respectively. The column names and row names, $Map$ or

$Tseitin$, means the number of formulas that solved efficiently when encoding to CNF by Technology mapping or Tseitin algorithm. In Table II, of 3,000 formulas classified to $Map$ (*c.f.*, first line), XGBoost judged that 2,618 were $Map$. But In Table III, MLP judged 2,095 were $Map$ of the same 3,000 formulas. XGBoost predicts 523 samples correctly more than MLP, which is 17% in $Map$ class.

TABLE II
XGBOOST CONFUSION MATRIX.

| | | Predicted: Map | Predicted: Tseitin |
|---|---|---|---|
| Actual | Map | 2618 | 407 |
| | Tseitin | 1357 | 14400 |

TABLE III
MLP CONFUSION MATRIX.

| | | Predicted: Map | Predicted: Tseitin |
|---|---|---|---|
| Actual | Map | 2095 | 930 |
| | Tseitin | 670 | 15087 |

> *Answer to RQ1:* XGBoost have better performance than MLP on $recall$ in the imbalance dataset. More specifically, XGBoost correctly predicts 17% of the samples on the minority class.

**Answer to RQ2.** To answer the second research question, we compare our symbolic execution framework with intelligent selection integrating, with baseline KLEE. We evaluate them in path number which have been explored during symbolic execution.

Figure 1&2 show the comparison results of new paths in BFS and BFS, respectively. The X-axis shows the benchmark programs ordered by the values in Y-axis. The Y-axis shows the relative increasing of the explored paths, which is defined as follows, where $N_{\textbf{OPT}}$ denote the number of paths explored after employing our method, and $N_{\textbf{BASELINE}}$ represents the number of original symbolic execution.

$$\frac{N_{\textbf{OPT}} - N_{\textbf{BASELINE}}}{N_{\textbf{BASELINE}}} \tag{3}$$

As shown by Figure 1, our method can improves the number of explored paths on 62(73%) programs. On the other hand, there are 24(27%) programs on which we decrease the number of paths because of the feature extraction overhead; however, the decreasing is slight, *i.e.*, -2.81% (-5.2%∼-0.14%) on average. Our method can on average improves the number of explored paths by 27.2% (-5.2%∼469%).

Figure 2 depicts the corresponding results in DFS. we improve the number of explored paths on 60(69%) programs. Our method decreases the number of the explored paths on 26(31%) programs since the feature extraction overhead. The decreasing is still slight as BFS, -6.1%(-36%∼-0.5%) on average. Our method improves the number of explored paths by 26.7% (-36%∼522%).

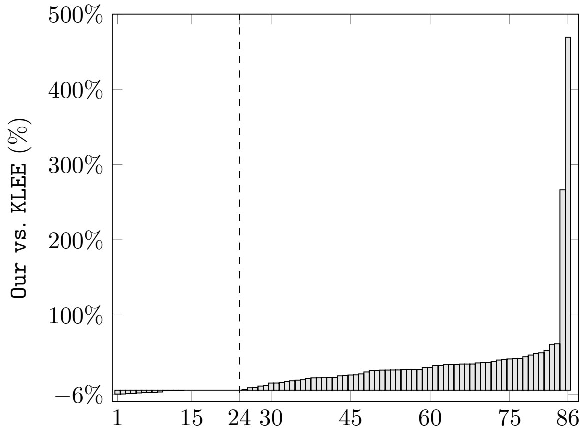

Fig. 1. Relative increasing of Path number in BFS.

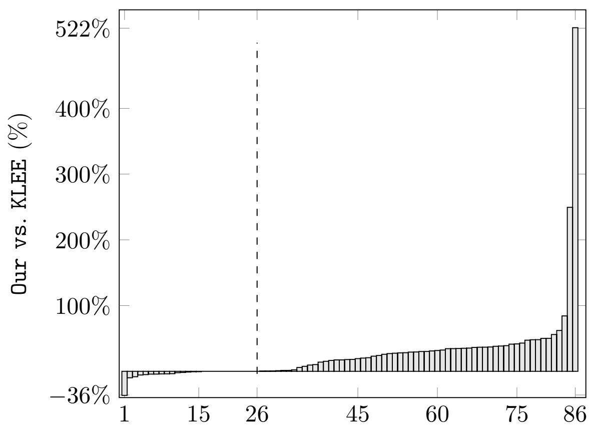

Fig. 2. Relative increasing of Path number in DFS.

