Uncovering and Classifying Bugs in MaxSAT Solvers through Fuzzing and Delta Debugging

Tobias Paxian, Armin Biere

University of Freiburg, Georges Koehler Allee 51, 79110 Freiburg, Germany

Abstract

In this study we continue the success story of fuzz testing automated reasoning tools by providing the first extensive fuzzing study on MaxSAT solvers. As somewhat expected we identify interesting defects and failures in almost all MaxSAT solvers from the MaxSAT Evaluation 2022. A classification of these bugs into four main classes and various subclasses can help developers in debugging them. Finally, we show how to uncover additional issues by a new MaxSAT specific delta debugging strategy on top of reducing the failing test cases significantly. This study clearly shows that MaxSAT solvers are less reliable and robust than expected, and further suggests that fuzzing and delta debugging can help to improve this situation. Furthermore, we provide a regression suite of interesting small instances.

Keywords

Fuzzing, Fuzz Testing, Delta Debugging, Testing and Debugging

1. Introduction

Reliable maximum satisfiability (MaxSAT) solving is of great interest due to wide-ranging applications such as hardware and software verification, constraint programming, and AI planning [1, 2, 3, 4, 5, 6, 7]. It is crucial to develop efficient and robust MaxSAT solvers to address ever-growing complexity and reliability in these domains. The continuous improvement of MaxSAT algorithms which can be seen at the yearly MaxSAT evaluation (MSE) [8], allows for increasingly complex problems to be solved.

MaxSAT and its variations are optimization variants of SAT solving, seeking a truth assignment to a Boolean formula in Conjunctive Normal Form (CNF) such that the number of satisfied clauses is maximized [9, 10]. In the weighted variant, a weight is assigned to each clause, where the goal is, to maximize the accumulated weight of the satisfied clauses.

There are different ways of achieving a reliable MaxSAT solver. One method is programming the whole MaxSAT solver in a verified programming language, as it is already done in SAT with IsaSAT [11]. This has the drawback that all applied techniques have to be proven, which is not easily achieved, and therefore the solver is generally slower on complex problems. Another way is adding proofs to the solutions [12, 13] verifiable by a proof checker. Unfortunately, proofs are

POS'23: 14th Pragmatics of SAT international workshop, July 04, 2023, Alghero, Italy

*Corresponding author.

†These authors contributed equally.

paxiant@cs.uni-freiburg.de (T. Paxian); biere@cs.uni-freiburg.de (A. Biere)

https://cca.informatik.uni-freiburg.de/ (T. Paxian); https://cca.informatik.uni-freiburg.de/ (A. Biere)

Table 1
This overview shows the used techniques in the fuzzed solvers and their rank (sometimes two solver variants) in the MSE 22' (Pacose MSE 21'). We combined the MaxSAT preprocessor maxpre2 with EvalMaxSAT, the most reliable solver according to our results. Most of the solvers using multiple solving techniques as: Branch and Bound (BB); Pseudo Boolean Constraints (PB); Hitting Set (HS); Unsat-based (UB); Sat-Unsat-based (SUB); Satisfiability Modulo Theories (SMT) and recently Integer Linear Programming (ILP) is becoming popular to solve instances (Top 4 solver of 2022 using ILP).

Rank	MaxSAT Solver	HS	UB	SUB	PB	BB	ILP	SMT	Others
1. & 2.	CASHWMaxSAT [19, 20]		X				X		
3. & 7.	UWrMaxSat [21]		X				X		
4.	MaxHS [22]	X	X	X			X		
5. & 6.	WMaxCDCL [23, 24]					X			
8.	EvalMaxSAT [25]		X						
9.	CGSS [26]		X						
10.	Exact [27]		X		X				
6.	Pacose [28]			X	X				
	z3rc2 [29, 30]		X					X	X
	z3maxres [29, 30]		X					X	X
	z3wmax [29, 30]		X					X	X
	maxpre 2.0 [31, 32] 2.0								X

not yet available for the weighted variant. In our study we chose a third technique, a dynamic software testing approach, requiring no changes in the code of solvers. This approach, called fuzz testing or fuzzing, is applied to enhance the robustness of solvers.

Fuzz testing has been successful in detecting software vulnerabilities and bugs across various fields [14]. The first paper in 1990 shows the efficiency of identifying reliability issues in UNIX utilities [15]. In MaxSAT related fields such as Satisfiability Modulo Theories (SMT) [16], SAT, Quantified Boolean Formulas (QBF) [17], and And-Inverter Graph Verification [18] fuzz testing has demonstrated its effectiveness. This study presents the first extensive study in fuzzing MaxSAT solvers. As expected, we found numerous failures in almost all the 15 fuzzed MaxSAT algorithms and solvers as detailed in Section 3. We then classified these bugs into 4 main and 14 subcategories, as outlined in Section 2.4. Additionally, we employ our novel delta debugger to shrink the formulas, simplifying the instances, as described in Section 2.3. During the delta debugging phase many additional faults were triggered, due to the reduction process as described in Section 3.

In Table 1 we describe the tested solvers, their ranking in the MSE and also point out the techniques they use.

Regarding related work, we are only aware of two available MaxSAT fuzzers supporting the old pre MSE22 WCNF format. The first fuzzer [33] tests only for invalid exit codes of the solver and a missing o-value or one which is bigger than sum of weights. The second fuzzer [34] comes along with a MaxSAT solver GaussMaxHS [35], which has not yet participated in the MSE. The authors generated a CNF, added xor gates and converted it with bit blasting into WCNF with a bundle of python and shell scripts. Our understanding is that both fuzzers do not check whether the o-value matches the model or an optimum is reached.

2. Methodology

In this section we introduce the four key components of our study: We begin by discussing the techniques to construct random WCNF formulas by our fuzzer `WCNFuzz`. Next, we describe our `WCNFCompare` tool, developed to compare and log the faults of solvers, providing a valuable direct comparison of their results. Following that, we discuss our implementation `WCNFddmin` of the delta debugging algorithm and explain its unique features. Finally, we present our fault classification scheme, which categorizes the discovered faults from `WCNFCompare`.

2.1. Fuzz testing

Fuzzing is a technique to detect software vulnerabilities with the idea to treat software as a black box and generate random inputs in order to uncover critical defects as segmentation faults, overflows or incorrect results [14]. Our novel tool `WCNFuzz` is a generation-based grammar-aware [36] fuzzer. It generates random WCNF instances, following the input language rules, to identify crashes, performance bottlenecks, invalid solver outputs and hard to solve instances. This allows developers to understand weaknesses and strengths of their solvers to improve the reliability and efficiency of their software [14].

