Automatic Bug Discovery in SAT Solvers via Fuzzing

Ahmad Ziada, Reza Behishti, Ali Sbeih
Department of Computer Science
Amherst College
Email: {aziada26, rbehishti27, asbeih25}@amherst.edu

***Abstract*—This project aims to test the robustness and correctness of SAT solvers through the application of automated testing techniques, namely fuzz-testing. Fuzz testing techniques revolve around the generation of randomized, syntactically valid CNF formulas with varying complexity, difficulty, and structure. The project involved carrying out six experiments (three fuzzing techniques, two rounds per technique) to test eight solvers from 2007, 2011, and 2024 SAT competitions and validating the results outputted by the solvers. Overall, the team did not find any logical issues in any of the experiments (incorrect satisfiability status or satisfying assignment). However, the increase in the complexity of the test CNF statements resulted in a significant number of timeouts (solver exceeding the time limit set by the team). Compared with results obtained in the literature from previous years using the same techniques, the project demonstrates that significant improvements have been made in the accuracy and robustness of SAT solvers.**

I. Introduction

Ensuring the correctness and robustness of SAT solvers is essential, given their critical role in formal verification, AI planning, and constraint solving. A lot of focus in modern SAT solvers is on speed and performance, so they remain vulnerable to subtle bugs that can lead to incorrect results or crashes. These kinds of errors are produced by edge-case inputs that are rarely covered by conventional test suites.

For that reason, this project explores the use of fuzzing, an automated software testing technique that generates valid and unexpected, random input to uncover such bugs in SAT solvers. By generating a diverse set of random, syntactically valid CNF formulas, fuzzing can effectively stress-test SAT solvers to reveal weaknesses. The focus of our work is to test and compare the effectiveness of three fuzzers, established in previous work in the literature, in exposing bugs across both legacy and state-of-the-art SAT solvers.

The team evaluates solvers from different competitions (2007-2024), classifies the types of bugs found, and develops validation tools to verify the satisfiability results. Through this approach, we seek to deepen understanding of solver robustness and evaluate the effectiveness of fuzzing as a practical bug discovery tool.

II. Motivation and Background

SAT solvers have widespread usage in critical systems. But here's the issue: if a solver gives the wrong answer, everything built on top of it can go wrong too. And as Brummayer et. al showed in "Automated Testing and Debugging of SAT and QBF Solvers", even top-performing SAT solvers can have serious bugs, and correctness can't be taken for granted. However, these issues are hard to detect, and traditional testing practices often fail to reveal them.

Fuzzing is a powerful testing tool. Due to its automated nature, fuzzing allows for high-throughput testing and reveals bugs in reasoning procedures such as DPLL, CDCL, unit propagation, and decision heuristics. These core reasoning procedures form the backbone of modern SAT solvers. This project aims to uncover faults within these algorithmic components through automated test generation. The team performed background research on fuzzing techniques and prior work in testing SAT solvers and decided to base the project on the fuzzing techniques developed by Brummayer et. al in "Automated Testing and Debugging of SAT and QBF Solvers" [1]. In particular, the fuzzing techniques we examine are 3SAT-Gen, CNFuzz, and FuzzSAT. They are designed to produce progressively more structured CNFs, ranging from randomly generated 3-CNF formulas to CNFs derived from Boolean circuits via Tseitin transformation.

Building on this previous work, we apply these fuzzing techniques to new solvers and investigate the effects of the input complexity by adjusting the parameters of the fuzzers. We develop scripts for automatic validation of satisfying assignments and for verifying unsatisfiability through majority voting.