> *Answer to RQ2:* Our method is effective to improve symbolic execution's ability of path exploration. On average, our method increases the number of paths by 27.2%.

## V. THREAT TO VALIDITY

The external validity is a major threat to our experimental results. It is mainly due to the limited benchmark we used and the generalization of machine learning model. For the former, although the number and type of benchmark may be insufficient, Coreutils is a widely used benchmark for evaluating the performance of symbolic execution [3], [16], [28], and the current experimental results have demonstrated the effectiveness of our method. However, we plan to evaluate our prototypes on more benchmarks in the next step.

## VI. CONCLUSION

In this paper, we propose a method to intelligently select a suitable CNF encoding algorithm for a given logical formula, which is more efficient for constraint solving than the one using a specific CNF encoding algorithm for all formulas. Our approach leverages offline trained machine learning models to predict the suitable CNF encoding algorithm for a given logical formula. We integrate our selection algorithm into the symbolic execution framework based on KLEE and STP, which are the state-of-the-art symbolic execution engine for C programs and its default underlying constraint solver, respectively. The experimental results, based on extensive evaluation of 86 real-world C programs in Coreutils benchmark, indicate that our method can effectively improve the efficiency of symbolic execution. On average, our method increases the number of the explored paths by 27.2%.

## ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China (No.61632015)

## REFERENCES

[1] M. Balunovic, P. Bielik, and M. T. Vechev, "Learning to solve SMT formulas," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 10 338–10 349.

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

[3] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[4] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, Eds. ACM, 2016, pp. 785–794.

[5] D. R. Cok, "jsmtlib: Tutorial, validation and adapter tools for smt-libv2," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 480–486.

[6] N. Eén, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up SAT," in *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, ser. Lecture Notes in Computer Science, J. Marques-Silva and K. A. Sakallah, Eds., vol. 4501. Springer, 2007, pp. 272–286.

[7] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.

[9] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning*, ser. Adaptive computation and machine learning. MIT Press, 2016.

[10] 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, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 177–187.

[11] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[12] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, "Machine learning: a review of classification and combining techniques," *Artif. Intell. Rev.*, vol. 26, no. 3, pp. 159–190, 2006.

[13] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[14] M. G. Lagoudakis and M. L. Littman, "Learning to select branching rules in the DPLL procedure for satisfiability," *Electron. Notes Discret. Math.*, vol. 9, pp. 344–359, 2001.

[15] X. Li, Y. Liang, H. Qian, Y. 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, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 554–559.

[16] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, ser. Lecture Notes in Computer Science, E. Yahav, Ed., vol. 6887. Springer, 2011, pp. 95–111.

[17] E. OMahony, E. Hebrard, A. Holland, C. Nugent, and B. OSullivan, "Using case-based reasoning in an algorithm portfolio for constraint solving," in *Irish conference on artificial intelligence and cognitive science*, 2008, pp. 210–216.

[18] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 179–180.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[20] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 68–78.

[21] Preiner, Mathias, Niemetz, Aina, Scott, Joseph, and Ganesh, Vijay, "Machsmt: A machine learning-based algorithm selector for smt solvers," 2020.

[22] 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, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272.

[23] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153.

[24] K. M. Ting, *Confusion Matrix*, 2017.

[25] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, 1983.

[26] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proc. IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015.

[27] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, 2008.

[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 745–761.

[29] G. P. Zhang, "Neural networks for classification: a survey," *IEEE Trans. Syst. Man Cybern. Part C*, vol. 30, no. 4, pp. 451–462, 2000.

[30] Y. Zhang, R. Jin, and Z. Zhou, "Understanding bag-of-words model: a statistical framework," *Int. J. Mach. Learn. Cybern.*, vol. 1, no. 1-4, pp. 43–52, 2010.

[31] Y. Zhang, Z. Chen, Z. Shuai, T. Zhang, K. Li, and J. Wang, "Multiplex symbolic execution: Exploring multiple paths by solving once," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 846–857.