MaxSAT Fuzzing and Delta Debugging

TOBIAS PAXIAN, University of Freiburg, Germany
ARMIN BIERE, University of Freiburg, Germany

This article presents the first systematic study to evaluate a suite of automated fuzzing techniques for Maximum Satisfiability (MaxSAT) solvers. It combines large-scale stress testing with a novel MaxSAT-specific delta debugging method to assess and improve solver robustness. A parallel framework orchestrates the generation of millions of structured MaxSAT instances. It efficiently isolates failure-inducing input and distills failing cases into minimal counterexamples for precise failure localization. Over a 100-hour fuzzing effort, this approach revealed previously unknown failures in almost all 43 solvers from recent MaxSAT competitions and in three certificate-producing solvers. Failures ranged from crashes and incorrect optimality bounds to severe performance slowdowns. Notably, a critical soundness error was identified in one certified solver. The resulting corpus of minimal counterexamples was published as a public regression suite. This suite was adopted as a mandatory check in the 2024 MaxSAT Evaluation, helping cut average solver failure rates by more than half. The MaxSAT community quickly embraced these resources: some solver development teams have already integrated our fuzzer into their workflows. The complete tool chain and benchmark corpus are available at Zenodo (Paxian 2025a). Our study demonstrates that systematic fuzz testing coupled with targeted debugging can significantly raise the reliability standards of MaxSAT solvers and provide valuable resources for future solver development.

JAIR Associate Editor: Chu-Min LI

JAIR Reference Format:
Tobias Paxian and Armin Biere. 2026. MaxSAT Fuzzing and Delta Debugging. *Journal of Artificial Intelligence Research* 85, Article 9 (February 2026), 39 pages. DOI: 10.1613/jair.1.19716

1 Introduction

Maximum Satisfiability (MaxSAT), an optimization variant of Boolean Satisfiability (SAT) solving, seeks a truth assignment to a Boolean formula in Conjunctive Normal Form (CNF) that maximizes the number of satisfied clauses (Ansótegui et al. 2013; Bacchus, Järvisalo, et al. 2021; C. M. Li and Manyà 2021). In its weighted variant, each clause is assigned a weight, and the objective is to maximize the total weight of the satisfied clauses.

The development of robust and efficient software is crucial due to the ever-increasing demands for reliability and performance in various applications. This is especially true for MaxSAT solvers, which are used in hardware and software verification, constraint programming, and artificial intelligence planning (Berg, Hyttinen, et al. 2018; Berg and Järvisalo 2017; Chen et al. 2010; Ghosh and Meel 2020; Raiola et al. 2020; Seufert et al. 2023; L. Zhang and Bacchus 2012). These solvers need to handle growing complexity and maintain high reliability. Continuous advancements in MaxSAT algorithms, highlighted in the annual MaxSAT Evaluation (MSE) (Bacchus, Berg, et al. 2022), have significantly enhanced the capability to solve increasingly complex problems.

Several methods exist to enhance the reliability of MaxSAT solvers. One approach involves programming the entire solver in a verified programming language, similar to IsaSAT in SAT solving (Fleury et al. 2018). While this ensures correctness, it can be slow due to the need for comprehensive proofs of all techniques used. Another

Authors' Contact Information: Tobias Paxian, ORCID: 0009-0005-2044-1393, paxiant@cs.uni-freiburg.de, University of Freiburg, Freiburg im Breisgau, Germany; Armin Biere, ORCID: 0000-0001-7170-9242, biere@cs.uni-freiburg.de, University of Freiburg, Freiburg im Breisgau, Germany.

DOI: 10.1613/jair.1.19716

approach is to add proofs to the solving process, which can be verified by a proof checker, as demonstrated by early works for unweighted solvers (Gocht et al. 2022; Vandesande et al. 2022). Recently, these proofs have been extended to weighted solvers, such as the unsatisfiability core (Berg, Bogaerts, Nordström, Oertel, and Vandesande 2023) and solution-improving MaxSAT solving (Berg, Bogaerts, Nordström, Oertel, Paxian, et al. 2024). We can integrate these solvers into our fuzz testing to verify that the best found solution is indeed optimal.

Our study uses a dynamic software testing method called fuzz testing, which is a proven way to find software vulnerabilities and failures in different areas (Zeller, Gopinath, et al. 2024; X. Zhao et al. 2024; Zhu et al. 2022). Fuzz testing started with a key paper in 1990 that showed its effectiveness in finding reliability problems in UNIX utilities (B. P. Miller et al. 1990). Since then, fuzz testing has been used in many areas, more about that you can read in the related work Section 2.1. Closest to our work are fuzzing efforts for SAT and a range of SAT-based formalisms with progressively richer expressiveness, including theories, quantifiers, non-monotonic rules, and more complex circuit structures.

Satisfiability Modulo Theories (SMT), which extends SAT with arithmetic and other background theories, fuzzing has been used to find failures in solvers. Brummayer and Biere introduced an approach for fuzzing SMT solvers in 2009, generating random, valid SMT instances, to find differences in solver implementations (Brummayer and Biere 2009). More recently, SMT-Murxla introduced a more advanced SMT fuzzer with full support for the semantics and features of SMT-LIB (Niemetz et al. 2022). Another approach uses mutation-based fuzzing to detect critical failures in SMT-Solvers (Mansur et al. 2020)

In logic programming, Answer Set Programming (ASP) fuzzing has been explored to improve solver reliability by generating valid ASP instances (Brummayer and Järvisalo 2010; Mushthofa et al. 2016). Reducing test cases to find the root cause of failures is an important complement to modern fuzz testing techniques. C-Reduce, for example, simplifies compiler failures by iteratively reducing test cases while keeping their failure-inducing properties (Regehr et al. 2012).

Fuzz testing has also been used in SAT and Quantified Boolean Formula (QBF, adding alternating quantifiers to SAT) solvers. Brummayer and Biere expanded their fuzzing approach to cover both SAT and QBF solvers (Brummayer, Lonsing, et al. 2010). In SAT fuzzing, the aim is to generate random SAT instances that expose inconsistencies or weaknesses in solver implementations, helping to improve solver stability by uncovering edge cases. QBF fuzzing introduces additional complexity due to the presence of quantifiers, making it important for evaluating solver performance under quantifier alternations and identifying specific challenges related to quantifier interactions.

Kaufmann et al. explored fuzzing for And-Inverter Graph (AIG, checking equivalences in Boolean circuits) verification, focusing on generating challenging test cases that stress verification tools and reveal subtle failures in AIG transformations and optimizations (Kaufmann and Biere 2022). These methods are closely related to our MaxSAT fuzzing approach, as they target the robustness and reliability of SAT-based solving techniques, providing insights that are directly applicable to enhancing MaxSAT solver effectiveness.

In this work we employ what we term *generation-based, grammar-aware black-box fuzzing*. A single driver script takes the path to a MaxSAT executable as its sole argument, so swapping solvers is literally a simple change in the configuration script, making large-scale head-to-head testing effortless. Each random instance is generated in *Weighted Conjunctive Normal Form* (WCNF) also known as the WDIMACS standard, the plain-text, clause-per-line standard used in the MaxSAT Evaluation, which every solver in our study can parse out of the box (both versions of the standard).

For every instance we launch a *portfolio* of solvers and record the smallest objective value among the returned models as the candidate optimum. Whenever at least one solver produces a correctness certificate we pipe it to an independent verifier, thereby upgrading the candidate optimum to a *proved optimum*. In all cases we additionally invoke an off-the-shelf SAT solver to ensure that the hard clauses are satisfiable and that each model satisfies the full formula and its reported objective value (o-value). This workflow simultaneously stresses the solvers'

optimization logic, model printing, and proof infrastructure, while guaranteeing that any best-of-portfolio result is either provably optimal or flagged for further inspection.

A silent but critical certified failure. To see why our cross-checks matter, consider the failure shown in Figure 12: one solver claims to have found a optimum with an objective value and a matching model together with a certificate that its own verifier accepts, yet a second solver on the same instance later produces a model with a strictly better cost. The first solver is *self-consistent but wrong*—a so-called *soundness failure* classified as 2.1 in Figure 3 with other examples shown in Figure 11. These errors quietly invalidate results and therefore rank as the most severe failures our fuzzer detects.

We further use delta debugging to strengthen our testing methodology. By systematically reducing the size of failure-inducing instances, delta debugging helps identifying the smallest set of soft and hard clauses causing the issue. Our novel tool, `WCNFddmin`, extends the methods of existing SAT, QBF, and SMT delta debuggers by introducing new reduction phases and techniques. This allows us to generate minimal yet effective test cases that conform to MaxSAT instance rules, uncovering failures through minimizing, a kind of mutation, that might not be detected through fuzzing alone, as detailed in Section 4.4.

In our earlier POS article (Paxian and Biere 2023) we **(i)** introduced the first grammar-aware MaxSAT fuzzer (WCNFuzz) that uncovered failures in *all but one* weighted solvers of the 2022 MaxSAT Evaluation, **(ii)** shrinks failing inputs with a prototype delta-debugger, and **(iii)** ships a preliminary shell script *Regression Suite* of those minimized instances that the community could run in seconds.

The present paper keeps all aforementioned results and adds five substantial extensions:

(1) **Improved delta debugging.** We provide a fully engineered tool (WCNFddmin) with six reduction phases, clause/literal shuffling, variable renaming, and on-the-fly failure logging. It now runs autonomously inside the fuzzer and contributes up to *18 % additional* unique failures. It was published shortly before the MaxSAT Evaluation 24 hand in deadline and advertised by the Organizers [1] of the Evaluation.
(2) **Stronger instance generation.** An improved instance generation tool now written in python called (PaxianPy with variants small, tiny), produces much smaller yet harder formulas, quadrupling throughput while preserving failure diversity.
(3) **In depth fuzzing of multiple MSE tracks.** We analyze our fuzzers on all exact weighted solvers from MSE22/23/24 and anytime solvers from MSE23 and, showing that the mandatory Regression Suite cut the average failure rate by more than half of the weighted track.
(4) **Report on community impact.** The regression suite became a *required* check for MSE24 submissions; four solver teams have already reported adopting our fuzzer during development.
(5) **Certified-solver case study.** We reveal a critical bound-violation in the proof-logging solver CGSS: the certificate verifies yet the model is sub-optimal, questioning blind trust in current certificates.

Scope. Our primary goal is to evaluate how effectively the proposed fuzzers generate high-quality test instances; the solver failures we report are valuable, but they should be regarded as illustrative by-products rather than the main target of the study.

These additions answer all research questions set out in Section 5: instance quality (RQ1), comparison with prior generators (RQ2), the extra failure coverage brought by delta debugging (RQ3), the effect of the Regression Suite on solver robustness (RQ4), and the behavior of anytime as well as certified solvers (RQ5).

The original motivation for testing MaxSAT solvers stems from our university course, "Debugging and Fuzzing." Students were tasked with creating their own delta debugger and fuzzer. We received functional delta debuggers from nine students, all of which reduced instances at the clause level. Seven students submitted fuzzers, some

[1] Tobias Paxian, the author of this paper was one of the Organizers.

incorporating innovative ideas. The most promising one is participating in this work's instance generation comparison within our experimental setup.

2 Related Work

This section reviews existing research on fuzzing and delta debugging techniques that are relevant to our study.

2.1 Related Work on Fuzzing

Fuzz testing, or fuzzing, is a widely used technique to identify software vulnerabilities by generating or mutating inputs to a system. There are various approaches to fuzzing, each with its own strengths and weaknesses:

- *Black Box Fuzzing:* This approach treats the system under test as a "black box," meaning the internal workings are not known to the tester. Inputs are generated randomly without any knowledge of the program structure. This method is straightforward and easy to implement as no insights to the original code is needed (Ghasemisharif 2018; Pferscher and Aichernig 2022).
- *White Box Fuzzing:* Unlike black box fuzzing, white box fuzzing involves understanding the internal workings of the program. It uses code analysis techniques to generate inputs that can explore different paths in the code. While this method is more effective in finding failures, it is also more complex and requires access to the source code (Godefroid, Kiezun, et al. 2008; Godefroid, Levin, et al. 2012).
- *Mutation-Based Fuzzing:* This approach starts with a set of valid inputs and creates new test cases by making small modifications, or mutations, to these inputs. Mutation-based fuzzing is effective in testing how well a system can handle slightly altered inputs, which can often reveal hidden failures (Bhattacharjee et al. 2024; Lemieux and Sen 2018; C. Miller and Peterson 2007).
- *Generation-Based Fuzzing:* This method involves creating inputs from scratch based on a specification or model of the input format. This approach can be more thorough than mutation-based fuzzing as it can generate a wider range of inputs, including those that follow the input format's rules and those that intentionally break them (Liu et al. 2025; Paxian and Biere 2023).
- *Grammar-Aware Fuzzing:* A specialized form of generation-based fuzzing, grammar-aware fuzzing uses the grammar of the input language to generate syntactically valid test cases. This approach is particularly useful for testing systems that process structured input data, such as compilers or interpreters (Sochor et al. 2024; Srivastava and Payer 2021; Vasylenko 2024).
- *Command Line Argument Fuzzing:* This approach tests how a program handles different command line arguments. By changing the inputs given via the command line, testers can find vulnerabilities and ensure the program can handle a wide range of inputs (Gupta et al. 2022; Z. Zhang et al. 2023).
- *API Fuzzing:* API fuzzing tests the Application Programming Interfaces (APIs) of a system. It sends various unexpected or malformed inputs directly to API endpoints to find security flaws or crashes. This method ensures the security and reliability of web services and other systems that expose APIs (Atlidakis et al. 2019; Mahmood et al. 2022).
- *Coverage-Guided Fuzzing:* This technique uses code coverage information to guide test input generation. By analyzing which parts of the code are executed, it aims to explore more of the code-base, increasing the chance of finding failures. There are several tools like AFL (American Fuzzy Lop) and libFuzzer which are popular for this approach (Golla and Godboley 2024; Y. Zhang et al. 2022).

These are the most well-known types of fuzzing, though other methods exist. Our approach can be classified as *Generation-Based, Grammar-Aware Black Box Fuzzing*.

Our approach combines elements of black box fuzzing, random input generation, and grammar-aware techniques. It generates random test inputs based on the grammar of the input language without knowledge of the

program's internal structure. We aim to create valid and meaningful inputs that can effectively test how the system handles different input scenarios while remaining easy to implement due to its black box nature.

To get a better idea of different fuzzing methods, we look at detailed surveys that explains and categorizes many fuzzing techniques (X. Zhao et al. 2024; Zhu et al. 2022). These surveys helps us understand the various strategies used in fuzzing and their uses in different areas.

We are aware of only two publicly available MaxSAT fuzzing tools that support solely the old pre-MSE22 WCNF format. The first fuzzer by Norbert Manthey (Manthey 2020) primarily tests solvers for invalid exit codes, missing or invalid printed objective values (o-values), but does not verify if the solver-reported o-value matches the provided model or if the solution is optimal. The second publicly available fuzzer, developed by Mate Soos (Soos and Meel 2021b), accompanies the MaxSAT solver GaussMaxHS (Soos and Meel 2021a), which has not yet participated in the MSE. This tool generates CNF formulas, enhances complexity by adding XOR gates, and then converts these to WCNF using bit-blasting via Python and shell scripts. Additionally, a third fuzzer by Florian Pollitt, developed within a student project during a university lecture, generates random instances featuring a layered structure but limited complexity. To our knowledge, none of these fuzzers implement checks for the optimality of reported solutions nor incorporate delta debugging techniques or other input-shrinking methods to minimize failing test cases.

2.2 Related Work on Delta Debugging

Delta debugging is a powerful technique to isolate and simplify failure-causing inputs in software testing (Zeller 2006; Zeller and Hildebrandt 2002, 2025). By systematically removing parts of the input, it aims to identify the minimal subset that still triggers the failure. This method helps to pinpoint the root cause of a problem by progressively reducing the size of the test input while maintaining the failure-inducing property, thereby making it an efficient tool for debugging software.

Several advanced methods similar, build on or extend traditional delta debugging:

- *QuickXplain:* Similar to delta debugging, it focuses on conflict-driven minimization with a divide-and-conquer approach to find minimal failure-inducing inputs (Junker 2004; Rodler 2022; Tazl et al. 2022).
- *Hierarchical Delta Debugging (HDD):* This method structures the input hierarchically and reduces it at multiple levels, making it efficient for complex inputs (Misherghi and Su 2006).
- *Hoisting:* An extension of HDD, hoisting reduces parse trees by replacing subtrees while maintaining syntactic correctness, useful for structured inputs like code (Vince et al. 2021, 2022).
- *C-Reduce:* Used for C/C++ programs. It employs several reduction strategies, including delta debugging, to minimize inputs while keeping the failure-triggering behavior (Regehr et al. 2012; M. Zhang 2025).
- *Probabilistic Delta Debugging:* Uses Bayesian optimization to estimate each element's relevance and adaptively delete statistically chosen subsets, achieving state-of-the-art input minimization; the simplified CDD variant retains the same efficiency with a leaner model (G. Wang et al. 2021; M. Zhang et al. 2025)
- *Grammar-Aware Delta Debugging:* This technique incorporates grammar rules into delta debugging to handle inputs with specific structures more effectively (Eberlein et al. 2023; Y. Zhao et al. 2024).