III. Methodology

A. Fuzzing Techniques

The team used the three fuzzing techniques developed in [1] to generate the CNF statements and performed two rounds of experiments per technique with modified parameters (1000 CNF statements in the first round and 300 in the second). The aim of modifying some of the parameters from the original techniques was to change the difficulty level of the generated instances. Below is a description of the techniques:

a) 3SATGen: picks a number of variables m randomly from a certain range, then chooses a hardness ratio $r \in [3, 5]$ and generates c ternary CNF clauses, with $c = m \cdot r$. The range for m was chosen to be [10, 100], then [10, 400] in the second round.

b) CNFuzz: introduces locality restrictions by generating instances that contain more internal structure than 3SATGen. Based on the authors in [1], the introduction of internal

structure will test the more interesting features of an SAT solver. CNFuzz's additional features include building the CNF statement as layers, with the number of layers chosen randomly across a certain range and a width $w \in [10, 70]$, which represents the number of new variables introduced per layer. [1] includes additional features of this technique, which the team used as is in this project. In the first round, the team used a range of layers [1,20], and in the second, the number was fixed at 10.

c) FuzzSAT: introduces more structure and complexity by using Boolean circuits to generate CNF instances. To clarify, a directed acyclic graph (DAG) illustration of a random Boolean circuit is produced. Afterwards, the graph is translated into a CNF formula using Tseitin transformation. Similar to the number of variables in 3SATGen and the number of layers in CNFuzz, the number of nodes on the FuzzSAT DAG is also randomly chosen from a range, which was picked by the team to be [10, 100] in the first round and [10, 400] in the second. The DAG generation process involves picking operands from the nodes (with a probability of negation of 1/2) and then choosing a random operator from the set of operators O = AND, OR, XOR, IFF. The process involves more details which were not modified in this project and are explained in detail in [1].

B. Time Limit

The team first attempted to run the experiments without a time limit on the solvers since introducing a time limit might generate false positives (unsolved just because the solver needed more time, not because of an issue in robustness). However, even with the supposedly simplest technique, the solvers took a significantly long time to finish running (more details can be found in the Results section), and thus the team decided to introduce a time limit of 5 seconds per statement for the first round and 20 seconds per statement for the second.

C. SAT Solvers

The team picked a total of eight SAT solvers to test:

- **From the 2024 SAT Competition:**
  - hCaD
  - Kissat_MAB_ESA
  - Kissat_MAB_DC
  - Kissat_sc2024
  - CaDiCaL
- **From the 2011 SAT Competition:**
  - Cryptominsat
  - Lingeling
- **From the 2007 SAT Competition:**
  - picosat

D. Running Experiments

The team developed Python scripts [2] to automate the CNF statements generation process, feeding instances into solvers, and saving results to CSV files for further analysis (results include satisfiability status outputted (or unsolved if timeout) and satisfying assignment in case of SAT).

IV. Evaluation Methodology

A. Validating Satisfiability Status

Since the team used the same CNF instances across solvers in all six experiments carried out, we decided to combine all the outputs in one CSV file and write a Python script `validate_status.py` [2] to discover if there is any conflict in terms of satisfiability status per statement (ignoring unsolved instances).

B. Validating Satisfying Assignment

The next step in the experiments involved validating the satisfying assignments obtained in the case of SAT. To achieve that, the team developed a Python script `validate_assignment.py` that scans the CSV files obtained and filters the SAT cases [2]. Afterwards, another script takes the filtered CSV and matches the CNF statement to its corresponding assignment that was outputted by a solver to feed them into the Z3 solver. The team here assumes that it is very unlikely for Z3 solver, with its reputation and popularity in the field, to make the same mistake as our subject solvers in case there was a bug.

V. Results

A. Time Limit

The first experiments with 3SATGen were run with 1000 CNF statements without a time limit on the solvers. However, the solvers took a significantly long time to finish running, with Cryptominisat being the fastest with a total runtime of 34 minutes, and six of the other seven solvers taking more than five hours without completing the run. Thus, the team decided to set a time limit of 5 seconds/ CNF statement for the easier versions of the fuzzing techniques (each generating 1000 CNF statements) and 20 seconds/ CNF statement for the more challenging counterparts (each generating 300 CNF statements).