There already exist successful fuzzing tools for CNF formulas like CNFuzz and FuzzSAT [17]. CNFuzz generates structured instances, which results in problems closer to industrial examples, than simply applying a variable clause ratio [37, 38, 39], as done in many studies to generate hard random 3-SAT formulas. Our goal is to construct difficult problems with only a few clauses, because it is shown in previous studies [17, 14], that with such instances most faults are triggered. `WCNFuzz` modifies CNFuzz to generate such WCNF formulas.

In the following we use our implementation of a "Linear Congruential Generator", with values from the "Art of Computer Programming" [40], to pick all random choices. `WCNFuzz` adds up to 10 layers of clauses with each containing up to 70 variables. Layers consist entirely either of hard or soft clauses. Soft clause layers are chosen randomly with higher chance initially and lower chance for the following soft layers. The clauses of the n'th layer contains variables of its own layer with high probability and with decreasing chances variables of lower layers. Most of the clauses (around 2/3) are ternary, with decreasing chances they are of higher order and around every tenth clause is binary. We calculate the number of clauses in each layer by picking a suitable clause-variable-ratio. As we want hard clauses to be satisfiable with a high chance, we pick a low ratio $r \in [1, 2.5]$ for hard clause layers. For soft clause layers we want to have at least some clauses making the problem unsatisfiable, therefore we choose a high ratio $r \in [3.5, 5.5]$.

Additionally, we add Tseitin encoded Equality, AND, 3-XOR and 4-XOR gates. We include an activation literal to all clauses of $3/4$ of the gate encodings and add one additional soft clause containing only the negated activation literal. Furthermore, one out of ten instances is forced to contain only soft clauses. In very rare cases, all layers and gates are decided to consist only of hard clauses.

The maximal weight in the MSE is often relatively small, and often there are unweighted problems to solve. Therefore, the maximal weight is chosen to be in one of the following ranges, for each interval the probability is 1/5: $[1, 1]$; $[2, 32]$; $[33, 256]$; $[257, 65535]$; with a probability of $4/25$ it is in the range of $[65536, 2^{32}]$; and with the probability $1/25$ it is in the range of $[2^{32}+1, 2^{63}-1]$; with $2^{63}-1$ being the maximal possible weight. We further ensure that the maximal sum of weights is less than $2^{64}-1$, as described in the official rules of the MSE [8].

2.2. Comparing and Logging Results

In the following, we discuss challenges in fuzzing a single MaxSAT solver and present our solution `WCNFCompare`, a Python tool to automate the comparison, validation, and logging of multiple MaxSAT solver results.

Evaluating the optimality of a single fuzzed MaxSAT solver presents a challenge in the absence of a certified proof or solver. To address these issues, we introduce `WCNFCompare`, a Python tool that automates the process of comparing the results produced by multiple MaxSAT solvers. In its default configuration, `WCNFCompare` runs all solvers mentioned in Section 1, with a default timeout of 20 seconds for each solver. It then verifies the satisfiability of hard clauses, and checks the o-value against the model for each solver result, using our `WCNFVerifier`. We use the best model verified solution as a representative for the unverified optimal outcome. If other solvers do not produce the same o-value, it indicates an erroneous result. Subsequently, the tool classifies results into approximately 30 different fault classes, which are doubled again, depending on the sum of weights, as discussed in Section 2.4. Each solver is assigned a number, as are the types of faults that occur (see Section 2.4), with fault types numbered from 1 to 60 within the tool. If multiple errors occur, then the exit codes and solver position are added up and taken modulo 255, as 255 is the highest possible exit code.

One limitation of the comparison script is that identical exit codes can emerge from various solver failure combinations. Consequently, as the delta debugger reduces instances only considering the exit code, we can end up with different solver-fault combinations at the end of a reduction. To address this issue, we introduced a command line option that restricts the script to only return the exit code of a single solver for the delta debugger run.

`WCNFCompare` also logs results, generating individual files for each WCNF-solver-fault combination. As fuzzing and delta debugging can run concurrently on multiple cores, we need to prevent access to the same solver-fault combinations logfile from multiple cores, which is done by adding a unique seed. These files contain a clear fault comparison overview, the final o-values of each solver, error messages, and the solver's output to stdout and stderr. At the end of the whole fuzzing/delta debugging run, these log files are consolidated into a single log file per solver fault combination. This approach offers a significant advantage: it permits the use of any instance generation tool or shrinking tool, while maintaining a consistent logging process.

2.3. Delta Debugging

Delta debugging [41, 42, 43] is a powerful and efficient technique to isolate and simplify failure causing inputs in software testing. If we apply delta debugging without restarts, we have a complexity of $\mathcal{O}(n)$. It assists to identify the root cause of a problem by systematically decreasing the size of a test input while preserving the failure triggering property.

This greedy approach attempts to remove portions of the input not contributing to the fault. Initially, the approach attempts to remove the whole input, then successively reduces this to half, a quarter, and so on. In the worst-case scenario, every second atomic element needs to be removed, which makes the algorithm worse than merely iterating once through all elements.

Introducing `WCNFddmin`, a novel delta debugger with innovative features that extend beyond capabilities of existing SAT, QBF, and SMT delta debuggers. Our tool includes the following reduction phases, which are processed sequentially, with the delta debugging algorithm performed on each:

1. Removing Clauses: Tries to remove as many clauses as possible, not differentiating between soft and hard clauses.
2. Removing Variables: Creates a list of all variables in the problem instance and iterates over them in the delta debugging approach to remove as many variables as possible. The WCNF printing function takes care of removing the corresponding literals out of the clauses. Empty clauses are treated as if the entire clause was removed.
3. Removing Literals: Treats all literals in all clauses as a long list of literals and applies the algorithm on this list. As for the variables, the printing function removes finally the literals.
4. *NEW* Converting soft to hard clauses: `WCNFddmin` tries to transform as many soft into hard clauses as possible. Soft clauses are generally more difficult to handle by a MaxSAT solver and the optimization problem becomes easier for a human reader to understand with more hard clauses.
5. *NEW* Weight reduction to 1: As big weights are normally harder to handle by the solver or human debugger, we try to reduce as many weights as possible to 1.
6. *NEW* Binary weight reduction phase: The debugger systematically lowers the weight of soft clauses with weights greater than 1 through a kind of binary search. If multiple weights are selected the weight is halved, as long as the compare script returns the same exit code. Only if a single weight is processed a real binary search can be performed. In the default case the weight is reduced until the upper bound minus the lower bound are 10% or less of the original weight. In practice this phase is the most expensive one.