These methods can be selected based on the input's nature and the minimization task's requirements. A detailed survey on delta debugging techniques offers further insights into these methods (Njeru et al. 2017).

In our work, we employ *Grammar-Aware Delta Debugging* for MaxSAT solvers. This approach is particularly suited for handling MaxSAT problems. By integrating grammar-awareness, our method generates valid and meaningful reductions and produces only correct MaxSAT instances, which in the end preserve the failure-inducing properties. In addition, we introduce new variations and extensions to enhance the effectiveness of delta debugging in this context. These improvements explained in Section 4.4 allow us to thoroughly minimize MaxSAT solver benchmarks.

3 Preliminaries

Fuzzing is a dynamic testing method used to discover software vulnerabilities and failures by providing random or semi-random inputs to a system. It aims to find inputs that cause unexpected behavior, crashes, or security issues. Delta debugging is another technique employed to isolate and minimize failure-inducing inputs by systematically reducing the input size while preserving the failure-inducing properties.

3.1 Generation-Based, Grammar-Aware Black Box Fuzzing

Our approach, *Generation-Based, Grammar-Aware Black Box Fuzzing* (Brummayer, Lonsing, et al. 2010; C. Miller and Peterson 2007; J. Wang et al. 2018; Yu et al. 2024), merges several key fuzzing strategies to maximize test effectiveness. This method generates inputs from scratch based on specific rules or models. These inputs adhere to the grammar of the input language, ensuring syntactically correct and meaningful test cases. This approach treats the system under test as a black box, meaning no prior knowledge of the internal workings is required.

By generating inputs according to the input language grammar, we ensure that the test cases are valid and meaningful, especially for systems that process structured data. This method is particularly useful for testing MaxSAT solvers, which solve the Maximum Satisfiability Problem. For MaxSAT, we generate inputs in the form of weighted CNF (Conjunctive Normal Form) instances and test the solvers' ability to find optimal solutions. Our grammar-aware approach ensures these inputs conform to the WCNF format, thus enabling thorough testing of the solvers' robustness and efficiency.

Using this technique, we can generate a wide variety of test cases without needing detailed knowledge of the program's internal structure. This makes it a powerful tool for uncovering vulnerabilities and ensuring the reliability of MaxSAT solvers.

3.2 Delta Debugging

Delta debugging (Misherghi and Su 2006; Zeller 2006; Zeller and Hildebrandt 2002) is a powerful and efficient technique to isolate and simplify failure-causing inputs in software testing. It systematically reduces the size of a test input while preserving the failure-triggering property, helping to identify the root cause of a problem. The algorithm operates within linear complexity without restarts and up to quadratic number of tests with restarts, which try to remove parts of the input several times.

The delta debugging algorithm follows these steps:

(1) *Initialization:* Start with a failure-inducing input formula ϕ for a MaxSAT solver and try to remove portions to create a simpler input that still causes the failure.
(2) *Division:* Divide the input into smaller chunks. Initially ϕ is divided into two halves.
(3) *Testing Subsets:* Test the MaxSAT solver with the current formula ϕ, while leaving out each candidate subset independently for evaluation:
    - If a formula without a candidate subset still causes the failure, that subset becomes an eliminated subset and is removed for all subsequent solver calls from ϕ. Continue by trying to leave out the next subset from the new formula.
    - If all subsets are tested without failure, halve the size of the chunks and divide the current formula into chunks of this new size. Repeat from step 3.
(4) *Refinement:* Continue subdividing and testing the input until the smallest possible failure-inducing input is found. In the worst-case scenario, each atomic element (e.g., a clause or literal) needs to be tested individually. If we cannot remove a single atomic element and we have a size of n elements, we need $2n$ MaxSAT solver calls in the worst case. Note that this greedy algorithm is not guaranteed to always find the smallest failure-inducing input formula.

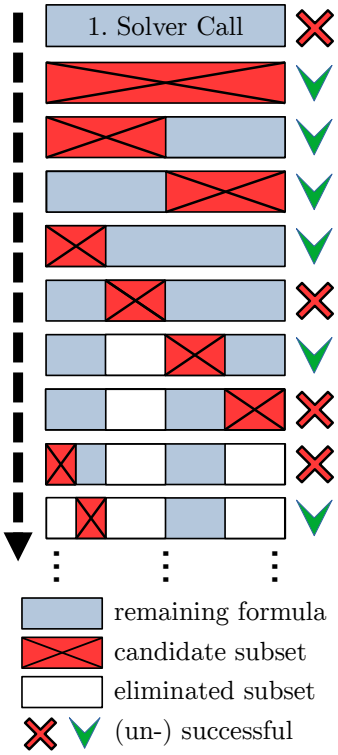


Fig. 1. Illustration of the solver calls in the Delta Debugging Algorithm.

This greedy approach systematically removes parts of the input that do not contribute to the failure. Initially, it attempts to remove larger parts, then progressively smaller parts, until it narrows down to the smallest subset responsible for the failure.

Example 3.1. Figure 1 illustrates the idea of the delta debugging algorithm. Let us assume we have a formula consisting of 8 clauses, denoted as $\phi = \{C_1, \dots, C_8\}$. The smallest granularity for this delta debugging run is a clause, without distinguishing between soft and hard clauses.

Initially, we test whether the whole formula induces the failure. If so, we try to remove the whole formula, but let us assume, as in the figure, that the empty formula does not cause a failure. Then, we divide the formula into chunks of size 4. We start by leaving out $\{C_1, \dots, C_4\}$ as the first candidate subset, and then $\{C_5, \dots, C_8\}$, and test resulting instances with a solver.

Since neither subset left out alone induces the failure, we proceed by testing combinations of 6 clauses, leaving out subsets of two clauses. Assume that leaving out $\{C_3, C_4\}$ induces the failure for the first time after testing the whole formula. Then, this candidate subset becomes an eliminated subset and is permanently removed from $\phi = \{C_1, C_2, C_5, \dots, C_8\}$. We continue testing by leaving out the next subset $\{C_5, C_6\}$, and then $\{C_7, C_8\}$, which is the next to induce a failure. Remove these clauses as well, resulting in $\phi = \{C_1, C_2, C_5, C_6\}$.

Next, we decrease the granularity to a single clause and test the formula by leaving out one single clause at a time. Leaving out the first clause is the last action producing a failure, as shown in Figure 1. After these four solver calls, the algorithm returns $\phi = \{C_2, C_5, C_6\}$ as the smallest formula.

Because the procedure explores candidates in a greedy, partition-based order, the returned subset is only locally minimal with respect to that search path; an even smaller subformula—potentially a single clause—may still exist that provokes the same failure.

4 Methodology

In this section we introduce the key components of the study. We follow a five-step pipeline—generate, execute, classify, minimize, regress—to uncover and analyze failures in modern MaxSAT solvers.

Terminology. The software-engineering literature distinguishes *errors*, *bug/faults/defects*, and *failures* (Duarte et al. 2018; Krawiec 2018), but, following Zeller's practice in systematic debugging (Zeller 2006; Zeller and Hildebrandt 2002), we use only the word **failure**: any observable, unwanted solver behavior such as a crash, wrong model, incorrect optimum, timeout, or runaway memory use. The underlying "bug" is irrelevant to the fuzzer; what matters is that a concrete input triggers the misbehavior.

Overview: The remainder of this section is organized as follows:(1) we introduce our fault-classification scheme, (2) describe the instance-generation strategies that improve failure coverage,(3) explain how we compare solvers and log results, (4) detail the delta-debugging algorithm used to minimize failing instances, and (5) present the regression suite that guards against future regressions.

4.1 Fault Classification

Fuzzing a MaxSAT solver surfaces a broad spectrum of failures—from outright crashes to subtle logical mistakes. A clear taxonomy groups these issues, lets us compare solvers on equal footing, and directs debugging effort where it matters most. The worst case—a silent *bound violation* (class 2.1 in Fig. 3)—is when a solver proudly reports "OPTIMUM FOUND" and supplies objective value with a matching model, yet the result is not optimal. Because such errors quietly invalidate results, we put them—and the other classes—on a systematic footing in the remainder of this subsection.

First, we examine how failures in MaxSAT solvers can be classified. Categorizing these failures provides a basis for comparison with other works and helps in discussions about the severity and nature of different failures. This foundation is essential before differentiating and categorizing them further. Consequently, we evaluate the types of failures that can occur in a MaxSAT solver and introduce four main failure classes: Crashes, Lower/Upper Bound Violations, Performance Regressions, and Other Issues. Our fuzzing toolkit's compare tool, detailed in Section 4.3, is finally responsible for differentiating these failures and categorizing them accordingly.

1. **Crashes** refer to invalid return codes of a MaxSAT solver as segmentation failures, arithmetic errors, bus failures or inconsistencies. These return codes can even be used by a developer to test if certain branches can be triggered by the MaxSAT fuzzer or by a specific Benchmark. This can be very useful to find certain failures either by introducing assertions into the code or exit directly in the code with a forced exit code. In pre-MSE24 rules the exit codes were not defined and since MSE24 '0' (unknown), '10' (satisfiable, but not yet proven optimal), '20' (unsatisfiable) and '30' (optimum found) were added.
2. **Bound Violations** are inconsistent results between the solver's output and the expected results. This includes situations where the solver's reported optimal value does not match the value calculated from the model it provides. For exact solvers, the reported result must be the best possible solution. For anytime solvers, the results are acceptable as long as the solver's reported value matches the value derived from the model. Additional violations occur if the solver incorrectly reports the satisfiability of the instance.
3. **Performance Regressions** include unusual solver timeouts or a very high memory usage in comparison to the other solvers. These issues may not be correctness issues in the solver but should be checked to ensure there are no infinite loops or memory leaks in the code.

Fig. 2. Refined illustration on how the Bound Violations 2.1-2.3 are classified. Note o_{model} can never be smaller than o_{min}. All possible value combinations of the three o-values are presented.

4. **Other Issues** are inconsistencies or errors in the solver output, such as a model line with many variables not in the original formula or other unexpected behaviors. Parser issues also fall into this category.

We considered adding another category focused on parser robustness. This could be triggered by a mutation-based fuzzing approach that does not take the grammar of the MaxSAT solvers' WCNF files into account. However, we concluded that these failures are not as severe as inconsistent results or crashes found using a grammar-aware fuzzing approach. These failures could result in wrong exit codes or wrong solutions, depending on the kind of flaw in the parser. As a result, we decided against including it as a main category but included it as a subsection under 'Other Issues'. If an instance does not fulfill the rules of the MaxSAT grammar, then the misbehavior of a solver should be categorized as a parser issue. This grammar-unaware fuzzing / mutation based fuzzing approach is not part of our work since there are already existing fuzzers that handle this effectively.

Our solver comparison tool `WCNFCompare` categorizes failures using numbers from 1 to 999. All failures from 1 to 255 represent the actual return codes of the MaxSAT solver in case of a crash. Bound violations are assigned three-digit numbers beginning with 5xx, performance regressions start with 3xx, and other issues start with 4xx. Additionally, `WCNFCompare` can distinguish between positive integer weights greater than 2^{32} or smaller by adding 1000 to the aforementioned failure categories. In total, it differentiates about 40 different failures which are all translated into the categories of Figure 3.

Objective values (o-values). To pinpoint bound violations we track three objective values—the total weight of unsatisfied soft clauses:

o_{solver} value the solver *claims* is optimal (taken from status line),
o_{model} value that actually follows from the model printed by the solver (computed by our `WCNFtool`),
o_{min} best o_{model} value achieved by *any* solver so far, tested on the same instance.

Whenever a new model with cost $< o_{\text{min}}$ appears we update o_{min}—hence o_{model} can never be strictly better than o_{min} without redefining the minimum. Figure 2 enumerates every combination of o-values and shows how they map to the bound-violation subclasses: 2.2 covers the cases where *all three* values differ, whereas 2.3 covers those where exactly one value deviates from o_{min}.

Most critical failure—class 2.1 (bound violation). Class 2.1 occurs when $o_{\text{solver}} = o_{\text{model}} > o_{\text{min}}$. The solver delivers a self-consistent certificate "OPTIMUM FOUND" with a matching model, yet another solver later provides

Fault Classification:

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 failure)
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 failure)
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} differs from o_{min} and $o_{\text{model}} \neq o_{\text{solver}}$.
2.4. SAT Solver states hard clauses are UNSATISFIABLE but solver states otherwise.
2.5. SAT Solver states hard clauses are SATISFIABLE, but solver states UNSATISFIABLE.
2.6. Provided model falsifies the formula but a model exists.
3. **Performance Regressions (Potential Failures):**
3.1. Solver had timeout, but this timeout is 100 times larger than the average time of the non-timeout solvers.
3.2. Solver had timeout and a very high memory usage, but this memory usage is 100 times larger than the average memory usage of all other solver.
4. **Other Issues:**
4.1. Inconsistency in status line and output.
4.2. Model is too big (>10x the number of variables).
4.3. Solver has an error stated in stdout or printed something in stderr.
4.4. Exact solver status line is 's UNKNOWN', indication that algorithm isn't complete.
4.5. Unexpected behavior of a verifier.
4.6. Parser Issues

Fig. 3. Fault Classification of failure classes and their respective failures.

a model with a strictly better o-value. This is the same silent *soundness failure* highlighted earlier and is the hardest to catch: it slips past the usual sanity check (hard-clause satisfiable? model matches cost?) and is revealed only by cross-checking against independent runs. Concrete examples appear in Figures 11 and 12.

Performance vs. crash vs. bound violation. Failure class 3.1 may signal a performance regression (e.g. an infinite loop); class 3.2 can hint at a memory leak. Such issues are *potential* failures until confirmed by code analysis, and they are generally less severe than crashes (class 1.x), where a failure is certain. In order of descending severity we therefore rank:

1) bound violations (2.x),
2) crashes (1.x),
3) performance issues (3.x),
4) other anomalies (4.x).

The sequence in which failures are evaluated during the classification process is vital for accurate detection. The order should minimize the risk of missing failures, an issue in previous versions of the compare script. For instance, solver status should be checked before evaluating the o-value and model. If multiple failures occur in a single solver run, such as an error message alongside a bound violation, we prioritize the bound violation as we

do not interpret error messages. Some solvers, like MaxHS, print error messages but still produce correct results. The approach balances not over-categorizing failures while not overlooking important ones.

4.2 MaxSAT Instance Generation

WCNFuzz is a generation-based, grammar-aware fuzzer (J. Wang et al. 2018) built to stress-test MaxSAT solvers. It quickly spins up weighted-CNF (WCNF) instances that (1) crash solvers, (2) expose slow paths or memory blow-ups, (3) provoke wrong answers, and (4) act as "stress cases" that are simply hard to solve. The rest of this section first contrasts WCNFuzz with earlier SAT fuzzers such as CNFuzz and FuzzSAT, then explains the design choices—clause layering, gate encodings, weight distributions, and special-case modes—that let us generate small yet challenging benchmarks reproducibly from a single random seed.

There are already successful fuzzing tools for CNF formulas, such as CNFuzz and FuzzSAT (Brummayer, Lonsing, et al. 2010). CNFuzz generates structured instances, resulting in problems closer to industrial examples than simply applying a clause-to-variable ratio (Mitchell et al. 1992; Nudelman et al. 2004; Pérez and Voronkov 2005), as done in many studies to generate hard random 3-SAT formulas. Our goal is to construct difficult problems with only a few clauses, as previous studies have shown that hard to solve instances typically trigger most failures (Brummayer, Lonsing, et al. 2010; Zeller and Hildebrandt 2002, 2025). WCNFuzz modifies CNFuzz to generate these WCNF formulas.

How do we categorize a complex, hard to solve MaxSAT problem? A straightforward approach might be to modify CNFuzz by making half of the clauses soft (by adding weights) and the other half hard, which must be satisfied. However, CNFuzz is designed to generate difficult SAT instances, leading to around 50 % unsatisfiable instances, as it aims for a balance between satisfiable and unsatisfiable problems.

The following ideas drive the development of our instance generation tool:

- Instances should be generated quickly and repeatable with an initial seed.
- Decide between normal, small, and tiny instances to determine which type best triggers failures.
- Hard clauses should be almost always satisfiable, but not too easy to solve.
- The set of soft clauses should almost always be unsatisfiable, avoiding an objective value of 0.
- Multiple layers of clauses are desired, as is done in SAT solving. Each layer adds additional variables.
- Tseitin-transformed AND, EQUAL, XOR3 and XOR4 gates are added to our instances.
- Proper distribution of weights, adjustable upper bounds and the option to generate unweighted instances.
- Occasionally, all clauses should be soft.
- Occasionally, all soft clauses should be unit.
- Occasionally, we want to generate generalized boolean multilevel (GBMO) problems.
- Occasionally, weight differences should be picked close to a certain value to introduce variety.