B. 3SATGen

The first experiment was 1000 CNF statements generated by 3SATGen, with the number of variables generated randomly between 10 and 1000. All the solvers were able to solve all 1000 statements, with a unanimous output of 689 satisfiable and 311 unsatisfiable (Table I). The fact that there was no disagreement between the solvers in the counts of SAT and UNSAT was a first indication of the absence of bugs. The team verified the expectation first by feeding the satisfiable statements with their corresponding outputted assignments into Z3, and all the assignments were verified to be valid. Afterwards, using the majority vote technique, the team cross-verified all the unsatisfiable statements across the eight solvers, and no mismatches were found (Table VII). Therefore, 3SATGen did not result in the discovery of any bugs with the current parameters.

After 3SATGen range of variables was changed to [10,400], 300 CNF statements were generated and fed into the eight solvers with 20 20-second time limit. None of the solvers

was able to solve all the statements: the number of unsolved statements ranged from 13 to 23, but without a clear trend between the older and the modern solvers (Table II). Z3 confirmed all the outputted satisfying assignments to be valid, and similarly, there was consensus among the solvers regarding the unsatisfiable statements (considering the solvers that were able to solve the statement in time) (Table VII).

TABLE I: Results for 1000 CNF inputs generated by 3SATGen with number of variables ∈ [10, 100]

Solver	UNSAT	SAT	UNSOLVED
CaDiCaL	311	689	0
hCaD	311	689	0
kissat_MAB_DC	311	689	0
kissat_MAB_ESA	311	689	0
kissat_sc2024	311	689	0
lingeling	311	689	0
cryptominisat	311	689	0
picosat	311	689	0

TABLE II: Results for 300 CNF inputs generated by 3SATGen with number of variables ∈ [10, 400]

Solver	UNSAT	SAT	UNSOLVED
CaDiCaL	76	206	18
hCaD	76	204	20
kissat_MAB_DC	71	206	23
kissat_MAB_ESA	73	206	21
kissat_sc2024	79	206	15
cryptominisat	75	202	23
lingeling	74	203	23
picosat	84	203	13

C. CNFuzz

The experiment with CNFuzz involved a first round of 1000 CNF statements with the number of layers randomly picked between 1 and 20, and 300 statements with a number of layers fixed at 10. The results for both rounds are included in Tables III and IV, respectively. There were no unsolved statements, nor invalid satisfying assignments (verified by Z3 solver), nor disagreements regarding the unsatisfiable statements (Table VII). Therefore, no bugs were discovered in both rounds of CNFuzz.

TABLE III: Results for 1000 CNF inputs generated by CNFuzz with number of layers ∈ [1, 20]

Solver	UNSAT	SAT	UNSOLVED
CaDiCaL	780	220	0
hCaD	780	220	0
kissat_MAB_DC	780	220	0
kissat_MAB_ESA	780	220	0
kissat_sc2024	780	220	0
cryptominisat	780	220	0
lingeling	780	220	0
picosat	780	220	0

TABLE IV: Results for 300 CNF inputs generated by CNFuzz after fixing the number of layers at 10

Solver	UNSAT	SAT	UNSOLVED
CaDiCaL	158	142	0
hCaD	158	142	0
kissat_MAB_DC	158	142	0
kissat_MAB_ESA	158	142	0
kissat_sc2024	158	142	0
cryptominisat	158	142	0
lingeling	158	142	0
picosat	158	142	0

D. FuzzSAT

The first FuzzSAT round with 1000 CNF statements, with the number of variables ∈ [10, 100]. The number of unsolved cases within the time limit was minimal (Table V), and again, no invalid satisfying assignments nor conflicts regarding the satisfiability status (Table VII).