Techniques 1 to 3 are already known by CNF delta debugger as CNFddmin [17] and the additional techniques are new reduction ideas. To the best of our knowledge, we are the first to apply techniques repeatedly in a delta debugger. We do this until either no further improvements are possible or until the reduction progress after the i'th round for $i > 3$ falls below $i - 3\%$. We consider each technique separately in this process. As far as we know, our tool is the first to present the following features, each can be reversed if not successful:

1. *NEW* Shuffling clauses: Randomly changes the order of clauses.
2. *NEW* Shuffling literals in each clause: Randomly rearranges the literals in each clause.
3. *NEW* Renaming variables: Ensures that there are no variable numbering gaps.

`WCNFddmin`'s unique features and $\mathcal{O}(n)$ complexity make it an advanced tool for isolating and simplifying failure-inducing inputs in MaxSAT problems, supporting developers in discovering root causes of solver bugs. Additionally, during the reduction process, `WCNFCompare` is called for all created WCNF instances. We are the first to log errors and saving fault triggering WCNFs arising during the reduction phase. Instances that might not have been generated by the initial fuzzer are produced. This aims to uncover additional interesting new solver-fault combinations that might have remained undetected otherwise.

2.4. Fault Classification

Next, we discuss how faults, in the context of MaxSAT solvers, can be classified. Therefore, we introduce four main fault classes: Crashes, Lower/Upper Bound Violations, Performance Regressions, and Other Issues.

`WCNFCompare` originally returns 60 different fault classes for which 1-30 are for a sum of weight smaller than 2^{32} and the 31-60 debug the same faults for bigger weights. We simplified this list into four main and 14 subclasses, still differentiating between small and big weights.

In order to simplify the fault classification list, we propose the following notation for the different types of o-values: let o_{solver} denote the best o-value given in the solver output, o_{model} denotes the o-value represented by the solver's model (as calculated by the model-verifier), and finally o_{min} denotes the best verified o-value of all solvers. Using this notation, we present the different fault classes and their respective errors:

1. Crashes:
   - 1.1. MaxSAT solver's exit code is 134 (SIGABRT, internal error or inconsistency)
   - 1.2. MaxSAT solver's exit code is 135 (SIGSEGV, segmentation fault)
   - 1.3. MaxSAT solver's exit code is 136 (SIGFPE, arithmetic error or overflow)
   - 1.4. MaxSAT solver's exit code is 137 (SIGKILL, immediately shutdown)
   - 1.5. MaxSAT solver's exit code is 139 (SIGSEGV / SIGBUS, segmentation or bus fault)
   - 1.6. MaxSAT solver's exit code is XXX (all other exit codes)
2. Bound Violations:
   - 2.1. $o_{\text{min}} < o_{\text{solver}}$ and $o_{\text{solver}} == o_{\text{model}}$
   - 2.2. $o_{\text{solver}} \neq o_{\text{model}}$ and $o_{\text{model}} \neq o_{\text{min}}$ and $o_{\text{solver}} \neq o_{\text{min}}$.
   - 2.3. Either o_{model} or o_{solver} unequals o_{min}.
   - 2.4. Verifier asserts that provided model is UNSATISFIABLE.
   - 2.5. Verifier states hard clauses are SATISFIABLE, but solver states UNSATISFIABLE.
3. Performance Regressions:
   - 3.1. Potential Fault: Solver had timeout, but this timeout is 50 times larger than the average time of the non-timeout solvers.
4. Other Issues:
   - 4.1. Solver has an error either stated in stdout or stderr.
   - 4.2. Inconsistency in status line and output.
   - 4.3. Unexpected behavior of a verifier.

Determining the severity of these errors is a crucial aspect. As an example, fault 3.1. is only a potential fault that may indicate a performance issue or a more severe infinite loop problem. Generally it is not considered a serious fault. Crashes are more severe, but at least they do not deliver an incorrect value/model which we tend to trust. Bound violations, on the other hand are considered serious, as we cannot trust the solver reliability. Most of these faults can be detected with a sanity check, which implies a check if the hard clauses are satisfiable and if the model's o-value matches the given solver o-value. This suggests that the most critical fault in these violations could occur if this quick check appears to be sane, yet a fault such as the one indicated by 2.1. is still present. In the current version of our tool, we overlooked the inclusion of a model sanity check. This means we only verify if the provided model already contradicts the formula, without checking if the number of variables is correct. We have acknowledged this oversight and plan to address it in the upcoming version of the tool.

The sequence in which these faults are evaluated during the fault classification process plays an important role in ensuring an accurate fault detection. The order should minimize the risk of missing important bugs as occurred in previous versions of the compare script. For instance, the solver status should be evaluated before evaluating the o-value and model. Is it worth considering the occurrence of multiple faults in a single solver run? If, for example an error message is thrown, but has at the same time a bound violation, we decided to only catch the bound violation, as we do not interpret error messages. Further some solver as MaxHS print often such messages but still provides correct results. We believe that our approach has a good balance between not over-categorizing faults and not neglecting important faults.

3. Results

In this section, we present results of our MaxSAT solver fuzzing and delta debugging experiments. The tests were run on a system powered by an i9-12900 processor with 16 cores and 128 GB of memory. The experiments were executed on all 16 cores around one week for fuzzing and afterwards we performed delta debugging on the first five faults that occurred in each class of the original 60 classes, which took another week. All experimental data, the regression suite, and source code is available at https://cca.informatik.uni-freiburg.de/maxsatfuzz. During setup, we challenged the following issues:

- Z3 doesn't support competition standard output, therefore we implemented a transformation script.
- The MSE provides a useful benchmark code base for verifying models, transforming WCNFs from new to old format and vice versa, and more. However, we could not use these tools as they only accept a sum of weights up to 2^{63}, and we aimed to support the full range up to $2^{64} - 2$, as it is standard in the competition.
- Z3, Pacose, and maxpre2 require the old evaluation format as input, we added a script to rewrite the instances. However, this led to additional fault classes during parallel execution, which were non-reproducible and hence, excluded from our results.
- Maxpre2 outputs the old v-line format regardless the command line options. We implemented a script to rewrite the output, which likely triggered unverifiable fault classes.