What follows is a closer look at the adjustments:

Research indicates a clause-to-variable ratio of around 4.26 produces about 50% satisfiable random 3-SAT instances (Cheeseman et al. 1991), as shown in Figure 4. We do not use 3-SAT instances, but we select clauses to have size three on average. It still holds that a high ratio produces more unsatisfiable instances and a low ratio more satisfiable instances. With this knowledge we adjust the ratio to ensure hard clauses are almost always satisfiable with a lower ratio and soft clauses should be challenging and not all should be satisfiable, achieved by using a higher ratio. Since we additionally include Tseitin-transformed gates, we refined these values to produce high-quality instances, ensuring hard clauses are typically satisfiable and avoiding trivial SAT checks where all soft clauses are satisfied.

Our tool WCNFuzz adds up to 10 layers of clauses, each containing up to 70 variables depending on the version. Layers consist entirely of either hard or soft clauses. Soft clause layers are chosen randomly, with a higher chance initially until the first soft clause layer is selected, and a lower chance for the following layers. Clauses in the

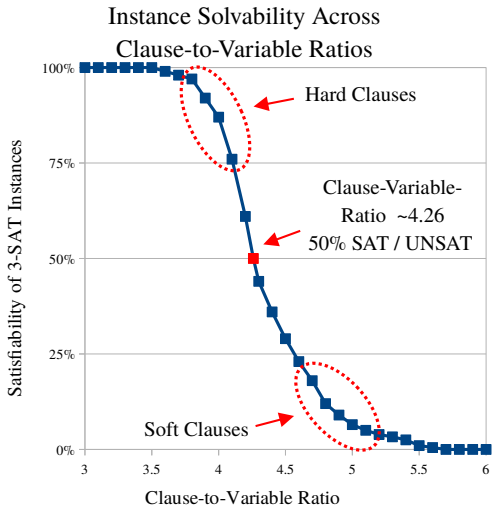


Fig. 4. Number of 3-SAT instances solved at each clause-to-variable ratio, highlighting the thresholds used for hard (mostly satisfiable) and soft (mostly unsatisfiable) clause sets.

n'th layer contain variables from their own layer with high probability and variables from lower layers with lower probability. New variables of the current layer not yet occurring in other clauses are chosen first. As this is randomly chosen, this may result sometimes in a variable which does not occur at all in the whole instance.

Clause lengths are chosen with an exponentially decreasing chance to be longer than three, and similarly for lengths shorter than three. This works as follows: The initial length of a clause is set to three. With a probability of 1/3, we increase the size by one until either reaching the maximum clause length of 20 or stopping the increase loop. But reaching that length is already a incredibly small probability of $\frac{1}{3}^{16}$. If the clause remains size 3, we decrease the size by one with a probability of 1/10 until either reaching the minimum clause size of one or stopping the decrease. Thus, 9/15 of the clauses are ternary, with a 1/3 chance of being longer and a overall 1/15 chance of being shorter than three. We calculate the number of clauses in each layer by selecting a suitable clause-to-variable ratio. To ensure hard clauses are satisfiable with a high probability, we pick a low ratio $r \in [1, 3]$ for hard clause layers. For soft clause layers, to have some clauses making the problem unsatisfiable, we choose a high ratio $r \in [4.5, 7]$. These values are based on empirical testing and vary by program version.

Additionally, we add Tseitin-encoded Equality, AND, 3-XOR, and 4-XOR gates. We include an activation literal in 3/4 of the gate encodings and add one additional soft clause containing only the negated activation literal. Variables for these encodings are chosen randomly from all available variables. The number of gates depends on various circumstances: more layers result in fewer gates, and the more gate types are added, the less likely others will be added too.

The maximum weight in the MSE is often small or unweighted but always a positive integer. Consequently, we also choose the maximum weight to be in one of the following ranges, each with a probability of 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 a probability of $1/25$, it is in the range of $[2^{32} + 1, 2^{63} - 1]$, with $2^{63} - 1$ being the maximum possible weight. We also ensure that the maximum sum of weights is less than $2^{64} - 1$, as described in the official MSE rules (Bacchus, Berg, et al. 2022). Other upper bounds can be set optionally.

Every fourth instance contains only unit soft clauses, and one out of ten instances is forced to contain only soft clauses. In the version with only soft clauses, the gates also contain only soft clauses without activation literals.

In summary, the steps described so far capture the baseline functionality of `WCNFuzz`. Building on that baseline, the Python version introduces several enhancements as described in the following.

In the Python version, we can produce a multilayer problem approximately every 12th instance. Each layer can contain weighted clauses and can be solved separately while ensuring the optimal solution of the previous layer is maintained. This technique is called generalized boolean multilevel optimization (GBMO) (Paxian, Raiola, et al. 2021). Additionally, every 17th instance is generated with tight weight bounds, less than 10% around an initially chosen weight.

In the result Section 6 we discuss the adjustments, which are made to these numbers to ensure instances trigger as many failures as possible. What measurements can indicate instance quality? As mentioned, we want challenging problems that trigger special cases or unusual paths in the solver code, while also running as many instances as possible in a short time. How small can instances be while still triggering interesting failures in solvers? We decided on several metrics to indicate instance quality besides the actual triggered failures over time. We aim for:

(1) Triggering as many failures as possible over time.
(2) Few clauses, but still hard-to-solve instances.
(3) Hard clauses should be satisfiable most of the time.
(4) Soft clauses should be unsatisfiable most of the time.
(5) Sufficient diversity in the instance generation process.

The realization for measuring all these attributes are described in our fuzzing framework Section 4.5.

4.3 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 one or 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 the set of hard clauses, and checks the o-value against the model for each solver result, using our `WCNFTool`. We use the best solution we found a valid model for, as a representative for the unverified optimal outcome. If other exact solvers do not produce the same o-value, it indicates an erroneous result. If this solution is produced by a certified solver and the produced proof could be successfully verified, we state that the best found solution is indeed a verified optimum.

The comparison tool tags every run with two identifiers: the solver's index and the failure category (enumerated $1 \ldots 999$ as defined in Section 4.1). Crashes are detected via abnormal process termination, whereas the four MaxSAT-Evaluation status codes introduced in 2024—0 (unknown), 10 (best-known model), 20 (unsatisfiable), and 30 (optimum proven)—are treated as normal outcomes. Because the correspondence between exit codes and failure categories is stored in a configuration file, the tool can reproduce either the legacy or the current competition policy without code changes.

After processing all solvers, `WCNFCompare` emits a single exit value computed as

$$\left(\sum_i \left(\text{fault\_code}_i + \text{solver\_index}_i\right)\right) \bmod 255 + 1,$$

where 255 is the highest exit code available. Different failure combinations can therefore collide to the same number, which is acceptable for batch statistics but undesirable during delta debugging. To avoid ambiguity in that setting, the script offers a flag that restricts the reported value to the exit code of one designated solver, ensuring that every reduced instance still reproduces the original failure unambiguously.

For logging, individual files for each WCNF-solver-fault combination are generated. 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 failure 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 failure 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.

The `WCNFCompare` tool supports exact, anytime, certified MaxSAT solvers and MIP solvers as it can translate the WCNF into the MPS format used by MIP solvers. It further handles both, the old and new versions of the WCNF format as input or output. This flexibility allows `WCNFCompare` to be used in a wide range of MaxSAT solving and fuzzing scenarios.

4.4 Delta Debugging

When a fuzzer stumbles upon a failure, the offending instance is often far larger than necessary: dozens of clauses or variables may be irrelevant to the misbehaviour. *Delta debugging* is an automated "zoom-lens'' that trims that excess. Starting from the full formula, it repeatedly removes or rewrites pieces—whole layers, single clauses, even individual literals—while re-running the solver after each edit. If the failure persists, the change is kept; if the failure disappears, the change is rolled back. After many such trials we obtain a *1-of-a-kind, minimal counter-example*: the smallest known WCNF instance that still reproduces the bug and therefore pinpoints exactly what the solver cannot handle.

In practice, this reduction serves three purposes. First, it accelerates manual debugging: a six-clause witness is far easier to inspect than a six-hundred-clause monster. Second, it yields compact regression tests that future solver versions must pass in milliseconds. Third, the rewriting itself acts like a mutation-based, grammar-aware fuzzer: because the algorithm explores many near-by variants, it occasionally uncovers *new* failures not found by the original generator. The details of our six-phase reduction strategy:

(1) Removing Clauses: Attempts to remove as many clauses as possible, without distinguishing between soft and hard clauses.
(2) Removing Variables: Iterates over all variables in the problem instance and tries to remove them. 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 list and applies the algorithm to this list. The printing function ensures that literals are correctly removed.
(4) *MaxSAT-specific* Converting Soft to Hard Clauses: Attempts to convert as many soft clauses into hard clauses as possible. Soft clauses are generally more challenging for a MaxSAT solver, making the optimization problem easier to understand with more hard clauses.
(5) *MaxSAT-specific* Weight Reduction to 1: Reduces as many weights as possible to 1, as larger weights are typically harder for the solver or human debugger to handle.
(6) *MaxSAT-specific* Binary Weight Reduction: Systematically lowers the weight of soft clauses using a binary search approach. If multiple weights are selected the weight is halved for each weight. A real binary search is only performed if a single weight is processed. This phase is the most computationally expensive and

in the worst case: $\mathcal{O}(\log(\text{max weight}) * \text{soft clauses})$. Due to that we only perform the reduction until the upper bound minus the lower bound are less than 10% of the original weight.

Techniques 1 to 3 are known from CNF delta debuggers like CNFddmin (Brummayer, Lonsing, et al. 2010), while the additional phases are novel. Removing Literals is very time consuming, due to that we leave this technique out in the first round. To our knowledge, we are the first to apply these techniques repeatedly in a delta debugger, stopping only when no further reductions for a technique is possible or when progress for a technique falls below a specified threshold. The program ends if all techniques are finished. In between those delta debugging rounds, our tool also introduces new features to enhance the debugging process:

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

These techniques require additional solver calls after each shuffling or renaming step. Variable renaming is performed only if some variables do not occur in the WCNF. To ensure we produce valid input instances, we exclude the initial reduction step that reduces instances to an empty instance. While empty instances should be accepted by a MaxSAT solver, they are not yet a standard. Consequently, we have chosen to omit them from our reduction process. `WCNFddmin` can be interrupted by a SIGTERM signal, which waits for the current solver call before terminating the program. If any reduction has been made, it saves the smallest instance found so far.

The unique features and $\mathcal{O}(n)$ complexity (for most phases) make `WCNFddmin`'s an advanced tool for isolating and simplifying failure-inducing inputs in MaxSAT problems. Additionally, during the reduction process, `WCNFCompare` is used to log errors and save failure-triggering WCNFs. We are the first to log errors and saving failure-triggering WCNFs arising during the reduction phase. This helps uncover new, interesting solver-fault combinations that might have remained undetected otherwise.

4.5 Fuzz Testing Framework Overview

Figure 5 sketches the architecture of `runWCNFuzz`, our parallel fuzz-testing pipeline for MaxSAT solvers. The framework orchestrates three tasks in a tight feedback loop: *instance generation* by one or several fuzzers, *solver comparison and failure detection* via `WCNFcompare`, and *instance minimisation* through the delta debugger `WCNFddmin`. All data flow—raw instances, solver output, and reduced witnesses—is routed through `WCNFtool`, which converts between legacy and current WCNF as well as MPS encodings.

In more detail, `runWCNFuzz` is notable for several reasons. Firstly, it processes generated or given MaxSAT instances in parallel. It supports running multiple MaxSAT instance generation tools in parallel, selecting the one with the least processing time to generate the next instance. It integrates closely with `WCNFCompare`, running multiple MaxSAT solvers sequentially and comparing their output. Errors occurring in `WCNFCompare` are counted and displayed on the terminal for each fuzzer-solver-fault combination. For each solver-fault combination, it creates a log file containing information about the failure-triggering fuzzer, the seed, the failure cause, a failure description, and the solver's output for each occurrence of the failure. Additionally, it automatically starts the delta debugger for up to five times for each solver-fault combination, unless specified otherwise via command line. It checks if the reduction was successful or if further delta debugging is required. Each solver-fault combination occurring during the delta debugging process is logged. If a new solver-fault combination that did not occur before is found, the delta debugger is called recursively with that new solver-fault combination, starting with the smallest failure-triggering instance found during the reduction process.

A single settings file configures all solvers and instance generation tools, specifying the type of each solver (certified, exact, anytime, and MIP solver). The script can be interrupted with Ctrl+C, after which all solver and delta debugging processes are terminated, first with SIGTERM and then with SIGKILL if necessary. It then cleans up and writes a log file with all information about the found delta debugger, solver, fuzzer, error combinations

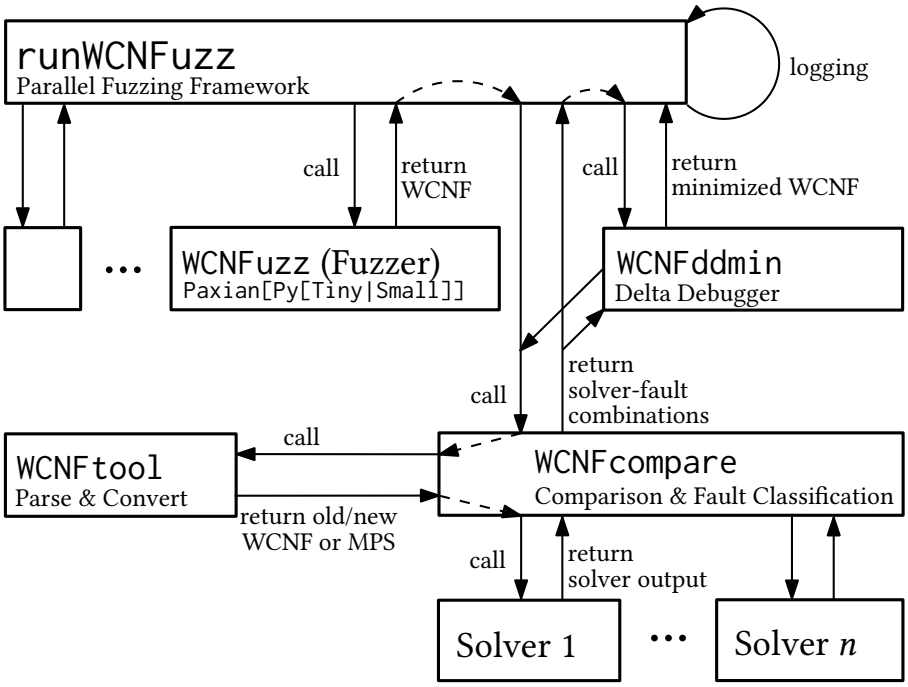


Fig. 5. Parallel architecture of the runWCNFuzz framework. Multiple fuzzers (e. g. WCNFuzz) generate test instances in parallel. Each instance is passed to WCNFcompare, which invokes all selected MaxSAT solvers and uses WCNFtool to parse or convert between legacy and current WCNF as well as MPS formats. Whenever a solver–fault combination is detected, the offending instance is forwarded to the delta debugger WCNFddmin, which iteratively minimises the input by repeatedly calling WCNFcompare. Newly discovered bugs are additionally reported back to runWCNFuzz. All components (fuzzers, comparison, conversion, and delta-debugging tools) can also run stand-alone but are orchestrated in runWCNFuzz as one integrated, parallel testing pipeline.

in a parallel fuzzer log, the delta debugging log files, the log files of all occurred solver-fault combinations, all minimized instances, and all saved unreduced WCNF files that produced errors (up to twice the amount of reductions for each failure). All these log files are saved in a subfolder named after the current date and number of seconds since midnight.

runWCNFuzz also offers additional features, most of which can be activated via the command line:

- It analyzes the generated instances for each fuzzer, including the number of hard and soft clauses, variables, maximal weight, sum of weights, and best o-value found, each with min, max, and average values. Additionally, it calculates the percentage of instances containing hard clauses, soft clauses, unweighted instances, instances with a sum of weight greater than 2^{32}, instances where all soft clauses are satisfiable, and instances where all hard clauses are satisfiable.
- For live fuzzer improvement, it displays these metrics for the last xx instances, allowing immediate feedback on changes made to the fuzzer code. This facilitates adjusting the fuzzer, live during testing, towards the desired statistics as explained in Section 4.2.
- It analyzes all solvers given to the WCNFCompare script by min, max, and average time and memory consumption, and counts the number of timeouts (default timeout is 20 seconds per solver).

- The script is able to differentiate between instances with a sum of weights higher than 2^{32} and lower. Delta debugging minimizes instances without knowledge of these distinctions to identify failures triggered by high weights, possibly indicating an overflow, or failures that can be minimized to lower sums of weights.
- An upper bound for weights of each fuzzer can be set, either by forwarding the number to the fuzzer or discarding instances that do not meet the criterion.
- The script can process all WCNF files in a given folder instead of generating instances.
- All generated or given instances are checked for grammar correctness before calling `WCNFCompare`.
- Instances can be generated/given in the old (pre MSE23) or new WCNF format.
- It can save a CSV file with all generated instances and solver timings. This, combined with the folder processing capability, allows easy comparison of solver timings for either generated or given instances.