The second round involved 300 statements with a new variable range [10, 400]. No invalid assignments or incorrect satisfiability status incidents were obtained (Table VI). However, nearly 20% of the statements were not solved in time (Tables VI and VII). To maintain consistency across experiments, the team kept the time limit at 20 seconds, but the proportion of unsolved cases would have been lower had the time limit been increased to 30 or 40 seconds. Although the oldest solver, picosat had the highest number of unsolved cases at 66, the difference between this number and the numbers for other solvers is not big enough to draw any conclusions. The number of unsolved instances ranged between 54 and 66, which reflects the need for more time per statement by all the solvers.

TABLE V: Results for 1000 CNF inputs generated by FuzzSAT with number of variables ∈ [10,100]

Solver	UNSAT	SAT	UNSOLVED
CaDiCaL	204	793	3
hCaD	205	794	1
kissat_sc2024	205	794	1
kissat_MAB_DC	205	794	1
kissat_MAB_ESA	205	794	1
cryptominisat	204	794	2
lingeling	204	794	2
picosat	204	793	3

TABLE VI: Results for 300 CNF inputs generated by FuzzSAT with number of variables ∈ [10,400]

Solver	UNSAT	SAT	UNSOLVED
hCaD	85	161	54
CaDiCaL	85	159	56
kissat_sc2024	83	160	57
kissat_MAB_DC	82	159	59
kissat_MAB_ESA	82	158	60
cryptominisat	82	159	59
lingeling	82	156	62
picosat	80	154	66

TABLE VII: Overall experimental results

	Invalid Assignment	Conflicting UNSAT	Total Unsolved
3SATGen 1st round	0	0	0
3SATGen 2nd round	0	0	156 (6.5%)
CNFuzz 1st round	0	0	0
CNFuzz 2nd round	0	0	0
FuzzSAT 1st round	0	0	14 (0.014%)
FuzzSAT 2nd round	0	0	473 (19.7%)

E. Base Case: Empty CNF File

The team tested the eight solvers' output in case of an empty CNF file. Seven of the eight solvers yielded an error signaling an empty input. Nevertheless, one solver, Cryptominisat, gave SAT as an output, which was interesting to observe.

F. Results Summary

In all, none of the six experiments performed using CNF statements generated by the three Fuzzing techniques 3SAT-Gen, CNFuzz, and FuzzSAT resulted in the discovery of a logical bug in any of the eight solvers tested, namely an incorrect satisfiability status or an invalid satisfying assignment. Furthermore, none of the solvers terminated early or gave a segmentation fault, both of which were errors yielded by 2007 and 2009 solvers in [1]. This reflects an improvement in the reliability of SAT solvers.

VI. Conclusion

In conclusion, no logical bugs (incorrect satisfiability status or satisfying assignments) were discovered in any of the six experiments carried out. However, the number of unsolved statements under the time limit increased considerably when the parameters of the most complex fuzzing technique, FuzzSAT, were modified. This does not necessarily mean the solvers were not able to solve these statements, but rather a sign that there is potential for improvement in terms of speed.

The absence of logical bugs and unexpected crashes in our results represents an optimistic sign regarding the advancement in the robustness of modern SAT solvers, especially when comparing our results to those in [1]. However, our project does not by any means imply the logical perfection of the eight solvers tested: Due to the popularity of the fuzzing techniques used in the experiments, the developers of SAT solvers almost certainly have tested their solvers on fuzzer-generated statements before releasing and fixed the potential bugs before releasing their solvers. This means that for bugs to be discovered in contemporary SAT solvers, fuzzing techniques need to be enhanced in terms of their complexity or combined with other bug discovery techniques, such as machine learning's adversarial attacking algorithms.

References

[1] R. Brummayer, F. Lonsing, and A. Biere, "Automated Testing and Debugging of SAT and QBF Solvers," in *SAT 2010*, Springer, pp. 44–57.

[2] A. Sbeih, R. Behishti, and A. Ziada, "AR Fuzzing Project," GitHub Repository: https://github.com/AliS25/AR_Fuzzing_Project.git