Table 2

The fault occurrences in each fault class outlined in Subsection 2.4. Each cell contains 4 values, represented as $\substack{a|b \\ c|d}$, with the first row (a|b) representing results from fuzzing, and the second row (c|d) representing fault occurrences triggered by delta debugging. The first value of each cell-row $\binom{a}{c}$ corresponds to instances with a sum of weights less than 2^{32}, while the second value $\binom{b}{d}$ corresponds to a higher sum of weights, but less than $2^{64}-1$. Several faulty instances triggered faults in multiple solvers. The MSE 22' solvers are arranged according to their rank in the weighted category. The table shows $\substack{44|70 \\ 60|74}$ fault-solver occurrences in the four categories. It is evident that not all solver can reliably handle higher weights, as indicated especially often by fault 2.5. (false classification of an instance as unsatisfiable). Delta debugging triggered a wider range of faults, possibly due to the presence of less structured instances with variable gaps, resulting from reduction and shuffling. Note that fault 4.3. is a special case, in which the sum of weights equals $2^{64}-1$ and the instance structure did not permit a simple reduction. In these cases our verifier failed (only possible with exactly this sum). Still, some of these instances from 4.3. caused all solver to produce incorrect results or crashes.

<table>
<tr><th></th><th>Crashes</th><th>Bound Violations</th><th></th><th></th><th></th><th></th><th>Perf.</th><th>Other Issues</th><th></th><th></th></tr>
<tr><th></th><th>1.</th><th>2.1.</th><th>2.2.</th><th>2.3.</th><th>2.4.</th><th>2.5.</th><th>3.</th><th>4.1.</th><th>4.2.</th><th>#faults</th></tr>
<tr><td>CASHWMaxSAT-CorePlus [19]</td><td>|8
822|2e3</td><td>1e4|3e3
6e3|1e4</td><td>1e4|5e3
9e3|7e4</td><td>2e4|8e4
2e4|3e5</td><td>1|
260|</td><td>|
|</td><td>|13
|1e4</td><td>|20
82|1e4</td><td>|
|</td><td>4e4|9e4
3e4|4e5</td></tr>
<tr><td>CASHWMaxSAT-Plus [20]</td><td>|8
822|2e3</td><td>1e4|3e3
6e3|1e4</td><td>1e4|5e3
9e3|7e4</td><td>2e4|8e4
2e4|3e5</td><td>1|
260|</td><td>|
|</td><td>|13
|1e4</td><td>|20
82|1e4</td><td>|
|</td><td>4e4|9e4
3e4|4e5</td></tr>
<tr><td>UWrMaxSat-SCIP [21]</td><td>|7
822|1e3</td><td>3e4|8e3
7e3|1e4</td><td>5e4|2e4
1e4|1e5</td><td>5e4|9e4
2e4|3e5</td><td>266|56
505|265</td><td>|
|</td><td>|16
|5e3</td><td>|21
|6e3</td><td>|
|</td><td>1e5|1e5
4e4|4e5</td></tr>
<tr><td>MaxHS [22]</td><td>1|1
249|395</td><td>6e4|2e4
2e4|3e4</td><td>3e4|8e3
1e4|3e4</td><td>5e4|3e4
2e4|1e5</td><td>|
|</td><td>|
|</td><td>2e4|5e3
1e4|3e4</td><td>1e4|4e3
2e3|9e3</td><td>8e3|3e3
9e3|5e3</td><td>2e5|7e4
7e4|2e5</td></tr>
<tr><td>WMaxCDCL [23]</td><td>|2
48|2e3</td><td>|78
3|3e3</td><td>|138
|7e3</td><td>9|3e3
2e3|3e4</td><td>2e5|5e4
3e4|7e4</td><td>|2e4
|4e4</td><td>|61
|1e4</td><td>2|3e4
932|2e5</td><td>|
|</td><td>2e5|1e5
3e4|3e5</td></tr>
<tr><td>WMaxCDCL-BandAll [24]</td><td>|25
48|3e3</td><td>|78
3|3e3</td><td>|138
|7e3</td><td>9|3e3
2e3|3e4</td><td>2e5|5e4
3e4|7e4</td><td>|2e4
|4e4</td><td>|61
|1e4</td><td>2|3e4
932|2e5</td><td>|
|</td><td>2e5|1e5
3e4|3e5</td></tr>
<tr><td>UWrMaxSat [21]</td><td>|
822|</td><td>2|14
792|609</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
55|4</td><td>|
|</td><td>|
|</td><td>2|14
2e3|613</td></tr>
<tr><td>EvalMaxSAT [25]</td><td>|
822|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>0|0
822|</td></tr>
<tr><td>CGSS [26]</td><td>3e5|
2e4|</td><td>39|17
9e3|5e3</td><td>|4
|2e3</td><td>|3e4
|1e5</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>3e5|3e4
2e4|1e5</td></tr>
<tr><td>Exact [27]</td><td>|
2|</td><td>1|
1e3|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>4|12
2e3|4e3</td><td>|
|</td><td>|
|</td><td>5|12
3e3|4e3</td></tr>
<tr><td>Pacose [28]</td><td>3e5|9e4
2e4|3e5</td><td>|
|</td><td>11|
889|9e3</td><td>|
|1</td><td>|2
|3</td><td>|
|</td><td>|8
|3</td><td>12|
1e3|268</td><td>23|10
2e3|3e3</td><td>3e5|9e4
3e4|3e5</td></tr>
<tr><td>z3maxres [29, 30]</td><td>|
|</td><td>78|7e3
5e3|3e4</td><td>12|3e4
1e3|8e4</td><td>5|6e3
2e3|4e4</td><td>|
|</td><td>|1e5
|4e5</td><td>|3
359|2e3</td><td>|
|</td><td>|
|</td><td>95|1e5
8e3|5e5</td></tr>
<tr><td>z3wmax [29, 30]</td><td>|
|</td><td>|8
|374</td><td>|185
|2e3</td><td>|206
281|9e3</td><td>|32
481|1e3</td><td>|1e5
|4e5</td><td>|
|</td><td>|
|</td><td>2e6|4e5
3e5|7e5</td><td>2e6|5e5
3e5|1e6</td></tr>
<tr><td>z3rc2 [29, 30]</td><td>|
|</td><td>7e5|1e5
1e5|2e5</td><td>5e4|4e4
1e4|1e5</td><td>2|4e3
1e3|3e4</td><td>|
|</td><td>|1e5
|4e5</td><td>|2
|646</td><td>|
|</td><td>|
|</td><td>8e5|3e5
1e5|8e5</td></tr>
<tr><td>maxpre2 [31, 32] +EvalMaxSAT</td><td>2e5|8e3
4e4|5e4</td><td>|
|</td><td>|
|</td><td>9e4|2e4
4e4|7e4</td><td>2e6|4e5
2e5|9e5</td><td>|
|</td><td>|
|</td><td>|
|</td><td>|
|</td><td>2e6|5e5
2e5|1e6</td></tr>
<tr><td>#faults</td><td>7e5|1e5
8e4|3e5</td><td>9e5|2e5
2e5|3e5</td><td>2e5|1e5
6e4|5e5</td><td>2e5|4e5
1e5|1e6</td><td>3e6|5e5
2e5|1e6</td><td>|3e5
|1e6</td><td>2e4|6e3
2e4|9e4</td><td>1e4|6e4
5e3|4e5</td><td>2e6|4e5
3e5|7e5</td><td>7e6|2e6
1e6|6e6</td></tr>
<tr><td>#faulty solver</td><td>4|8
12|8</td><td>9|11
11|11</td><td>7|10
7|11</td><td>9|11
10|12</td><td>6|6
7|6</td><td>|5
|5</td><td>2|10
4|11</td><td>4|6
6|7</td><td>3|3
3|3</td><td>44|70
60|74</td></tr>
</table>

```
cat red.wcnf
2 -1 0
2 -2 -3 0
1 1 4 0
3 -3 2 0
1 -5 3 -6 0
1 -6 3 -2 0
h 1 6 0
h 3 5 0
h 4 0
```

```
CASHWMaxSAT-CoreP*
... SCIP 7.0.3 ...
c SCIP optimum = 2
v 100110
o 2
s OPTIMUM FOUND
```

```
CASHWMaxSAT-Plus
... SCIP 7.0.3 ...
c SCIP optimum = 2
v 100110
o 2
s OPTIMUM FOUND
```

```
UWrMaxSat-SCIP
... SCIP 8.0.0 ...
c SCIP optimum: 2
v 100110
o 2
s OPTIMUM FOUND
```

```
MaxHS
... c #vars: 5
c #Clauses: 8 ...
o 2
s OPTIMUM FOUND
v 11011
```

```
z3rc2
c Convert WCNF
c Convert Output
s OPTIMUM FOUND
o 2
v 010111
```

```
EvalMaxSAT
s OPTIMUM FOUND
o 1
v 000111
c Total time: 347 µs
```