Together, these features turn `runWCNFuzz` into a self-contained laboratory for stress-testing MaxSAT solvers. The complete framework—including every standalone tool—is openly available on GitHub (Paxian 2025b) and a snapshot together with logfiles is available on zenodo (Paxian 2025a). The repository's `README.md` provides a step-by-step tutorial.

4.6 MaxSAT Regression Suite

The large-scale fuzzing runs with our `runWCNFuzz` framework (Section 4.5) yielded thousands of failure–inducing instances for every solver submitted to the 2022/2023 MaxSAT Evaluation. For each *distinct* solver-failure pair `runWCNFuzz` applied `WCNFddmin` and retained a minimized instance, eliminating duplicates and variants that triggered the same failure. Whenever possible, the reduced instance was equipped with a certified optimum. The resulting collection forms the core of the *Regression Suite* that has been officially adopted for the 2024 Evaluation and targets both exact and anytime solvers.

Solvers participating in different tracks need to pass specific tests. For example with `--unweighted` only instances with maximal weight of 1 are considered. With `--anytime` only instances which were causing a previous anytime solver to crash are considered. We send a `SIGTERM` to the anytime solver within a short random time limit and check if the model adheres to the given o-value.

Instances can be downloaded directly from the MSE homepage which is done automatically by the install script. Additionally, the script requires access to a valid SAT solver to check if the hard clauses of a given instance are satisfiable, with the default being `kissat`. Other SAT solvers following standard exit codes can be specified with command line options. The `testSolver` script supports various configurations and detailed options to ensure comprehensive testing across different instance types and solver tracks.

The selection criteria for these instances include:

- Minimized instances that triggered failures in any solver during the 2022 or 2023 evaluations.
- Instances solvable within one second by at least one solver from MSE23 or certifiable by proof-logging solvers within 15 seconds.

The suite includes instances where all soft clauses are satisfiable (optimum cost 0) and unsatisfiable instances, as they are generated an minimized. Further the suite can be used to test empty instances or empty soft or hard clauses. It is recommended for any MaxSAT solver being able to handle this kind of instances, but not mandatory to participate in the Evaluation. These kind of instances do not occur in the MaxSAT Evaluation.

From Preliminary to Comprehensive: Evolution of the Regression Suite. In our earlier work, we already created a compact regression suite of small instances by triggering specific errors in all tested solvers. We provided these instances along with a script for executing and verifying the solver's results to the MaxSAT community. Additionally, we proposed new rules to include in the MaxSAT Evaluation rule-book, ensuring the standard handling of basic clauses as provided by our regression suite.

The new version of the regression suite has significantly expanded its capabilities. It now supports not only exact solvers but also anytime solvers, enhancing its applicability and robustness. The new `testSolver.py` script is designed to identify possible issues, such as ”error” messages in the solver output, and handle return codes more effectively. This new Python version replaces the previous bash script, offering a more sophisticated approach to error handling. It can differentiate between issues and errors, compare solver outputs against the best-found solutions (mostly certified), and test the solver against all instances in arbitrary folders containing WCNF files.

The new regression suite is now mandatory and is thus an indispensable tool for all solvers submitted to the main tracks of the MaxSAT Evaluation. Its advanced features and comprehensive testing capabilities make it an essential resource for ensuring solver robustness and reliability.

For step-by-step setup instructions and direct access to the suite, visit the public GitHub repository (Paxian 2025c) and a snapshot together with logfiles is available on zenodo (Paxian 2025a). The `README.md` provides a concise tutorial.

4.7 Standalone Use of Framework Components

Beside evaluating a given set of solvers a potentially even more common usage scenario of our toolset is to help developers of specific solvers developing fault free solvers.

In addition to their integration within the unified fuzzing pipeline, each component of the framework can function as an independent tool. This modular design ensures that the tools are widely applicable beyond our specific experiments. For instance, one can apply the delta debugger to minimize any external MaxSAT instance (e.g., directly shrinking a competition benchmark with `WCNFddmin`) or use the fuzzing and comparison components to test new solvers outside the scope of this paper. The independent functionalities of the main components are summarized below:

(1) `runWCNFuzz` can orchestrate any instance generator that produces WCNF output (in either the old or the new format), with the chosen generator and its parameters configured via `config.py`.
(2) `runWCNFuzz` supports any MaxSAT solver that accepts WCNF input (legacy or current format), and can even target MIP solvers by converting WCNF to MPS on the fly (though MPS-based results are not evaluated in this paper).
(3) `runWCNFuzz` can also process a directory of existing WCNF instances (old or new format) in lieu of generating new ones, enabling automated batch testing and reduction on fixed benchmark sets (e.g., the MaxSAT Evaluation instances).
(4) All variants of `WCNFuzz` can be executed directly from the command line, with or without a user-specified random seed, allowing reproducible generation of WCNF instances.
(5) `WCNFddmin` can be run independently on any given WCNF instance (new format only; including any MaxSAT competition benchmark), with many option e.g. only use reduction phases of variables, clauses, literals or weights.
(6) `WCNFTool` supports format conversion between the legacy (pre-2023) and current WCNF formats, and it can also export instances to the MPS format of MIP solvers.
(7) `WCNFCompare` can be invoked on its own to cross-validate the outputs of multiple solvers (with solver paths specified in `config.py`). As best practice, it is advisable to validate `WCNFCompare` on a small set of instances first before deploying it in a full fuzzing run via `runWCNFuzz`.

Again, more detailed instructions for using each of these tools individually or in combination, also for fuzzing and delta debugging a single MaxSAT Solver, are available in the readme files in GitHub (Paxian 2025b).

5 Research Questions

Our study asks how far we can push MaxSAT fuzzing. We begin by seeing whether we can craft better random instances (RQ1) and whether these stronger instances beat the cases produced by today's generators and benchmark suites (RQ2). Building on that, we fold delta-debugging into the loop and check if it finds failures that straight fuzzing would overlook, while a small, well-chosen regression set keeps solvers steady over time (RQ3–RQ4). For the first time, we also fuzz anytime solvers—a challenging step because we must set strict timeouts and judge them by approximate intermediate answers rather than exact optima (RQ5). The results Section 6 shows that each of these goals is met, highlighting the power and relevance of our approach.

RQ1: Improving Quality of Randomly Generated Instances
- *Criterion Definition:* Develop criteria to improve the quality of randomly generated instances. The goal is to create instances that are harder to solve, thus exploring more interesting paths in the code that might otherwise remain untested. For that we try to:
  - Minimize the number of instances with an unsatisfiable core of hard clauses.
  - Minimize the number of instances in which all soft clauses are satisfiable.
  - Generate instances with Boolean Multilevel Optimization problems.
- *Instance Complexity:* Evaluate whether it is better to generate larger, more complex instances or to run multiple smaller instances within the same time frame. We expect to trigger more failures with smaller instances.

RQ2: Comparison with Existing Tools
- *Comparison of Failure Detection:* Evaluate the performance and effectiveness of previous MaxSAT instance generation tools and fuzzers compared to our approach.
  - Measure the variety of failures found within a given timeout.
  - Analyze the generated instances in terms of the number of soft/hard clauses, clause lengths, number of unsatisfiable hard clauses, and instances in which all soft clauses are satisfiable.

RQ3: Effect of Delta Debugging in Finding New Failures
- *Enhanced Failure Detection:* Investigate how delta debugging assists in discovering new failures.
- *Benchmark Testing:* Analyze the potential of testing benchmarks during the reduction phase of `WCNFddmin`, which might not be generated by the `WCNFuzz` instance generation tool. For instance, the presence of significant variable gaps (i.e., some variables in the variable list do not occur in any clauses) is rare in `WCNFuzz` generated instances but these gaps can be generated in our delta debugger.
- *Fault Discovery:* Examine additional uncovered failures on different instance generation tools. The expectation is that the more failures an instance generation tool can find, less additional failures are found by the delta debugger.

RQ4: Impact of Regression Suite on Solver Robustness
- *Solver Robustness Improvement:* Measure the improvement in solver robustness following the introduction of the Regression Suite for the MSE 2024.
- *Framework Effectiveness:* Evaluate the effectiveness of our MaxSAT fuzzing framework, introduced on the homepage shortly before the solver submission deadline, in enhancing robustness.

RQ5: Finding Failures in Anytime and Certified Solvers
- *Detection of Failures:* Detecting failures in anytime solvers is particularly challenging because these solvers do not terminate by declaring they have found the optimum.
- *Detection of Interesting Failures:* Assess the potential to find failures beyond invalid exit codes, such as incorrect models or incorrect output, in anytime solvers.

Table 1. Properties of MaxSAT instances generated by different fuzzers on testing the MSE23 solver from the Exact Weighted track. The columns *Hard Clauses* and *Soft Clauses* indicate the percentage of instances containing at least one hard or soft clause, respectively. *Unweighted* refers to the percentage of instances where the highest clause weight is 1. *32-bit Weights* refers to the percentage of sum of the positive integer weights smaller than 2^{32}. The columns labeled with (min–max, avg) provide statistics on the minimum, maximum, and average numbers of hard clauses, soft clauses, and variables present in the instances.

Fuzzer	Hard Clauses	Soft Clauses	Unweighted	32-bit Weights	Hard Clauses (min–max, avg)	Soft Clauses (min–max, avg)	Variables (min–max, avg)
Paxian	90.10%	97.88%	19.47%	81.11%	0–832, 159	0–665, 117	4–195, 50
PaxianPy	87.46%	99.23%	17.68%	79.18%	0–564, 89	0–431, 99	3–134, 34
PaxianPySmall	77.92%	97.49%	17.50%	81.48%	0–550, 41	0–314, 55	1–96, 18
PaxianPyTiny	66.63%	96.11%	17.14%	82.07%	0–434, 29	0–230, 40	1–87, 14
Pollitt	98.59%	98.62%	0.01%	100.00%	0–1601, 247	0–1718, 247	1–123981, 10561
Manthey	93.79%	100.00%	2.42%	96.21%	0–1079, 737	2–1999, 826	1–500, 187
Soos	100.00%	100.00%	100.00%	100.00%	2–142237, 4804	2–40338, 942	2–40554, 1137

6 Results

Our primary goal in this study is *not* to rank individual solvers but to measure how much stress our fuzzers can generate and how each design tweak raises—or lowers—the resulting "stress per CPU-hour." Every fuzzer was given the same 100 h wall-clock budget; solver binaries were invoked only to decide whether a freshly generated instance actually triggered a fault. Hence, the *number and type* of solver failures should be read as a proxy for fuzzer power rather than as a definitive audit of solver quality.

The remainder of this section mirrors that objective. We first show how the four grammar-aware Paxian generators (Paxian, PaxianPy, PaxianPySmall, PaxianPyTiny) differ in speed and in the structural mix of their instances (Section 6.1). We then show reduction rates (Section 6.2) and quantify the extra faults exposed when those raw instances are *shrunk* by our delta debugger (Section 6.3). This shrinking phase effectively acts as a mutation-based, grammar-aware fuzzer in its own right, because it rewrites the original formula at progressively finer granularities. With this groundwork laid, we turn to solver-level failure counts on four MaxSAT Evaluation tracks (MSE22–24 Exact Weighted and MSE23 Anytime), highlight the sharp drop in fault density after the Regression Suite became mandatory, and conclude with a certified-solver case study that uncovers a still-verified yet sub-optimal proof.

All experiments were run on a dual-socket AMD EPYC 7343 platform (32 physical cores, 64 hardware threads, 2 TiB RAM, Ubuntu 22.04.5). We parallelised the fuzzer across 62 threads; upon failure detection, our delta debugger used at most half of those cores to minimise the instances. All reported timings are aggregate CPU time—i.e. the sum of per-thread runtimes, equivalent to a single-core execution.

6.1 Quality of Generated Instances

This subsection answers **RQ1 (Instance Quality)** and **RQ2 (Tool Comparison)**. We first describe the fuzzers and their instance characteristics, then compare their failure-finding power.

Fuzzers under test. To keep table captions short we name each fuzzer by the surname of its main author: Paxian, PaxianPy, PaxianPySmall, PaxianPyTiny, Manthey, Pollitt, and Soos.

The four Paxian variants differ in instance size (large to tiny) and programming language (C vs Python). Manthey builds structured formulas with high variety. Pollitt produces layered but relatively simple instances.

Table 2. the 10 **MSE22 solvers** from the **Exact Weighted** track. Each row shows the total number of failures found, number of generated instances, average solving time per instance (across 10 solvers), timeout rate (worst solver), number of solvers with a timeout on more than 0.01 % of all instances, percentage of instances with objective value 0, and percentage of instances with satisfiable hard clauses.

Fuzzer	Total Faults	Executions	Average Time (10 Solvers)	Max Timeouts	Timeouts > 0.01	Objective 0	Satisfiable Hard Clauses
Paxian	45	260,876	1.38	**1.35 %**	4	15.01 %	98.39 %
PaxianPy	46	153,087	2.36	**7.79 %**	4	4.03 %	95.59 %
PaxianPySmall	42	101,188	3.57	**14.53 %**	2	9.33 %	97.81 %
PaxianPyTiny	40	74,294	4.86	**20.98 %**	2	13.19 %	98.55 %
Pollitt	21	420,425	0.86	**1.03 %**	1	85.60 %	96.88 %
Manthey	42	129,483	2.79	**2.05 %**	9	4.64 %	25.90 %
Soos	4	3,824	94.37	64.35 %	10	2.33 %	27.98 %

Table 3. Results of fuzzing the 11 **MSE23 solvers** from the **Exact Weighted track**. Each row shows the total number of **faults** found, number of generated instances, average solving time per instance (across 11 solvers), timeout rate (worst solver), number of solvers with a timeout on more than 0.01 % of all instances, percentage of instances with objective value 0, and percentage of instances with satisfiable hard clauses.

Fuzzer	Total Faults	Executions	Average Time (11 Solvers)	Max Timeouts	Timeouts > 0.01	Objective 0	Satisfiable Hard Clauses
Paxian	53	47,371	7.61	4.77 %	4	14.94 %	98.37 %
PaxianPy	57	56,747	6.36	5.60 %	4	4.01 %	95.47 %
PaxianPySmall	59	129,595	2.78	0.44 %	3	9.23 %	97.77 %
PaxianPyTiny	59	191,660	1.88	0.10 %	3	13.05 %	98.52 %
Pollitt	30	291,389	1.24	0.09 %	1	85.66 %	96.87 %
Manthey	36	50,471	7.15	4.55 %	11	4.55 %	25.52 %
Soos	13	3,091	116.67	60.02 %	11	2.60 %	25.61 %

Table 4. Results of fuzzing the 15 **MSE24 solvers** from the **Exact Weighted track**. Each row shows the total number of **faults** found, number of generated instances, average solving time per instance (across 15 solvers), timeout rate (worst solver), number of solvers with a timeout on more than 0.01 % of all instances, percentage of instances with objective value 0, and percentage of instances with satisfiable hard clauses.

Fuzzer	Total Faults	Executions	Average Time (15 Solvers)	Max Timeouts	Timeouts > 0.01	Objective 0	Satisfiable Hard Clauses
Paxian	17	58,973	6.12	2.34 %	6	15.00 %	98.32 %
PaxianPy	31	86,290	4.18	1.23 %	6	4.07 %	95.40 %
PaxianPySmall	27	162,824	2.22	0.15 %	3	9.15 %	97.87 %
PaxianPyTiny	27	209,564	1.72	0.11 %	3	13.17 %	98.52 %
Pollitt	4	314,877	1.15	0.00 %	0	85.68 %	96.94 %
Manthey	22	55,529	6.50	4.43 %	15	4.48 %	25.64 %
Soos	8	2,897	124.48	64.55 %	15	2.08 %	28.39 %

Soos generates CNF formulas with to CNF translated XOR gates which are hard to solve unweighted instances aimed at GaussMaxHS.

Table 5. Results of fuzzing the 7 **MSE23 solvers** from the **Anytime track**. Each row shows the total number of **faults** found, number of generated instances, average solving time per instance (across 7 solvers), timeout rate (worst solver), number of solvers with a timeout on more than 0.01 % of all instances, percentage of instances with objective value 0, and percentage with satisfiable hard clauses.

Fuzzer	Total Faults	Executions	Average Time (7 Solvers)	Max Timeouts	Timeouts > 0.01	Objective 0	Satisfiable Hard Clauses
Paxian	28	184,866	1.95	93.97 %	7	15.03 %	98.42 %
PaxianPy	29	192,403	1.87	98.10 %	7	3.88 %	95.49 %
PaxianPySmall	30	192,690	1.87	93.94 %	7	9.27 %	97.82 %
PaxianPyTiny	31	191,795	1.88	90.41 %	7	13.00 %	98.50 %
Pollitt	27	363,761	0.99	34.86 %	7	85.56 %	96.91 %
Manthey	23	212,133	1.70	99.80 %	7	4.54 %	25.68 %
Soos	15	136,042	2.65	99.41 %	7	2.29 %	37.64 %

Table 6. Results of fuzzing the **MSE23 solvers** from the **Exact Weighted track**. Each cell gives the number of failures found by the fuzzer in the row that were *not* found by the fuzzer in the column, where failures are distinguished by the classification explained in Section 4.1.

Fuzzer	Total Failures	DeltaDebugger	PaxianPySmall	PaxianPyTiny	PaxianPy	Paxian	Manthey	Pollitt	Soos
VirtualBest	72	1	13	13	15	19	36	42	59
DeltaDebugger	71	–	12	12	15	18	35	41	58
PaxianPySmall	59	0	–	2	5	7	23	29	50
PaxianPyTiny	59	0	2	–	5	6	23	29	50
PaxianPy	57	1	3	3	–	7	22	27	49
Paxian	53	0	1	0	3	–	21	24	44
Manthey	36	0	0	0	1	4	–	11	28
Pollitt	30	0	0	0	0	1	5	–	22
Soos	13	0	4	4	5	4	5	5	–

Instance generation speed. The original C implementation of Paxian creates one instance in 0.005 s, whereas the Python port needs 0.016 s across 1000 generated instances.

Average single solver execution on C-generated instances Paxian lies between 0.28 s (Anytime) and 0.69 s (MSE23 Exact), while Python-generated ones range 0.27–0.58 s for the PaxianPy variant. Because of this timings, generation time is negligible: even the slower Python variant can prepare hundreds of inputs while two solvers (minimum to compare) are still running.

Instance structure. Table 1 shows the average formula size on the MSE23 Exact track. When we shrink the generator from Paxian to PaxianPyTiny:

- hard clauses: 159 → 29,
- soft clauses: 117 → 40,
- variables: 50 → 14.

We keep a healthy mix of clause types and weights while drastically reducing the search space. This matters for higher throughput and later delta debugging, whose run time is linear in formula size.

RQ1 metrics. To evaluate the generators against RQ1 criteria, we measure:

- the percentage of instances with an unsatisfiable core of hard clauses,
- the percentage of instances in which all soft clauses are satisfiable,
- the proportion of unweighted instances and 32-bit number instances.

Tables 2–5 show that across the Paxian variants, the unsatisfiability of the set of hard clauses ranges between 1.5 % and 4.5 %, while the all-soft-satisfiable (Objective 0) rate remains below 15 % for all variants. We improved the all-soft-satisfiable rate in the Python implementation from 15 % down to 4 %, but it increased again to 13 % for PaxianPyTiny, as the number of soft clauses dropped significantly. The proportion of unweighted instances is around 18 %, and the total weight sum remains within the 32-bit integer range in roughly 80 % of the cases as can be seen in Table 1. The proportion of instances with multi-level weights (Boolean multilevel optimization) is not measured explicitly, but based on the generation logic, approximately 13 % of the instances are Boolean multilevel optimization problems.

Solver-level performance. Tables 2–5 summarise 100 h of fuzzing per track.

For MSE23 Exact, smaller instances speed up solving from 7.61s (Paxian) to 1.88s (PaxianPyTiny) and cut the worst per-solver timeout rate to below 0.11 %.

The Paxian family finds 53–59 unique failures per variant, with PaxianPySmall and PaxianPyTiny offering the best trade-off between speed and coverage (throughput $\approx 4\times$ of Paxian) across all tracks and limited time. Due to a higher variation in PaxianPy we expect this fuzzer to be superior with a much higher time limit.

The same pattern repeats on the MSE24 Exact track, but not on MSE22, where MaxHS times out more frequently—see all bold entries in Table 2 for the Paxian variants. Surprisingly and contra intuitively, it times out more often on smaller instances: for PaxianPyTiny, the timeout rate is around 20 %. This significantly reduces both throughput and failure yield.

We traced this behavior to instances containing a contradictory unit soft clause combined with additional soft clauses. For example, the following three unit soft clauses already trigger the issue: `(weight, unit)` $(2, x_1)$, $(1, x_2)$, $(1, \neg x_1)$. These cause MaxHS to enter an apparent infinite loop, repeatedly printing lines like `c solve_w_lsu (coarse) found new UB: 1`.

PaxianPyTiny generates smaller instances with fewer variables, and every fifth instance is designed to contain only unit soft clauses. As a result, this failure-triggering pattern occurs more frequently than in the larger variants, which typically contain more diverse clause layers.

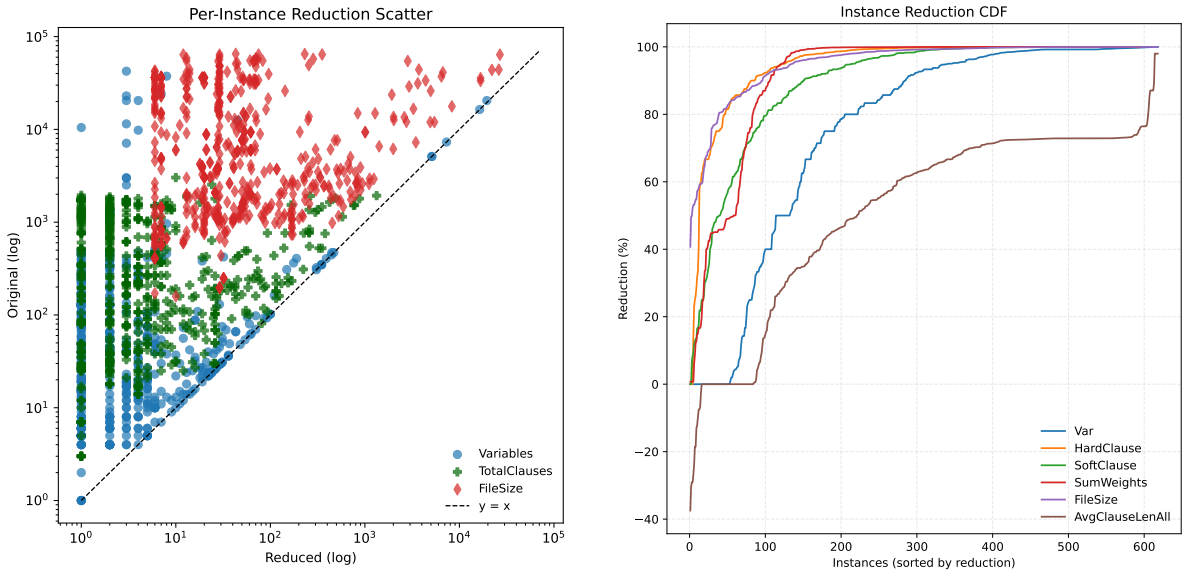
On the MSE23 Anytime track all Paxian variants achieve similar throughput because each instance carries an internal hard timeout (0.01–0.5 s) depending on the original seed. Pollitt produces twice as many runs (as its instances less often hit the time limit), whereas Soos almost always times out.

Tool comparison (RQ2). Figures 7–10 plot the cumulative number of unique solver-fault pairs found over time. PaxianPyTiny and Pollitt lead early, confirming our hypothesis that many small instances expose failures quickly. Overall, our Paxian family outperforms Manthey and Soos on all tracks as more failures are detected.

The fuzzer diversity can be seen in the Table 6 showing the MSE23 Exact results. The best two-fuzzer bundle there is (PaxianPyTiny or PaxianPySmall) + Soos, which adds four extra failure cases.

Take-aways:

- **RQ1:** Smaller, weight-preserving instances (PaxianPyTiny) run $\approx 4\times$ faster yet keep high structural diversity.
- **RQ2:** Our fuzzers uncover more unique solver failures than the existing generators we could find.



Original vs. Reduced Sizes

Reduction Rate Distribution

Fig. 6. Delta debugging effectiveness across 618 minimized instances. Left: scatterplot (log scale) comparing original to reduced sizes, showing that every instance was reduced. The one dot remaining on 10^0 means that we had some instances with only one variable already in the original case. This variable occured in multiple clauses (positive and negative), therefore the total number of clauses cross is not at the same position! Right: cumulative distribution of reduction percentages, highlighting that more than half of the instances shrink by at least 99 %.

Table 7. Reduction statistics for 618 non-recursively delta debugged instances. All values except Count denote percentage reduction. A value of 100% means the corresponding entity (e.g., hard or soft clauses, weights) was completely eliminated. Negative values for average clause length indicate that many small clauses were removed, leaving only larger clauses; this occurred only in 15 cases.

Metric	Min	Avg	Median	Max	Count
Variables	0.00 %	75.62 %	93.28 %	99.99 %	618
Hard Clauses	0.00 %	93.09 %	99.23 %	100.0 %	571
Soft Clauses	0.00 %	89.37 %	98.49 %	100.0 %	613
Total Clauses	0.00 %	90.63 %	98.18 %	99.95 %	618
Sum of Weights	0.75 %	91.30 %	99.95 %	100.0 %	596
File Size	40.71 %	95.31 %	99.25 %	99.99 %	618
Avg. Clause Len.	-37.46 %	51.41 %	63.45 %	97.99 %	618

6.2 Delta Debugging Effectiveness

These 618 minimized WCNF instances were obtained across all fuzzing runs. We focus here on the non-recursive reductions, since only they reflect the full effect of delta debugging: recursive delta debugger runs select the smallest instance in which a different fault still occurs, which makes the reduction appear less dramatic. Using the original seed–fuzzer combinations embedded in the filenames, we were able to regenerate all original instances and measure their reduction rates.

As Figure 6 and Table 7 show, the reductions are striking across all metrics. On average, file size shrinks by 95.31 %, and more than half of the instances are reduced by over 99 %, leaving only a tiny fraction of the original content. At such levels, differences between fuzzers become relatively minor. Still, larger original instances tend to yield higher reduction rates, and instances with very large weights experience greater weight drops.

Special cases are worth noting. A reduction of 100% means the corresponding object (hard clauses, soft clauses, or total weight) was completely eliminated. In 15 instances, the reduction of average clause length is negative: many small clauses were removed, leaving only comparatively large ones. Both effects are clearly visible in the plots of Figure 6.

Fuzzer-specific patterns remain. The Soos fuzzer, which generates some of the largest inputs, achieves the strongest compression: its minimized cases average 99.03 % file size reduction. PaxianPySmall, although starting from smaller inputs, still achieves 90.91 %. In terms of variables, PaxianPy reduces them by only 52.24 % on average, whereas Manthey and Soos each eliminate about 90% (90.21 % and 90.75 %). Clause counts are also cut dramatically, from 78.67 % in PaxianPySmall to 98.06 % in Soos. Overall, reductions are so severe that 54.21% of instances lose at least 90% of their variables, and 54.85% shrink in file size by more than 99%. These results underscore the effectiveness of delta debugging in producing minimal failing instances.

6.3 Effect of Delta Debugging in Finding New Failures

This subsection answers **RQ3**, we first describe our debugger workflow, then report how many failures it reproduces, how many new failures it uncovers, and what novel benchmarks it produces.

Delta debugging workflow. For each solver–fault combination identified during fuzzing, we run `WCNFddmin` to minimize the failing instance. We cap this at five solver-fault reductions with at most two parallel jobs, and always favour smaller instances in reduction. This often yields instances that no fuzzer ever generates during the minimization process. Some minimized instances uncover new faults, while others no longer reproduce the original error. Timeout-triggered failures (20 s per call) are expensive to shrink and may disappear under lighter system load. Non-deterministic solver behavior can also prevent successful minimization.

Fault modes. The debugger cannot reproduce certain failures when:

- the solver exhibits non-deterministic behavior,
- timeouts do not recur under different runtime conditions,
- the failure was only discoverd at the end of our whole run while terminating the delta debugger

Fault reproduction and new discovery. Figures 7–10 plot cumulative failure counts over time, including delta-debugged runs. On the Exact tracks the debugger reproduces almost all original failures; on the Anytime track it triggers (not minimizing) only about 80 % of them. Moreover, the debugger uncovers additional failure—benchmarks that the fuzzers never generate—by shrinking in ways that produce, for example, large variable gaps. This can be seen as a kind of mutation based fuzzing.

RQ3 metrics. Table 8 summarizes, per track, the *fault reproduction rate* (fraction of failures minimized which still trigger the original failure) and the *new-fault rate* (debugger-only failures as a percentage). We double checked the numbers, it is a pure coincidence that the MSE23 ($\frac{8}{72} = 11.11\%$) and MSE24 ($\frac{5}{45} = 11.11\%$) tracks have both the same percentage of new failures.

Performance note. Minimization cost is linear in formula size. A timeout failure may cost up to $2n$ calls at 20 s each. Large formulas (5 720 clauses) can take up to $\approx$ 228 800 s per round, while small ones (59 clauses) need only up to $\approx$ 2 360 s per round in clause reduction.

Table 8. Delta debugging failure-reproduction and new-fault rates and the overall failure detection rate.

Track	Minimized	New failures
MSE22 Exact Weighted	85.25 %	18.03 %
MSE23 Exact Weighted	79.17 %	11.11 %
MSE24 Exact Weighted	80.00 %	11.11 %
MSE23 Anytime	56.10 %	0.00 %

Table 9. Solver–fault matrix for the 10 **MSE22 solvers** from the **Exact Weighted track**. Each cell shows the weight class in which the failure was observed: S = small (<2^{32}, 41 cases), B = big (2^{32}–2^{62}, 12 cases), H = huge (>2^{62}, 8 cases). D (11 cases) marks failures found only through delta debugging, whereas x (9 cases) indicates failures that could not be minimized within the 100 h limit. The solver references: CASHWMaxSAT-CorePlus (Lei et al. 2022), CASHWMaxSAT-Plus (Y. Wang, Pan, Lei, Cai, Yin, et al. 2022), UWrMaxSat variants (Piotrów 2022), MaxHS (Bacchus 2022), WMaxCDCL (Coll, S. Li, C. Li, Manyã, Habet, and He 2022), WMaxCDCL-bandall (Coll, S. Li, C. Li, Manyã, Habet, Zheng, et al. 2022), EvalMaxSAT (Avellaneda et al. 2022), CGSS (Ihalainen et al. 2022) and Exact (Devriendt 2022).

	Crashes				Bound Violations						Perf.		Other	
Solver	1.1	1.3	1.5	1.6	2.1	2.2	2.3	2.4	2.5	2.6	3.1	3.2	4.3	4.4
CASHWMaxSAT-CorePlus	H				S	S	SD				Sx		BD	
CASHWMaxSAT-Plus	H				S	S	SD				SDx		BD	
UWrMaxSat-SCIP	H				S	S	S	S		S	S			
MaxHS				S	S	S	S				S	Sx	S	S
WMaxCDCL			Sx	S	B	B	BD	S	H	S	Hx		S	
WMaxCDCL-bandall		B	S	S	B	B	BD	S	H	S	HDx		S	
UWrMaxSat					S		H				S			
EvalMaxSAT														
CGSS				S	S	B	B							
Exact		BDx	S		S	SDx	SDx				S			

Take-aways:

- **RQ3:** Delta debugging reproduces most original failures (56–85 %) and uncovers up to 18 % new failures.
- Small instances with fewer clauses make debugging practical— as they finish an order of magnitude faster.
- Shrinked benchmarks exhibit novel patterns (e.g. variable gaps, variable separation, unconnected clauses) unseen in fuzz-generated inputs.

6.4 Solver Fuzzing

Exact-Weighted tracks (RQ4). To illustrate the power of fuzzing, consider the tiny instance in Figure 11: a 6-soft, 3-hard clause formula on 6 variables first uncovered errors in every MSE22 solver (CASHWMaxSAT, UWrMaxSat/SCIP, MaxHS/CPLEX, and Z3). Each solver reported objective 2, yet EvalMaxSAT later found a strictly better solution, revealing a soundness failure.

We also observed timeout-induced failures. One 3-soft, 1-hard instance (reduced from 41 soft, 157 hard clauses) forced MaxHS into an infinite loop, only halted by its 1500 s internal timeout. A second solver, Exact, timed out on a similar reduced instance (7 soft, 24 hard clauses), showing that even timeouts can expose logic errors.

Tables 9–11 aggregate all solver–fault pairs by failure class (Section 4.1), weight range (Small: $\leq 2^{32}$, Big: $\geq 2^{32}$, Huge: $\geq 2^{62}$), and minimization status (x: not minimized within timeout; D: delta-debugger-only). Notably, nearly

Table 10. Solver–fault matrix for the 11 **MSE23 solvers** from the **Exact Weighted track**. Each cell shows the weight class in which the failure was observed: S = small (<2^{32}, 61 cases), B = big (2^{32}–2^{62}, 2 cases), H = huge (>2^{62}, 9 cases). D (8 cases) marks failures found only through delta debugging, whereas x (15 cases) indicates failures that could not be minimized within the 100 h limit. The solver references: WMaxCDCL variants (Coll, S. Li, C. Li, Manyã, Habet, Cherif, et al. 2023), EvalMaxSAT variants (Avellaneda 2023), CASHWMAXSAT-CorePlus (Pan, Y. Wang, Lei, et al. 2023), CASHWMAXSAT-CorePlus-m (Y. Wang, Pan, Lei, Cai, X. Wang, et al. 2023), CGSS2 and CGSS2-SCIP (Ihalainen et al. 2023), Pacose and Pacose-MaxPre2 (Paxian and Becker 2023).