Figure 1: This shows a comparison of six solver outputs running the same WCNF instance (blue), all but the solver of the bottom right (green) are faulty (orange). The first five solver claim that the result is 2 which matches their given model, while other solvers found a better result as shown by EvalMaxSAT. We discovered this fault by our fuzzing (reproducible with seed 1633784527538860147) and reduced the instance by our delta debugging technique. Remarkably, all top four solvers from the MaxSAT Evaluation 22' and Microsofts Z3 solver with the RC2 technique failed. The first three solvers, which do not satisfy the first clause with their model, employ the SCIP solver in two versions as a preprocessor. By deactivating it, we get correct results. Interestingly, different incorrect results were observed among these SCIP versions in other examples (seed 2065838794411768763). MaxHS identifies only 5 variables and 8 clauses; however, it is unlikely that this is due to a parser error, as each variable appears in at least 2 clauses. The incomplete model still produces a wrong result of 2 (variable 6 is irrelevant). Microsoft's Z3 solver found another incorrect o-value and model (not satisfying clauses 5 and 6). In contrast, other solvers such as EvalMaxSAT (shown in green) found the optimal result of 1 (not satisfying clause 5).

These were also excluded from our results.

- Pacose sometimes writes "SATISFIABLE" instead of "s SATISFIABLE."
- Solvers throw different exit codes. For instance, when an optimum solution is found, Exact returns 30, while Pacose returns an exit code of 10.
- CGSS and Exact only work with *.wcnf named files.
- In UWrMaxSat-SCIP, grepping for "UNSAT" in the verbose=0 variant let the status line disappear.

Table 2 presents our findings, displaying the number of faults detected for each fault class, as outlined in Section 2.4, in fuzzing and delta debugging runs. The exact numbers are not crucial, as the detected faults increase at a consistent rate, if run for extended periods. A detailed examination of the solvers during this study led to several interesting observations:

- Only MaxHS and Z3 can handle empty instances. This is the reason why EvalMaxSAT crashed 822 times with exit code 255 and the UWrMaxSAT and CASHWMaxSAT variants with exit code 139. This happens only during the reduction phase, as no empty instance is generated with our fuzzer.
- CGSS and Pacose do not accept a file with only hard clauses, resulting in 271131 crashes.
- The following invalid exit codes occured causing a crash fault (fault category 1.): 1, 3, 105, 108, 134, 135, 136, 139, 141, 255
- CGSS throws exit code 1 for unsatisfiable instances - but the same exit code is thrown, if

the whole instance is empty. This means, that these instances are reduced to the empty instance, after the first solver call of WCNFddmin.

Our proposed reduction techniques showed significant effectiveness. In some cases, the initial reduction was minimal, but subsequent shuffling or clause renaming phases enabled significant reductions. During the binary weight reduction phase, weights were typically reduced until the difference between the upper and lower bounds was less than 10% of the original weight. Disabling this percentage rule led to excessively long runs for large weights. Even with the 10% rule in place, this phase, along with the literal reduction phase, was the most time-consuming. Some instances took up to a week to reduce, particularly when performance regression (3.1) occurred for instances with large numbers, making weight reduction a very time-intensive task.

The reduced instances uncovered intriguing problems, such as in Figure 1, where a fault was significantly minimized. Another issue occured with the timeout fault class 3.1 in MaxHS due to a simple instance with just three soft and one hard clause, the original instance (reproducible with seed 193251431004265909) having 41 soft and 157 hard clauses. This problem forced MaxHS into a type of infinite loop, only terminated by the technique's 1500-second timeout. A nearly identical error with another instance occurred with the Exact solver, but this time with a real timeout. That instance (seed 1795142913688699408) could be reduced to 7 soft and 24 hard clauses, underscoring that even timeouts can reveal interesting bugs.

Furthermore, we highlight that we could trigger 54 additional solver-fault combinations during our delta debugging phase. As demonstrated in Table 2, these 54 combinations are additional entries within the second line. The ability to invoke these additional combinations is significant, as it provides further opportunities to probe the robustness of the solvers and expose potential vulnerabilities. This observation underscores the value of our novel approach of logging during the delta debugging phase, as it notably enhances the comprehensiveness of our testing process.