	Crashes			Bound Violations						Perf.		Other Issues		
Solver	1.1	1.2	1.5	2.1	2.2	2.3	2.4	2.5	2.6	3.1	3.2	4.1	4.2	4.3
WMaxCDCL-S6-HS12				S	S	S	S		S	Sx		S		Sx
WMaxCDCL-S9-HS9				S	S	S	S		S	Sx		S		Sx
EvalMaxSAT-SCIP			S	S			S			SDx			S	B
CASHWMAXSAT-CorePlus	H			S	S	SD				Sx				
CASHWMAXSAT-CorePlus-m	H			S	S	SD				Sx				
EvalMaxSAT			S	H			S			S			S	
CGSS2-SCIP	S	S	S		B	S			S	Sx	SDx			S
CGSS2	S		S	Sx		S		S	S	Sx	Sx			
WMaxCDCL			S			H		H		Sx				
Pacose				S	S	SD				Sx		S		H
Pacose-MaxPre2	H			S	S	SD		H		Sx		SD		HD

every solver exhibits "2.1" soundness failures in MSE22 and MSE23 Exact: they output a model matching the reported objective and claim it to be optimal even when a strictly better model exists.

All solvers that participated in the MaxSAT Evaluation are assumed to support weights up to $2^{63}-1$ and total soft clause weight sums below 2^{64} (Bacchus, Berg, et al. 2022). Yet some solvers, such as UWrMaxSAT with SCIP, revealed issues only in the "H" weight category, suggesting an implementation-level restriction when handling large weight magnitudes. Although we did not audit individual solvers for true large-weight support, our experimental setup strictly adhered to the MaxSAT Evaluation constraints. Failures marked with "B" or "H" reliably indicate that the corresponding bugs occurred only under such high-weight conditions.

Average failure detection rate (RQ4). In ~800 hours per track we were able to discover:

- 61 failures in 10 MSE22 Exact solvers (6.1 failures/solver),
- 72 failures in 11 MSE23 Exact solvers (6.6 failures/solver),
- 45 failures in 15 MSE24 Exact solvers (3.0 failures/solver).
- 41 failures in 7 MSE23 Anytime solvers (5.9 failures/solver).

The simple "faults / solver" metric is intuitive but biased in two ways. First, we ran each of the seven fuzzers for 100 h, plus roughly 100 h of delta debugging. The time budget is therefore divided among all solvers: the more solvers in a track, the less CPU-time each one receives. The Exact tracks (10, 11, and 15 solvers) thus obtained fewer executions per solver than the Anytime track with 7 solvers. Second, failure discovery is super-linear in time, so normalizing by solver count or wall-clock hours does not equalize each solver's opportunity to fail.

A stricter but much more costly setup would give every solver the same CPU budget (e.g. 100 h each) or report failures per million solver executions. Executing all solvers on an identical instance set would be ideal, yet impractical: delta debugging would then have to compare *all* solvers at every reduction step. In its current state, the fuzzers alone achieved higher throughput in MSE24 (891 k executions / solver) than in MSE23 (770 k), see Tables 2–5. Normalizing by instances would therefore widen the observed drop in faults, not close it. Even

Table 11. Solver–fault matrix for the 15 **MSE24 solvers** from the **Exact Weighted track**. Each cell shows the weight class in which the failure was observed: S = small (<2^{32}, 27 cases), B = big (2^{32}–2^{62}, 0 cases), H = huge (>2^{62} 18 cases). D (5 cases) marks failures found only through delta debugging, whereas x (9 cases) indicates failures that could not be minimized within the 100 h limit. The solver references: CASHWMaxSAT-Disj variants (Pan, Y. Wang, Cai, et al. 2024), UWrMaxSat-SCIP (Piotrów 2024), EvalMaxSAT variants (Avellaneda 2024), wmaxcdcl variants (Avellaneda 2024), cgss variants (Ihalainen et al. 2024), Pacose variants (Paxian and Becker 2024) and Exact (Devriendt 2024).

	Crashes		Bound Violations					Perf.		Other Issues	
Solver	1.1	1.5	2.1	2.2	2.3	2.5	2.6	3.1	3.2	4.1	4.6
CASHWMaxSAT-DisjCom-S6	H				S			Sx			
CASHWMaxSAT-DisjCom-S9	H				S			Sx			
UWrMaxSat-SCIP	H		H		H			H			
EvalMaxSAT			S					Sx			
CASHWMaxSAT-DisjCad-S6	H		HD		S	H		S			
CASHWMaxSAT-DisjCad-S9	H		HD		S	H		S			
EvalMaxSAT-SBVA			S			S		Sx			
EvalMaxSAT-SBVA-saveCores			S			S		S			
wmaxcdcl-openwbo1200								S		H	S
cgss_default								HDx			
cgss_abst_cg			S					HDx			
wmaxcdcl				S	S	H	S	Sx			
Pacose		H						S	HDx		
PacoseMP2								S			
Exact					S			Sx			

Table 12. Solver–fault matrix for **MSE23 solvers** from **Anytime track**. Each cell shows the weight class in which the failure was observed: S = small (<2^{32}, 36 cases), B = big (2^{32}–2^{62}, 0 cases), H = huge (>2^{62}, 5 cases). D (0 cases) marks failures found only through delta debugging, whereas x (18 cases) indicates failures that could not be minimized within the 100 h limit.

	Crashes				Bound Viol.				Other	
Solver	1.1	1.3	1.5	1.6	2.2	2.3	2.5	2.6	4.1	4.3
NuWLS-c_static (Chu et al. 2023)		S	S	S		H			Sx	Sx
tt-open-wbo-inc-IntelSATSolver (Nadel 2023)			S	S		H			Sx	Sx
NuWLS-c-FPS (Zheng et al. 2023)		S	S	S		H			Sx	Sx
NuWLS-c_band (Zheng et al. 2023)		S	S	S		H			Sx	Sx
tt-open-wbo-inc-Glucose4_1 (Nadel 2023)			S	S		H			Sx	Sx
LOANDRA (Berg 2023)	S			Sx	S	S	S	Sx	S	Sx
NoSAT-MaxSAT (Lübke and Schupp 2023)					Sx	Sx		Sx	Sx	Sx

with this bias, the failure rate fell from 6.1–6.6 to 3.0 failures / solver, signaling a real robustness gain in the 2024 MaxSAT Evaluation.

We attribute part of this improvement to the **mandatory regression suite** introduced for the 2024 Evaluation—every solver now has to survive the minimized instances discovered by our fuzzing study. At least two entrants (CGSS and Pacose) also ran our fuzzer during development, further reducing failures. The remaining faults—including crashes, performance regressions, and soundness violations—highlight avenues for future hardening, but the overall down-trend confirms the suite's positive impact.

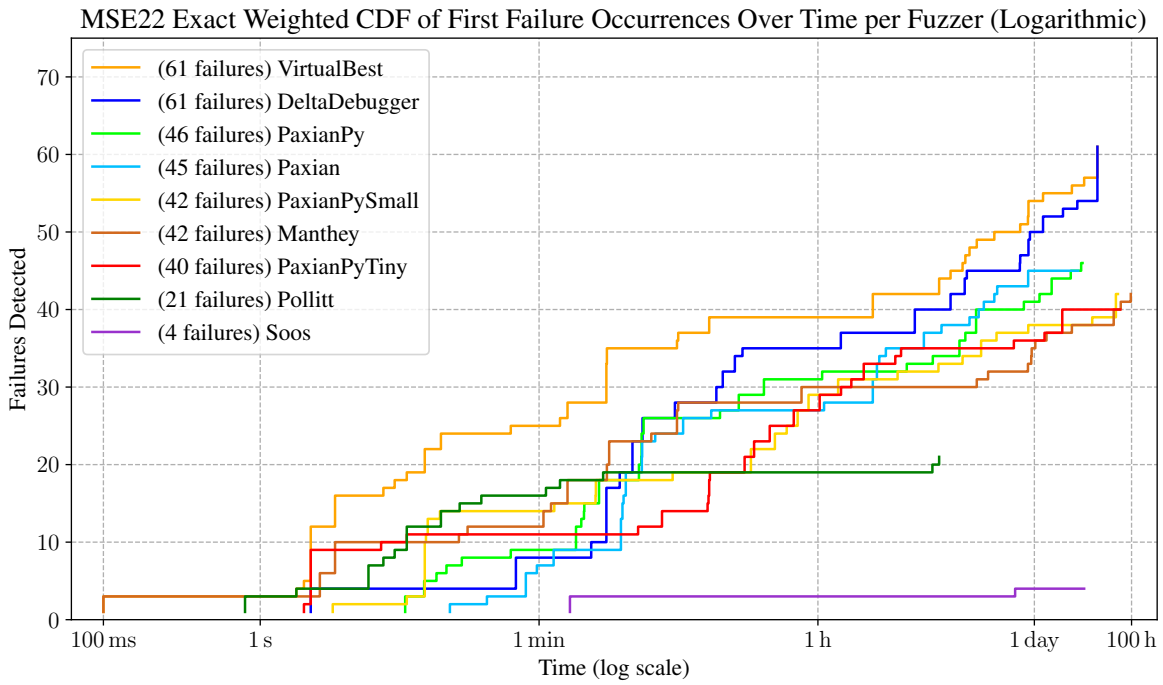


Fig. 7. **CDF of Failure Discovery Times for 10 MSE22 Exact Weighted Solvers.** This plot shows, for each unique solver–fault pair, the cumulative fraction of failures discovered over time by our suite of fuzzers (including the virtual best among them) and by the delta debugger. Each fuzzer was granted a 100 h window to generate instances and test immediately all 10 solvers on these instances, with run-times summed as if executed sequentially on one core. The delta debugger ran in parallel with the fuzzers and was terminated once the 100 h fuzzing window closed.

We did not include our earlier 2022 experiment, which fuzzed only the C implementation (Paxian) over six million generated instances, because it used a different setup and cannot be directly compared. More information to this experiment can be found in our POS article (Paxian and Biere 2023).

Anytime-track fuzzing (RQ5). For the MSE23 Anytime track we send `SIGTERM` after a seed-based timeout of 0.01–0.5 s and reuse the same limit during delta debugging. We ignore "performance-issue'' failures (3.1, 3.2) and bound-violation faults whose reported objective equals the model (2.1) (provided the solver does not claim optimality); otherwise a single `s SATISFIABLE` line is accepted.

Under these rules we uncovered 41 failures in 7 Anytime solvers (5.9 failures / solver), the first fuzzing-derived failures ever reported for this track. All Open-WBO-based solvers (every entrant except `NoSAT-MaxSAT`) crashed or produced bound violations—mostly type 2.3 negative objective values, likely due to integer overflow. In `LOANDRA` and `NoSAT-MaxSAT` we additionally found crashes caused by non-matching models, objective values or incorrect claims that the hard clauses are satisfiable.

Despite the very short timeouts, the delta debugger managed to minimize a surprising number of these failures, although, unlike the Exact tracks, it did not discover *new* failures beyond those already triggered by fuzzers.

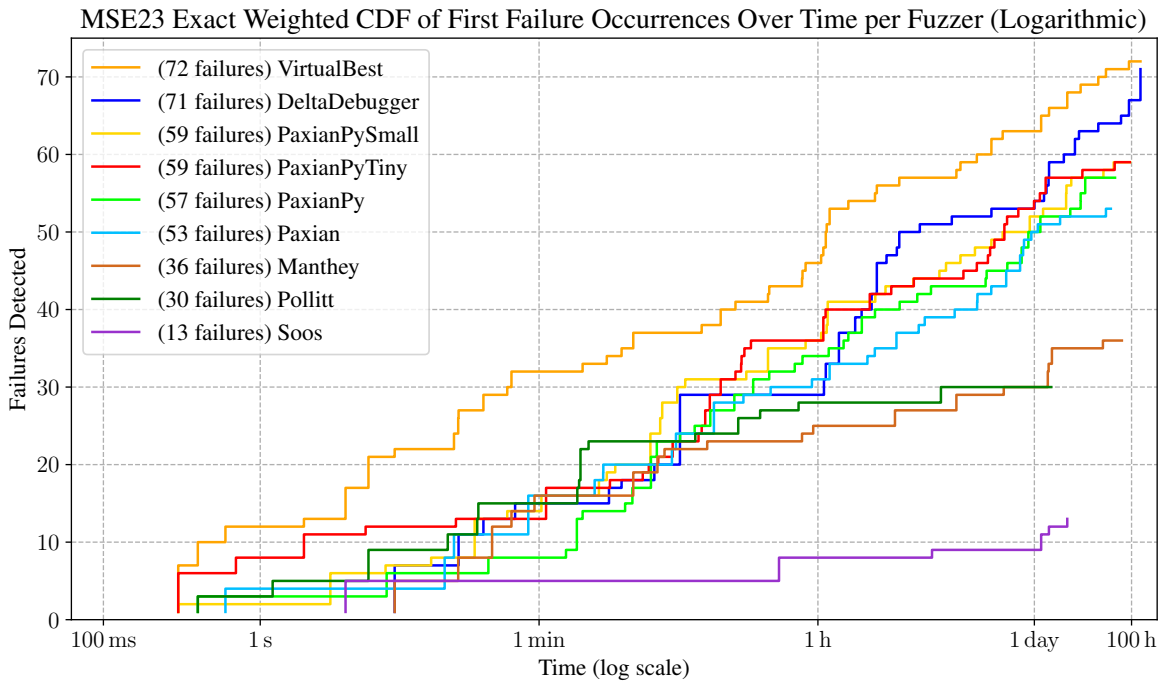


Fig. 8. **CDF of Failure Discovery Times for 11 MSE23 Exact Weighted Solvers.** This plot shows, for each unique solver–fault pair, the cumulative fraction of failures discovered over time by our suite of fuzzers (including the virtual best among them) and by the delta debugger. Each fuzzer was granted a 100 h window to generate instances and test immediately all 11 solvers on these instances, with run-times summed as if executed sequentially on one core. The delta debugger ran in parallel with the fuzzers and was terminated once the 100 h fuzzing window closed.

Certifying-solver fuzzing (RQ5). For these results we fuzzed three certifying solvers—Pacose (Berg, Bogaerts, Nordström, Oertel, Paxian, et al. 2024), CGSS (Berg, Bogaerts, Nordström, Oertel, and Vandesande 2023), and QMaxSATpb (Vandesande et al. 2022). A fourth contender is on the horizon: the branch-and-bound solver MaxCDCL (C. Li et al. 2022). It won the 2023 MaxSAT Evaluation and is currently being extended with proof-production support, which will make it the first state-of-the-art certified branch-and-bound solver.

Because a certified run involves *two* programs (solver → VeriPB), the resulting crashes are harder to bin into the failure classes used for the other tracks; we therefore report a qualitative summary instead of large tables.

Test harness: Our shell script for certifying solvers first calls the solver, then feeds the generated proof to `veripb`. We record the exit codes and scan both outputs for errors or `Verification failed/succeeded`. The pair of exit codes and shell-output errors is then mapped to a unique failure identifier (exit code).

CGSS results. CGSS[2] produced four main failure modes in the certified version:

- CGSS Crashes like: Python `TypeError`,
- VeriPB Crashes with exit codes 4 (not verifiable) and 5 (proof file issue) or segmentation faults
- Soundness failures of types 2.1, 2.2 and 2.3 wrong bound with "successfully verified'' proofs

[2]Git hash `799150ed5d59dae10b5a74831d5cdb06791f4921`, https://gitlab.com/MIAOresearch/software/certified-cgss.git with VeriPB V1 commit `54070be168642d4b14988841d958691c39c61350`, https://gitlab.com/MIAOresearch/software/VeriPB.git

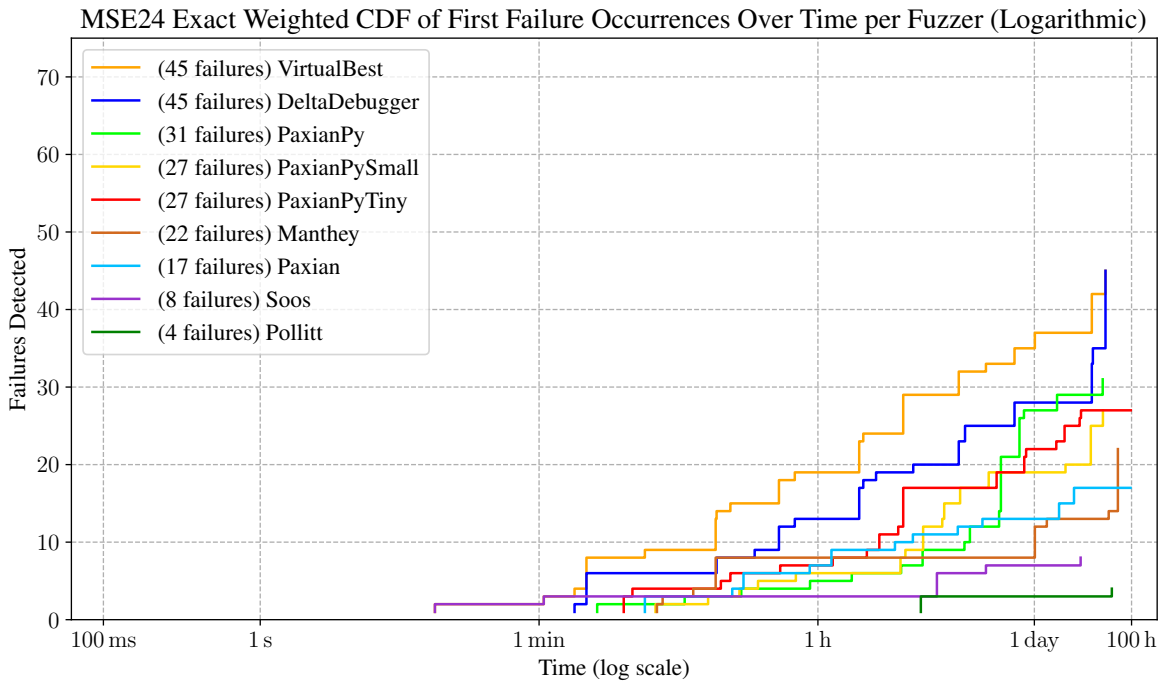


Fig. 9. **CDF of Failure Discovery Times for 15 MSE24 Exact Weighted Solvers.** This plot shows, for each unique solver–fault pair, the cumulative fraction of failures discovered over time by our suite of fuzzers (including the virtual best among them) and by the delta debugger. Each fuzzer was granted a 100 h window to generate instances and test immediately all 15 solvers on these instances, with run-times summed as if executed sequentially on one core. The delta debugger ran in parallel with the fuzzers and was terminated once the 100 h fuzzing window closed.

Most seriously are those of type 2.1, which claim an objective with a matching model and a VeriPB message telling us that the objective is optimal while another solver found a better solution. The latter appears only for large weights but can be reduced—in Figure 12 to five soft clauses with weights $< 2^{55}$.

The failure occurs due to VeriPB version 1 accepting incorrect proofs because CGSS of this version first parses clause weights as 64-bit IEEE 754 floating-point numbers and then converts them to integers.[3] In particular, both 10423761748800001 and 10423761748799999 are rounded to 10423761748800000 in this process. In this particular case the proof's internal model differs slightly from the solver's printed result. Since the solver still logs a correct optimal solution, VeriPB v1 does not verify that this model matches the solver's o and v lines and therefore "verifies" a failurey proof.

The version 2 proof format fixes this by requiring a conclusion line at the end of the proof, in which the solver must restate its o and v results. VeriPB v2 then checks that the solution used in derivations matches that conclusion. However, it still does not enforce that the solver's printed o and v values agree with the conclusion line—this final consistency check must be performed by an external tool.

[3]Personal communication with Andy Oertel, May 19 2025.

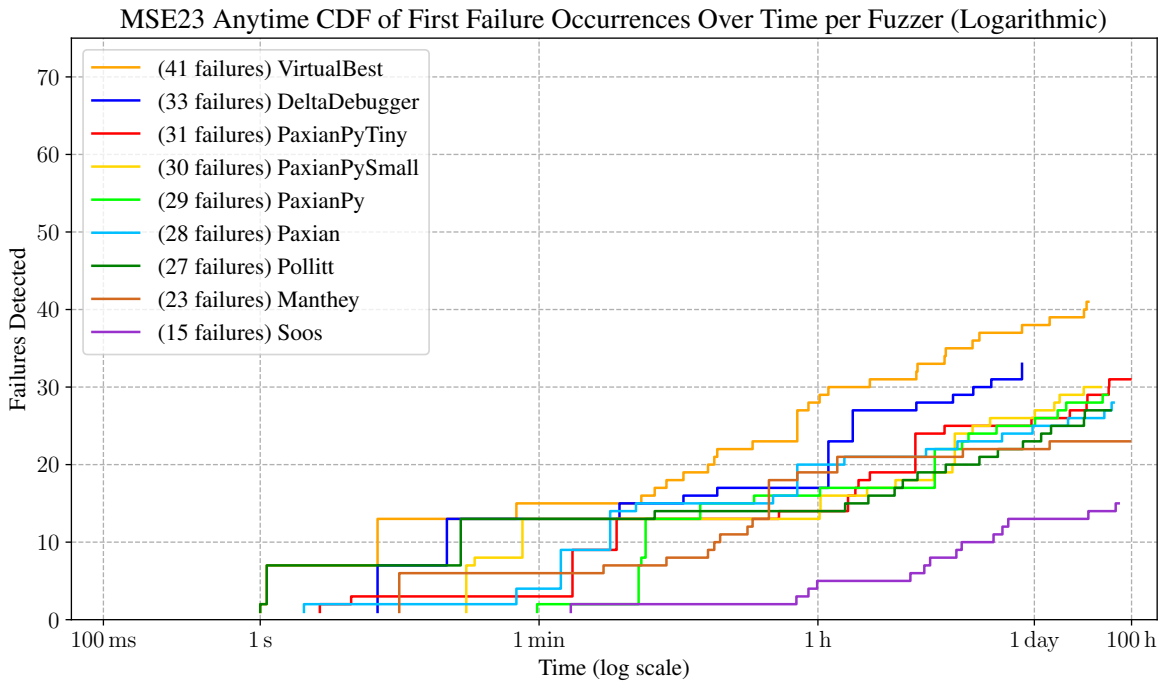


Fig. 10. **CDF of Failure Discovery Times for 7 MSE23 Anytime Solvers.** This plot shows, for each unique solver–fault pair, the cumulative fraction of failures discovered over time by our suite of fuzzers (including the virtual best among them) and by the delta debugger. Each fuzzer was granted a 100 h window to generate instances and test immediately all 7 solvers on these instances, with run-times summed as if executed sequentially on one core. The delta debugger ran in parallel with the fuzzers and was terminated once the 100 h fuzzing window closed.

We fuzzed both versions of CGSS and VeriPB, and in both versions we have CGSS crashes, veripb crashes with exit code 4 (not verifiable) and 5 (proof file issue), but in the new version we do not trigger soundness failures. Both exit codes 4 and 5 occur with both successful and crashing terminations of CGSS.

QMaxSATpb[4] produced frequent crashes both in QMaxSATpb and VeriPB. In QMaxSATpb we had strange exit codes with messages like *proof parsing failed*, segmentation failures, and assertions. Unlike CGSS, we found no type 2.1 soundness errors; all "Verification succeeded'' cases had a consistent objective and model, but sometimes the verification did not succeed. We were even able to produce segmentation faults in VeriPB and crashes with exit codes 4 (not verifiable) and 5 (proof file issue) with successful and crashing terminations of QMaxSATpb.

Pacose[5] is developed with our fuzzer in the loop and therefore remained failure-free in the certified version for all weights below 2^{62} (higher weights cannot be handled by the solver). *The verification successfully succeeded for all generated instances.*

[4]Private repository from author together with VeriPB V1, same commit as for CGSS.
[5]Git commit `6638a48a663b3fd359b193617d493ae5c16a7fcc`, https://github.com/tobipaxe/PacoseMaxSATSolver.git with VeriPB V2 commit `8f9c917ea14eaaf254bf5a12f7385e80ba3813e3`, https://gitlab.com/MIAOresearch/software/VeriPB.git

```
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: 6
c #Clauses: 10 ...
o 2
s OPTIMUM FOUND
v 100110
```

```
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
```

Fig. 11. This shows a comparison of six solver outputs running the same WCNF instance (blue, on the left), all but the solver of the bottom right (green) are failurey (orange, all others). 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 failure by fuzzing and reduced the instance by our delta debugger. Remarkably, all top four solvers from the Exact track of 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. MaxHS identifies one additional clause and the same model. 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).

```
cat min.wcnf
1 3 0
10423761748800001 -1 0
13684528244578459 -2 0
10423761748799999 1 0
13684528244578459 2 0
```

```
certCgss min.wcnf proof.pbp
...
s OPTIMUM FOUND
o 24108289993378460
v 111
---------------------------------
veripb --wcnf min.wcnf proof.pbp
c Verification succeeded.
```

```
certPacose min.wcnf proof.pbp
...
o 24108289993378458
s OPTIMUM FOUND
v 011
---------------------------------
veripb --wcnf min.wcnf proof.pbp
c Verification succeeded.
```

Fig. 12. Minimized certified-solver failure found by delta debugging a Manthey-fuzzer generated other failure. **blue, on the left):** The 5-clause WCNF has total weight in between 2^{54} and 2^{55}. Certification with CGSS **(orange, in the middle)** and Pacose **(green, on the right)** both report `s OPTIMUM FOUND` and pass VeriPB, yet disagree on the objective value (24, 108, 289, 993, 378, 460 vs. 24, 108, 289, 993, 378, 458) whereas both model are correct. This class-2.1 soundness failure is undetectable by checking model-objective equality alone, and exposes a flaw in the old VeriPB1 verifier (patched with the new certified CGSS version working with VeriPB2, but still not verifiable).

Overall, these experiments show that certified solvers are *not* immune to standard fuzzing: proof generation, proof checking, and their interaction form a rich failure surface. The minimal counter-example in Figure 12 underscores the need for independent proof verification and continuous fuzzing of both components.

Take-aways:

- **RQ5:** Our study revealed the first recorded failures in Anytime and Certified tracks—crashes, bound violations, and proof-validation inconsistencies.
- **RQ4:** Failure density in Exact-Weighted solvers fell from 6.1–6.6 to 3.0 failures / solver between MSE22/23 and MSE24, signaling real robustness gains.
  Our mandatory regression suite (MSE24) and early adoption of our fuzzer by some authors (e.g. CGSS, Pacose) appear to have driven these improvements.

7 Discussion

Our experiments confirm several key insights for MaxSAT fuzzing and solver robustness:

Instance-size vs. failure discovery. Small, weight-preserving instances (PaxianPyTiny) deliver up to 4× the failure-finding throughput of large instances without losing failure variety. This supports RQ1 and suggests a mixed-size strategy—many small tests for breadth, occasional large tests for deep structural coverage.

Value of delta debugging (RQ3). Delta debugging reproduces nearly all original failures and uncovers up to 18 % additional failures by generating novel benchmarks (e.g. variable gaps). However, timeouts remain costly to shrink and non-deterministic solver behavior can block minimization. Future work should explore adaptive timeouts and repeat-run heuristics to improve reproducibility.

Extending fuzzing to Anytime and Certified solvers (RQ5). We present the first fuzzing results on Anytime and Certified tracks, revealing bound-violation and proof-validation errors across modern implementations. These findings show that fuzzing benefits any solver paradigm that reports intermediate or certified results.

Rise of portfolio solvers. An increasing number of MaxSAT systems now combine several strategies—e.g. CGSS calls sometimes a solution-improving algorithm, and several solvers give the first seconds to the MIP engine SCIP before reverting to their own code. Such hybrids complicate fuzzing: if the fixed SCIP slot exceeds our per-instance timeout, only SCIP is tested. Developers can still run our fuzzer on each component in isolation, gaining fine-grained feedback and raising confidence in individual strategies. When and whether to hand control to a sub-solver remains an open question (Alòs et al. 2023).

Regression Suite and Competition Impact. To prevent regressions, we released a public regression suite with at least one minimized instance per solver–fault class. Since 2024, passing this suite is mandatory for all Exact-Weighted solvers in the MaxSAT Evaluation. Our study shows this policy already improved robustness by blocking previously discovered failures.

Recommendations and New Evaluation Rules (RQ4). In our workshop paper we proposed new solver rules, and we are pleased that many have been adopted in the Evaluation. In particular, the failure rate in the Exact tracks fell from over 6 (MSE22/23) to 3 (MSE24) after making the regression suite mandatory.

8 Conclusion

We combined a family of grammar-aware generators, an on-the-fly failure checker, and a MaxSAT-specific delta debugger into a single 100-hour pipeline; solver binaries merely confirmed whether an input misbehaved.

The experiment showed that first, tiny weighted instances quadruple failure throughput while keeping diversity. Second, delta debugging is more than a convenience: its weight-cut and soft→ hard passes revealed up to 18 % *new* bugs that generation alone missed. Third, no single generator dominates; a fast mini-generator plus one big generator found the broadest mix of failures.

Although solver quality was only a by-product, the pipeline exposed 219 unique failures across three MaxSAT Evaluation editions—including the first reported issues in anytime and certified solvers. After the resulting regression suite became mandatory in 2024, the average fault density of exact (non approximate) weighted solvers dropped by more than half.

All tools and minimized benchmarks are open-source and already used by solver authors and evaluation organizers. Fuzzing of MIP solvers is underway: support for the MPS format has been integrated into the fuzzer and initial runs are in progress. Extending our approaches to other solver families—such as PB and SMT—together with feedback-guided generation and adaptive shrinking, represents a promising path forward. The core lesson is simple: **invest in smarter fuzzers and the whole solver ecosystem becomes more trustworthy.**

Data & Code Availability.

All generators, comparison scripts, delta-debugging tools, the full regression suite, and the raw logs from every run are archived on Zenodo (Paxian 2025a). A live development version is maintained on GitHub (Paxian 2025b,c).

References

J. Alòs, C. Ansótegui, J. M. Salvia, and E. Torres. 2023. "Exploiting Configurations of MaxSAT Solvers." In: *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada* (LIPIcs). Ed. by R. H. C. Yap. Vol. 280. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:16. doi:10.4230/LIPICS.CP.2023.7.

C. Ansótegui, M. L. Bonet, and J. Levy. 2013. "SAT-based MaxSAT algorithms." *Artif. Intell.*, 196, 77–105. doi:10.1016/J.ARTINT.2013.01.002.

V. Atlidakis, P. Godefroid, and M. Polishchuk. 2019. "RESTler: stateful REST API fuzzing." In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by J. M. Atlee, T. Bultan, and J. Whittle. IEEE, 748–758. doi:10.1109/ICSE.2019.00083.

F. Avellaneda. 2023. "EvalMaxSAT 2023." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 12–13.

F. Avellaneda. 2024. "EvalMaxSAT 2024." In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2024-2. Ed. by J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. University of Helsinki, Helsinki, Finland, 8.

F. Avellaneda, C.-E. Bilodeau-Savaria, and L. Normand. 2022. "Weighted Version of EvalMaxSAT 2022." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 12.

F. Bacchus. 2022. "MaxHS in the 2022 MaxSat Evaluation." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 17–18.

F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. (Niskanen. 2022. *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Tech. rep. Department of Computer Science, University of Helsinki, Helsinki. http://hdl.handle.net/10138/347396.

F. Bacchus, M. Järvisalo, and R. Martins. 2021. "Maximum Satisfiability." In: *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications. Vol. 336. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. IOS Press, 929–991. doi:10.3233/FAIA201008.

J. Berg. 2023. "Loandra in the 2022 (and 2023) MaxSAT Evaluation." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 21–22.

J. Berg, B. Bogaerts, J. Nordström, A. Oertel, T. Paxian, and D. Vandesande. 2024. "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability." In: *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain* (LIPIcs). Ed. by P. Shaw. Vol. 307. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:28. doi:10.4230/LIPICS.CP.2024.4.

J. Berg, B. Bogaerts, J. Nordström, A. Oertel, and D. Vandesande. 2023. "Certified Core-Guided MaxSAT Solving." In: *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings* (Lecture Notes in Computer Science). Ed. by B. Pientka and C. Tinelli. Vol. 14132. Springer, 1–22. doi:10.1007/978-3-031-38499-8_1.

J. Berg, A. Hyttinen, and M. Järvisalo. 2018. "Applications of MaxSAT in Data Analysis." EPiC Series in Computing 59, 50–64. Ed. by D. L. Berre and M. Järvisalo. doi:10.29007/3QKH.

J. Berg and M. Järvisalo. 2017. "Cost-optimal constrained correlation clustering via weighted partial Maximum Satisfiability." *Artif. Intell.*, 244, 110–142. doi:10.1016/J.ARTINT.2015.07.001.

R. Bhattacharjee, M. A. Sufian, F. Tasnim, S. A. Sara, and S. Hossain. 2024. "Enhancement Of Mutation-Based Fuzzing Methods." In: *Proceedings of the 3rd International Conference on Computing Advancements, ICCA 2024, Dhaka, Bangladesh, October 17-18, 2024*. ACM, 200–206. doi:10.1145/3723178.3723205.

R. Brummayer and A. Biere. 2009. "Fuzzing and delta-debugging SMT solvers." In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, 1–5.

R. Brummayer and M. Järvisalo. 2010. "Testing and debugging techniques for answer set solver development." *Theory Pract. Log. Program.*, 10, 4-6, 741–758. doi:10.1017/S1471068410000396.

R. Brummayer, F. Lonsing, and A. Biere. 2010. "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* (Lecture Notes in Computer Science). Ed. by O. Strichman and S. Szeider. Vol. 6175. Springer, 44–57. doi:10.1007/978-3-642-14186-7_6.

P. C. Cheeseman, B. Kanefsky, and W. M. Taylor. 1991. "Where the Really Hard Problems Are." In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*. Ed. by J. Mylopoulos and R. Reiter. Morgan Kaufmann, 331–340. http://ijcai.org/Proceedings/91-1/Papers/052.pdf.

Y. Chen, S. Safarpour, J. Marques-Silva, and A. G. Veneris. 2010. "Automated Design Debugging With Maximum Satisfiability." *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 29, 11, 1804–1817. doi:10.1109/TCAD.2010.2061270.