The results of our MaxSAT solver fuzzing and delta debugging experiments reveal crucial insights into the behavior and robustness of various solvers. Our fuzzer WCNFuzz effectively detects a significant number of interesting faults due to WCNFcompare, in various fault classes. Despite some initial challenges, our WCNFddmin tool leads to considerable reductions in input instances and exposes interesting issues, such as unexpected exit codes, timeout faults and interesting bound violations. These findings highlight the importance of rigorous testing and debugging in the development and refinement of MaxSAT solvers.

4. Discussion

In the course of our research, we have constructed a useful regression set of interesting instances that we believe will be beneficial for solver development. These instances include:

- Empty instance, empty soft/hard clause.
- Non trivial reducible maximal weight instances with a maximum single weight $2^{63} - 1$ and a maximal sum of weights $2^{64} - 2$.
- Simple unsatisfiable instances.
- Tautology soft/hard clauses

- With our fuzzer created and delta debugged instances for each fault class-solver combination, causing each at least one solver to crash.

At least one of these instances are triggering a fault in all the tested solvers, as some of these instances are not yet supported by the official rules. E.g. empty clauses / instances cannot be handled even by most SAT solvers. We suggest the following rules be incorporated into the competition solver rules:

- An empty instance should yield a weight "o 0", with an empty model line "v" and the status line "s OPTIMUM FOUND".
- An unsatisfiable instance should produce the status line "s UNSATISFIABLE".
- An empty hard clause should result in an unsatisfiable instance.
- An empty soft clause should be unsatisfiable, but the instance can still be satisfiable.
- The exit code of a solver should be 0 for all results but "s UNKNOWN".

We would like to offer the MaxSAT community these instances, provided as a zip file from the MSE homepage, along with a script executing the solver with a subset or all instances and verifying the results and models. This surely would assist developers in improving the robustness of their solvers.

5. Conclusion

In this research, we explored an automated testing approach for MaxSAT solvers, utilizing fuzzing and delta debugging techniques to uncover and minimize intriguing faults.

Our proposed techniques are notably effective. While initial reduction was not always significant, subsequent shuffling or clause renaming phases enabled substantial reductions. The input instances were significantly reduced during the delta debugging phase, and our methods allowed us to identify and isolate critical issues, even within large, complex instances.

We also created a compact regression suite of small instances for solver development, which were shown to trigger specific errors in all tested solvers. We will provide these instances along with a script for executing and verifying the solver's results to the MaxSAT community. We also proposed new rules to include in the MaxSAT Evaluation rulebook, to ensure the standard handling of basic clauses as provided by our regression suite.

In conclusion, our research demonstrates that automated testing methods, such as fuzzing and delta debugging, can trigger severe faults in MaxSAT solvers. We believe that our work will significantly contribute to the ongoing efforts to enhance the robustness and reliability of these solvers.

References

[1] O. J. Berg, A. J. Hyttinen, M. J. Järvisalo, Applications of MaxSAT in data analysis, Proceedings of Pragmatics of SAT 2015 and 2018 (2019).

[2] J. Berg, M. Järvisalo, Cost-optimal constrained correlation clustering via weighted partial maximum satisfiability, Artificial Intelligence 244 (2017) 110–142.

[3] B. Ghosh, K. S. Meel, IMLI: An incremental framework for MaxSAT-based learning of interpretable classification rules, in: Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society, 2019, pp. 203–210.
[4] L. Zhang, F. Bacchus, MaxSAT heuristics for cost optimal planning, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 26, 2012, pp. 1846–1852.
[5] Y. Chen, S. Safarpour, J. Marques-Silva, A. Veneris, Automated design debugging with maximum satisfiability, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 29 (2010) 1804–1817.
[6] P. Raiola, T. Paxian, B. Becker, Minimal witnesses for security weaknesses in reconfigurable scan networks, in: 2020 IEEE European Test Symposium (ETS), IEEE, 2020, pp. 1–6.
[7] T. Seufert, F. Winterer, C. Scholl, K. Scheibler, T. Paxian, B. Becker, Everything you always wanted to know about generalization of proof obligations in PDR, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2022).
[8] F. Bacchus, J. Berg, M. Järvisalo, R. Martins, A. e. Niskanen, MaxSAT Evaluation 2022: Solver and Benchmark Descriptions, Technical Report, Department of Computer Science, University of Helsinki, Helsinki, 2022. URL: http://hdl.handle.net/10138/347396.
[9] C. M. Li, F. Manya, MaxSAT, hard and soft constraints, in: Handbook of satisfiability, IOS Press, 2021, pp. 903–927.
[10] F. Bacchus, M. Järvisalo, R. Martins, Maximum satisfiabiliy, in: Handbook of Satisfiability, IOS Press, 2021, pp. 929–991.
[11] M. Fleury, J. C. Blanchette, P. Lammich, A verified SAT solver with watched literals using imperative HOL, in: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2018, pp. 158–171.
[12] D. Vandesande, W. De Wulf, B. Bogaerts, QMaxSATpb: a certified MaxSAT solver, in: Logic Programming and Nonmonotonic Reasoning: 16th International Conference, LPNMR 2022, Genova, Italy, September 5–9, 2022, Proceedings, Springer, 2022, pp. 429–442.
[13] S. Gocht, R. Martins, J. Nordström, A. Oertel, Certified CNF Translations for Pseudo-Boolean Solving, in: K. S. Meel, O. Strichman (Eds.), 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022), volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, pp. 16:1–16:25. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16690. doi:10.4230/LIPIcs.SAT.2022.16.
[14] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler, The fuzzing book, 2019.
[15] B. P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of UNIX utilities, Communications of the ACM 33 (1990) 32–44.
[16] R. Brummayer, A. Biere, Fuzzing and delta-debugging SMT solvers, in: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, 2009, pp. 1–5.
[17] R. Brummayer, F. Lonsing, A. Biere, Automated testing and debugging of SAT and QBF solvers, in: Theory and Applications of Satisfiability Testing–SAT 2010: 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 13, Springer, 2010, pp. 44–57.
[18] D. Kaufmann, A. Biere, Fuzzing and delta debugging and-inverter graph verification tools, in: Tests and Proofs: 16th International Conference, TAP 2022, Held as Part of STAF 2022, Nantes, France, July 5, 2022, Proceedings, Springer, 2022, pp. 69–88.

[19] Z. Lei, Y. Wang, S. Pan, S. Cai, M. Yin, CASHWMaxSAT-CorePlus: Solver Description, volume B-2022-2 of [8], 2022, p. 8. URL: http://hdl.handle.net/10138/347396.
[20] Y. Wang, S. Pan, Z. Lei, S. Cai, M. Yin, S. Hu, Y. Zhou, CASHWMaxSAT-Plus: Solver Description, volume B-2022-2 of [8], 2022, p. 9. URL: http://hdl.handle.net/10138/347396.
[21] M. Piotrów, UWrMaxSat Entering the MaxSAT Evaluation 2022, volume B-2022-2 of [8], 2022, pp. 21–22. URL: http://hdl.handle.net/10138/347396.
[22] F. Bacchus, MaxHS in the 2022 MaxSAT Evaluation, volume B-2022-2 of [8], 2022, pp. 17–18. URL: http://hdl.handle.net/10138/347396.
[23] J. Coll, S. Li, C.-M. Li, F. Manyà, D. Habet, K. He, MaxCDCL and WMaxCDCL in MaxSAT Evaluation 2022, volume B-2022-2 of [8], 2022, pp. 15–16. URL: http://hdl.handle.net/10138/347396.
[24] J. Coll, S. Li, C.-M. Li, F. Manyà, D. Habet, J. Zheng, K. He, MaxCDCL-BandAll in MaxSAT Evaluation 2022, volume B-2022-2 of [8], 2022, p. 23. URL: http://hdl.handle.net/10138/347396.
[25] F. Avellaneda, C.-E. Bilodeau-Savaria, L. Normand, Weighted version of EvalMaxSAT 2022, volume B-2022-2 of [8], 2022, p. 12. URL: http://hdl.handle.net/10138/347396.
[26] H. Ihalainen, J. Berg, M. Järvisalo, CGSS in the 2022 MaxSAT Evaluation, volume B-2022-2 of [8], 2022, pp. 10–11. URL: http://hdl.handle.net/10138/347396.
[27] J. Devriendt, Exact: evaluating a pseudo-Boolean solver on MaxSAT problems, volume B-2022-2 of [8], 2022, pp. 13–14. URL: http://hdl.handle.net/10138/347396.
[28] T. Paxian, S. Reimer, B. Becker, Dynamic polynomial watchdog encoding for solving weighted MaxSAT, in: Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21, Springer, 2018, pp. 37–53.
[29] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14, Springer, 2008, pp. 337–340.
[30] N. Bjørner, A.-D. Phan, L. Fleckenstein, νz-an optimizing SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21, Springer, 2015, pp. 194–199.
[31] H. Ihalainen, J. Berg, M. Järvisalo, Clause redundancy and preprocessing in maximum satisfiability, in: Automated Reasoning: 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8–10, 2022, Proceedings, Springer, 2022, pp. 75–94.
[32] T. Korhonen, J. Berg, P. Saikko, M. Järvisalo, Maxpre: an extended MaxSAT preprocessor, in: Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20, Springer, 2017, pp. 449–456.
[33] N. Manthey, MaxSAT fuzzer, 2020. URL: https://github.com/conp-solutions/maxsat-fuzzer.
[34] M. Soos, K. S. Meel, GaussMaxHS github repository, 2021. URL: https://github.com/meelgroup/gaussmaxhs, available at https://github.com/meelgroup/gaussmaxhs.
[35] M. Soos, K. S. Meel, Gaussian elimination meets maximum satisfiability, in: Proceedings of

the International Conference on Principles of Knowledge Representation and Reasoning, volume 18, 2021, pp. 581–587.
[36] J. Wang, B. Chen, L. Wei, Y. Liu, Superion: Grammar-aware greybox fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 724–735.
[37] D. Mitchell, B. Selman, H. Levesque, et al., Hard and easy distributions of SAT problems, in: Aaai, volume 92, 1992, pp. 459–465.
[38] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, Y. Shoham, Understanding random SAT: Beyond the clauses-to-variables ratio, in: Principles and Practice of Constraint Programming–CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10, Springer, 2004, pp. 438–452.
[39] J. A. Navarro, A. Voronkov, Generation of hard non-clausal random satisfiability problems, in: Proceedings of the National Conference on Artificial Intelligence, volume 20, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005, p. 436.
[40] D. E. Knuth, The art of computer programming. volume 2: Seminumerical algorithms, Bull. Amer. Math. Soc (1997).
[41] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Transactions on Software Engineering 28 (2002) 183–200.
[42] G. Misherghi, Z. Su, HDD: hierarchical delta debugging, in: Proceedings of the 28th international conference on Software engineering, 2006, pp. 142–151.
[43] A. Zeller, Why programs fail: a guide to systematic debugging, Elsevier, 2009.

A. Interesting Faults and Additional Tables

```
$ cat faulty/bug-2999777783999949289.wcnf
c seed 2999777783999949289
...
c variables      91
c hard clauses   312
c soft clauses   143
c sum of weights 2330668
...

$ ./cashwmaxsatcoreplus -no-bin -no-sat -m -no-par -bm bug-2999777783999949289.wcnf
c Using SCIP solver, version 7.0.3, https://www.scipopt.org
c scip_time = 600.000000
c Starting SCIP solver  (with time-limit = 600s) ...
c SCIP optimum = 47778
v 0001010100100000100110000000101100011100000001101101110001101110101001101001111001110110100000
o 47778
s OPTIMUM FOUND

$ ./cashwmaxsatplus -no-bin -no-sat -no-par -bm -m bug-2999777783999949289.wcnf
c Using SCIP solver, version 7.0.3, https://www.scipopt.org
c Starting SCIP solver  (with time-limit = 600s) ...
c SCIP optimum = 47778
v 0001010100100000100110000000101100011100000001101101110001101110101001101001111001110110100000
o 47778
s OPTIMUM FOUND

$ ./uwrmaxsat -v0 -no-bin -no-sat -no-par -scip-cpu=500 -m -bm bug-2999777783999949289.wcnf
c Using COMiniSatPS SAT solver by Chanseok Oh (2016)
c Using SCIP solver, version 8.0.0, https://www.scipopt.org
c Starting SCIP solver  (with time-limit = 500s) ...
c SCIP optimum: 47778
v 0001010110100000101110000000101101011110100011011010100011011101010011010010110001101001000
o 47778
s OPTIMUM FOUND

$ ./maxhs -no-printOptions -printSoln bug-2999777783999949289.wcnf
...
c WARNING blit in model not set to false when soft is satisfied
c Solved by solve_wt_lsu (fine).
o 35620
s OPTIMUM FOUND
v 1011011100101000100000000011100101101100100000100111101011110101110101000110110010101010000 1
...

$ ./z3rc2.sh bug-2999777783999949289.wcnf
c Converting to old format:
c Run Z3 Solver and convert Output to standard output format:
s OPTIMUM FOUND
o 29806
v 0010001101101100110010010011001011011101100101000111110101000111100010001000000111010100010

Correct solution:
$ ./EvalMaxSAT_bin bug-2999777783999949289.wcnf
s OPTIMUM FOUND
o 25997
v 0010001100101100010000000010001010010001000101000111110001000101100010001000000111010000010
c Total time: 2.98 ms
```

```
$ cat red.wcnf
1 1 2 0
2 -3 0
1 -4 3 1 0
2 3 -1 0
3 5 0
h 3 4 0
h 2 0
```

```
$ ./Cash-CP red.wcnf
... SCIP 7.0.3 ...
c SCIP optimum = 2
v 11111
o 2
s OPTIMUM FOUND
```

```
$ ./Cash-P red.wcnf
... SCIP 7.0.3 ...
c SCIP optimum = 2
v 11111
o 2
s OPTIMUM FOUND
```

```
$ ./UWMS-S red.wcnf
... SCIP 8.0.0 ...
c SCIP optimum: 2
v 11111
o 2
s OPTIMUM FOUND
```

```
$./MaxHS red.wcnf
... solve_unwt_lsu.
o 2
s OPTIMUM FOUND
v 11011
```

```
$ ./EvalMS red.wcnf
s OPTIMUM FOUND
o 1
v 01011
c Total time: 468 µs
```

Figure 2: Example: red-1633784527538860147.wcnf

Table 3
Fault classification for FUZZED faults with a sum of weights $0 <= \text{sow} < 2^{32}$

	1.	2.1.	2.2.	2.3.	2.4.	2.5.	3.	4.1.	4.2.	4.3.
CASHWMaxSAT-CorePlus		12272	12581	16103	1					
CASHWMaxSAT-Plus		12273	12581	16102	1					
UWrMaxSat-SCIP		26948	47838	51976	266					
MaxHS	1	64118	25031	53404			16987	13862	8008	
WMaxCDCL				9	246599			2		
WMaxCDCL-bandall				9	246599			2		
UWrMaxSat		2								
EvalMaxSAT										
CGSS	272131	39								
textscExact		1					4			
z3maxres		78	12	5						
z3wmax									2214409	
z3rc2		744136	53451	2						
z3pd-maxres		68	23	6						
z3maxres-bin		77	10	5						
Pacose	272512		11					12	23	
maxpre2EvalMaxSAT	165433			87668	2071078					
# faults	710077	860012	151538	225289	2564544		16991	13878	2222440	
# faulty solver	4	11	9	11	6		2	4	3	

Table 4

Fault classification for FUZZED faults with a sum of weights $2^{32} <= \text{sow} < 2^{64} - 1$

	1.	2.1.	2.2.	2.3.	2.4.	2.5.	3.	4.1.	4.2.	4.3.
CASHWMaxSAT-CorePlus	8	3476	5213	82742			13	20		
CASHWMaxSAT-Plus	8	3476	5213	82743			13	20		
UWrMaxSat-SCIP	7	7571	19026	88541	57		16	21		
MaxHS	1	15765	8474	34689			5358	3513	2555	
WMaxCDCL	2	78	141	3347	50183	24803	61	26451		
WMaxCDCL-bandall	25	78	141	3347	50183	24803	61	26428		
UWrMaxSat		15								
EvalMaxSAT										
CGSS		17	4	26523						
textscExact							12			
z3maxres		7082	25777	6144		95406	3			
z3wmax		8	185	207	32	95406			403532	
z3rc2		138926	36401	4093		95408	2			
z3pd-maxres		7083	25781	6144		95407	3			
z3maxres-bin		7074	25777	6144		95407	4			
Pacose	92198	1	7	5	112		48		10	
maxpre2EvalMaxSAT	8131			20953	443308					
# faults	100380	190650	152140	365622	543875	526640	5594	56453	406097	
# faulty solver	8	14	13	14	6	7	12	6	3	

Table 5

Fault classification for Delta Debugged faults with a sum of weights $0 <= \text{sow} < 2^{32}$

	1.	2.1.	2.2.	2.3.	2.4.	2.5.	3.	4.1.	4.2.	4.3.
CASHWMaxSAT-CorePlus	822	6486	9225	16811	260			82		
CASHWMaxSAT-Plus	822	6486	9224	16812	260			82		
UWrMaxSat-SCIP	822	7095	11828	18079	505					
MaxHS	249	17014	14059	18186			13134	2160	8887	
WMaxCDCL	48	3		2092	25051			932		
WMaxCDCL-bandall	48	3		2092	25051			932		
UWrMaxSat	822	792					55			
EvalMaxSAT	822									
CGSS	16294	8545								
textscExact	2	1046					1866			
z3maxres		4592	1312	1543			359			
z3wmax				281	481				324996	
z3rc2		134985	10870	1394						
z3pd-maxres	330	3949	1956	1439			359			
z3maxres-bin		4865	1490	1542			364			
Pacose	23933		889					1171	1935	
maxpre2EvalMaxSAT	37417			41664	160573					
# faults	82431	195861	60853	121935	212181		16137	5359	335818	
# faulty solver	13	13	9	12	7		6	6	3	

Table 6
Fault classification for Delta Debugged faults with a sum of weights $2^{32} <= \text{sow} < 2^{64} - 1$

	1.	2.1.	2.2.	2.3.	2.4.	2.5.	3.	4.1.	4.2.	4.3.
CASHWMaxSAT-CorePlus	2634	12214	70969	312752			13003	9677		
CASHWMaxSAT-Plus	2633	12219	70968	312752			12994	9677		1
UWrMaxSat-SCIP	1705	12172	118228	262718	265		5331	5855		
MaxHS	395	28091	31184	124795			31204	8665	4772	8
WMaxCDCL	1561	3150	7349	27103	84320	39035	12234	181274		8
WMaxCDCL-bandall	3195	3150	7349	27103	84320	39035	12234	179641		8
UWrMaxSat		611					4			8
EvalMaxSAT										16
CGSS		4729	1940	117190						8
textscExact							3887			8
z3maxres		32542	82538	35633		394407	2226			8
z3wmax		374	2497	8793	1232	394416			732843	8
z3rc2		240668	100737	31865		394410	646			8
z3pd-maxres	1	32501	82578	35645		394411	2224			8
z3maxres-bin		32559	82076	35632		394410	2048			8
Pacose	275911	100	9825	633	8327		488	268	3111	8
maxpre2EvalMaxSAT	47821			69915	909422					8
# faults	335856	415080	668238	1402529	1087886	2050124	98523	395057	740726	121
# faulty solver	9	14	13	14	6	7	13	7	3	15