Y. Chu, S. Cai, and C. Luo. 2023. "NuWLS-c-2023: Solver Description." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 23–24.

J. Coll, S. Li, C. Li, F. Manyà, D. Habet, M. S. Cherif, and K. He. 2023. "WMaxCDCL in MaxSAT Evaluation 2023." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 16–17.

J. Coll, S. Li, C. Li, F. Manyà, D. Habet, and K. He. 2022. "MaxCDCL and WMaxCDCL in MaxSAT Evaluation 2022." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 15–16.

J. Coll, S. Li, C. Li, F. Manyà, D. Habet, J. Zheng, and K. He. 2022. "WMaxCDCL-BandAll in MaxSAT Evaluation 2022." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 23.

J. Devriendt. 2022. "Exact: Evaluating a Pseudo-Boolean Solver on MaxSAT Problems." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 13.

J. Devriendt. 2024. "Exact: Evaluating a Pseudo-Boolean Solver on MaxSAT Problems (2024)." In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2024-2. Ed. by J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. University of Helsinki, Helsinki, Finland, 11–12.

B. B. Duarte, R. de Almeida Falbo, G. Guizzardi, R. S. S. Guizzardi, and V. E. S. Souza. 2018. "Towards an Ontology of Software Defects, Errors and Failures." In: *Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings* (Lecture Notes in Computer Science). Ed. by J. Trujillo, K. C. Davis, X. Du, Z. Li, T. W. Ling, G. Li, and M. Lee. Vol. 11157. Springer, 349–362. doi:10.1007/978-3-030-00847-5_25.

M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller. 2023. "Semantic Debugging." In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. Ed. by S. Chandra, K. Blincoe, and P. Tonella. ACM, 438–449. doi:10.1145/3611643.3616296.

M. Fleury, J. C. Blanchette, and P. Lammich. 2018. "A verified SAT solver with watched literals using imperative HOL." In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. Ed. by J. Andronick and A. P. Felty. ACM, 158–171. doi:10.1145/3167080.

M. Ghasemisharif. 2018. "State of the Fuzz : An Analysis of Black-Box Vulnerability Testing." *Technical Report, University of Illinois*, 1–15. https://api.semanticscholar.org/CorpusID:84835030.

B. Ghosh and K. S. Meel. 2020. "IMLI: An Incremental Framework for MaxSAT-Based Learning of Interpretable Classification Rules." *CoRR*, abs/2001.01891. http://arxiv.org/abs/2001.01891 arXiv: 2001.01891.

S. Gocht, R. Martins, J. Nordström, and A. Oertel. 2022. "Certified CNF Translations for Pseudo-Boolean Solving." In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel* (LIPIcs). Ed. by K. S. Meel and O. Strichman. Vol. 236. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:25. doi:10.4230/LIPICS.SAT.2022.16.

P. Godefroid, A. Kiezun, and M. Y. Levin. 2008. "Grammar-based whitebox fuzzing," 206–215. Ed. by R. Gupta and S. P. Amarasinghe. doi:10.1145/1375581.1375607.

P. Godefroid, M. Y. Levin, and D. A. Molnar. 2012. "SAGE: whitebox fuzzing for security testing." *Commun. ACM*, 55, 3, 40–44. doi:10.1145/2093548.2093564.

M. R. Golla and S. Godboley. 2024. "Automated SC-MCC test case generation using coverage-guided fuzzing." *Softw. Qual. J.*, 32, 3, 849–880. doi:10.1007/S11219-024-09667-3.

A. Gupta, R. Gopinath, and A. Zeller. 2022. "CLIFuzzer: mining grammars for command-line invocations." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. Ed. by A. Roychoudhury, C. Cadar, and M. Kim. ACM, 1667–1671. doi:10.1145/3540250.3558918.

H. Ihalainen, J. Berg, and M. Järvisalo. 2022. "CGSS in the 2022 MaxSAT Evaluation." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 10–11.

H. Ihalainen, J. Berg, and M. Järvisalo. 2024. "CGSS2 and CGSS2-AbstCG in the 2024 MaxSAT Evaluation." In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2024-2. Ed. by J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. University of Helsinki, Helsinki, Finland, 13–14.

H. Ihalainen, J. Berg, and M. Järvisalo. 2023. "CGSS2 in the 2023 MaxSAT Evaluation." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 10–11.

U. Junker. 2004. "QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems." In: *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*. Ed. by D. L. McGuinness and G. Ferguson. AAAI Press / The MIT Press, 167–172. http://www.aaai.org/Library/AAAI/2004/aaai04-027.php.

D. Kaufmann and A. Biere. 2022. "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* (Lecture Notes in Computer Science). Ed. by L. Kovács and K. Meinke. Vol. 13361. Springer, 69–88. doi:10.1007/978-3-031-09827-7_5.

M. Krawiec. 2018. "Terminological discrepancies in the field of software testing: A case of mistake, error, bug, defect, fault, and failure in the specialist language of IT." *Studia Linguistica*, 37, 71–81.

Z. Lei, Y. Wang, S. Pan, S. Cai, and M. Yin. 2022. "CASHWMaxSAT-CorePlus: Solver Description." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 8.

C. Lemieux and K. Sen. 2018. "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by M. Huchard, C. Kästner, and G. Fraser. ACM, 475–485. doi:10.1145/3238147.3238176.

C. M. Li and F. Manyà. 2021. "MaxSAT, Hard and Soft Constraints." In: *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications. Vol. 336. Ed. by A. Biere, M. Heule, H. van Maaren, and T. Walsh. IOS Press, 903–927. doi:10.3233/FAIA201007.

C. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He. 2022. "Boosting branch-and-bound MaxSAT solvers with clause learning." *AI Commun.*, 35, 2, 131–151. doi:10.3233/AIC-210178.

H. Liu, H. Guo, P. Liu, and T. Hou. 2025. "BoostPolyGlot: A Structured IR Generation-Based Fuzz Testing Framework for GCC Compiler Frontend." *Applied Sciences*, 15, 11, 5935. https://www.mdpi.com/2076-3417/15/11/5935.

O. Lübke and S. Schupp. 2023. "noSAT-MaxSATv2." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 27–28.

R. Mahmood, J. Pennington, D. Tsang, T. Tran, and A. Bogle. 2022. "A Framework for Automated API Fuzzing at Enterprise Scale." In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 377–388. doi:10.1109/ICST53961.2022.00018.

M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang. 2020. "Detecting critical bugs in SMT solvers using blackbox mutational fuzzing." In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by P. Devanbu, M. B. Cohen, and T. Zimmermann. ACM, 701–712. doi:10.1145/3368089.3409763.

N. Manthey. 2020. *MaxSAT Fuzzer*. (2020). https://github.com/conp-solutions/maxsat-fuzzer.

B. P. Miller, L. Fredriksen, and B. So. 1990. "An Empirical Study of the Reliability of UNIX Utilities." *Commun. ACM*, 33, 12, 32–44. doi:10.1145/96267.96279.

C. Miller and Z. N. J. Peterson. 2007. "Analysis of mutation and generation-based fuzzing." *Independent Security Evaluators, Tech. Rep*, 4. https://api.semanticscholar.org/CorpusID:199412673.

G. Misherghi and Z. Su. 2006. "HDD: hierarchical Delta Debugging." In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. Ed. by L. J. Osterweil, H. D. Rombach, and M. L. Soffa. ACM, 142–151. doi:10.1145/1134285.1134307.

D. G. Mitchell, B. Selman, and H. J. Levesque. 1992. "Hard and Easy Distributions of SAT Problems." In: *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*. Ed. by W. R. Swartout. AAAI Press / The MIT Press, 459–465. http://www.aaai.org/Library/AAAI/1992/aaai92-071.php.

M. Mushthofa, S. Schockaert, and M. D. Cock. 2016. "Computing attractors of multi-valued Gene Regulatory Networks using Fuzzy Answer Set Programming." In: *2016 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2016, Vancouver, BC, Canada, July 24-29, 2016*. IEEE, 1955–1962. doi:10.1109/FUZZ-IEEE.2016.7737931.

A. Nadel. 2023. "TT-Open-WBO-Inc-23: an Anytime MaxSAT Solver Entering MSE'23." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 29–30.

A. Niemetz, M. Preiner, and C. W. Barrett. 2022. "Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers." In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by S. Shoham and Y. Vizel. Vol. 13372. Springer, 92–106. doi:10.1007/978-3-031-13188-2_5.

K. Njeru, P. Mwangi, and J. Omondi. 2017. "Automatic Debugging Approaches: A literature Review." *ResearchGate*, 1–20.

E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. 2004. "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 (Lecture Notes in Computer Science). Ed. by M. Wallace. Vol. 3258. Springer, 438–452. doi:10.1007/978-3-540-30201-8_33.

S. Pan, Y. Wang, S. Cai, J. Li, W. Zhu, and M. Yin. 2024. "CASHWMaxSAT-DisjCad: Solver Description." In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2024-2. Ed. by J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. University of Helsinki, Helsinki, Finland, 25.

S. Pan, Y. Wang, Z. Lei, S. Cai, S. Wang, and M. Yin. 2023. "CASHWMaxSAT-CorePlus: Solver Description." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 8.

T. Paxian. June 2025a. *MaxSAT Fuzzing and Delta Debugging Paper Attachment*. Zenodo, (June 2025). doi:10.5281/zenodo.15773637.

[SW] T. Paxian, *MaxSAT-Fuzzer* version commit 5e0e365, 2025. URL: https://github.com/tobipaxe/MaxSAT-Fuzzer.

[SW] T. Paxian, *MaxSAT-RegressionSuite* version commit 46fbe36, 2025. URL: https://github.com/tobipaxe/MaxSATRegressionSuite.

T. Paxian and B. Becker. 2023. "Pacose: An Iterative SAT-based MaxSAT Solver." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 20.

T. Paxian and B. Becker. 2024. "Pacose: An Iterative SAT-based MaxSAT Solver." In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2024-2. Ed. by J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. University of Helsinki, Helsinki, Finland, 26.

T. Paxian and A. Biere. 2023. "Uncovering and Classifying Bugs in MaxSAT Solvers through Fuzzing and Delta Debugging." In: *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023* (CEUR Workshop Proceedings). Ed. by M. Järvisalo and D. L. Berre. Vol. 3545. CEUR-WS.org, 59–71. https://ceur-ws.org/Vol-3545/paper5.pdf.

T. Paxian, P. Raiola, and B. Becker. 2021. "On Preprocessing for Weighted MaxSAT." In: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings* (Lecture Notes in Computer Science). Ed. by F. Henglein, S. Shoham, and Y. Vizel. Vol. 12597. Springer, 556–577. doi:10.1007/978-3-030-67067-2_25.

J. A. N. Pérez and A. Voronkov. 2005. "Generation of Hard Non-Clausal Random Satisfiability Problems." In: *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. Ed. by M. M. Veloso and S. Kambhampati. AAAI Press / The MIT Press, 436–442. http://www.aaai.org/Library/AAAI/2005/aaai05-069.php.

A. Pferscher and B. K. Aichernig. 2022. "Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning." Lecture Notes in Computer Science 13260, 373–392. Ed. by J. V. Deshmukh, K. Havelund, and I. Perez. doi:10.1007/978-3-031-06773-0_20.

M. Piotrów. 2022. "UWrMaxSat Entering the MaxSAT Evaluation 2022." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 21–22.

M. Piotrów. 2024. "UWrMaxSat Entering the MaxSAT Evaluation 2024." In: *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2024-2. Ed. by J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. University of Helsinki, Helsinki, Finland, 27.

P. Raiola, T. Paxian, and B. Becker. 2020. "Minimal Witnesses for Security Weaknesses in Reconfigurable Scan Networks." In: *IEEE European Test Symposium, ETS 2020, Tallinn, Estonia, May 25-29, 2020*. IEEE, 1–6. doi:10.1109/ETS48528.2020.9131566.

J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. 2012. "Test-case reduction for C compiler bugs," 335–346. Ed. by J. Vitek, H. Lin, and F. Tip. doi:10.1145/2254064.2254104.

P. Rodler. 2022. "A formal proof and simple explanation of the QuickXplain algorithm." *Artif. Intell. Rev.*, 55, 8, 6185–6206. doi:10.1007/S10462-022-10149-W.

T. Seufert, F. Winterer, C. Scholl, K. Scheibler, T. Paxian, and B. Becker. 2023. "Everything You Always Wanted to Know About Generalization of Proof Obligations in PDR." *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 42, 4, 1351–1364. doi:10.1109/TCAD.2022.3198260.

H. Sochor, F. Ferrarotti, and R. Wille. 2024. "GrammarForge: Learning Program Input Grammars for Fuzz Testing." In: *Software Engineering and Formal Methods - 22nd International Conference, SEFM 2024, Aveiro, Portugal, November 6-8, 2024, Proceedings* (Lecture Notes in Computer Science). Ed. by A. Madeira and A. Knapp. Vol. 15280. Springer, 272–289. doi:10.1007/978-3-031-77382-2_16.

M. Soos and K. S. Meel. 2021a. "Gaussian Elimination Meets Maximum Satisfiability." In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*. Ed. by M. Bienvenu, G. Lakemeyer, and E. Erdem, 581–587. doi:10.24963/KR.2021/55.

M. Soos and K. S. Meel. 2021b. *GaussMaxHS GitHub Repository*. Ed. by M. Bienvenu, G. Lakemeyer, and E. Erdem. Available at https://github.com/meelgroup/gaussmaxhs. (2021). doi:10.24963/KR.2021/55.

P. Srivastava and M. Payer. 2021. "Gramatron: effective grammar-aware fuzzing." In: *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*. Ed. by C. Cadar and X. Zhang. ACM, 244–256. doi:10.1145/3460319.3464814.

O. A. Tazl, C. Tafeit, F. Wotawa, and A. Felfernig. 2022. "DDMin versus QuickXplain - An Experimental Comparison of two Algorithms for Minimizing Collections." In: *The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022*. Ed. by R. Peng, C. E. Pantoja, and P. Kamthan. KSI Research Inc., 481–486. doi:10.18293/SEKE2022-172.

D. Vandesande, W. D. Wulf, and B. Bogaerts. 2022. "QMaxSATpb: A Certified MaxSAT Solver." In: *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings* (Lecture Notes in Computer Science). Ed. by G. Gottlob, D. Inclezan, and M. Maratea. Vol. 13416. Springer, 429–442. doi:10.1007/978-3-031-15707-3_33.

M. Vasylenko. July 2024. *Input Invariants in Fuzz-testing*. (July 2024). http://essay.utwente.nl/101049/.

D. Vince, R. Hodován, D. Bársony, and Á. Kiss. 2021. "Extending Hierarchical Delta Debugging with Hoisting." In: *2nd IEEE/ACM International Conference on Automation of Software Test, ASTICSE 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 60–69. doi:10.1109/AST52587.2021.00015.

D. Vince, R. Hodován, D. Bársony, and Á. Kiss. 2022. "The effect of hoisting on variants of Hierarchical Delta Debugging." *J. Softw. Evol. Process.*, 34, 11. doi:10.1002/SMR.2483.

G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang. 2021. "Probabilistic Delta debugging." In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. Ed. by D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta. ACM, 881–892. doi:10.1145/3468264.3468625.

J. Wang, B. Chen, L. Wei, and Y. Liu. 2018. "Superion: Grammar-Aware Greybox Fuzzing." *CoRR*, abs/1812.01197. http://arxiv.org/abs/1812.01197 arXiv: 1812.01197.

Y. Wang, S. Pan, Z. Lei, S. Cai, X. Wang, and M. Yin. 2023. "CASHWMaxSAT-CorePlus-m: Solver Description." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 9.

Y. Wang, S. Pan, Z. Lei, S. Cai, M. Yin, S. Hu, and Y. Zhou. 2022. "CASHWMaxSAT-Plus: Solver Description." In: *MaxSAT Evaluation 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2022-2. Ed. by F. Bacchus, J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 9.

Z. Yu, Z. Liu, X. Cong, X. Li, and L. Yin. 2024. "Fuzzing: Progress, Challenges, and Perspectives." *Computers, Materials and Continua*, 78, 1, 1–29. doi:https://doi.org/10.32604/cmc.2023.042361.

A. Zeller. 2006. *Why programs fail - a guide to systematic debugging*. Elsevier. ISBN: 978-1-55860-866-5.

A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. 2024. *The Fuzzing Book*. Retrieved 2024-07-01. CISPA Helmholtz Center for Information Security. Retrieved July 1, 2024 from https://www.fuzzingbook.org/.

A. Zeller and R. Hildebrandt. 2002. "Simplifying and Isolating Failure-Inducing Input." *IEEE Trans. Software Eng.*, 28, 2, 183–200. doi:10.1109/32.988498.

A. Zeller and R. Hildebrandt. 2025. "Simplifying and Isolating Failure-Inducing Input: A Retrospective on Delta Debugging." *IEEE Transactions on Software Engineering*.

L. Zhang and F. Bacchus. 2012. "MAXSAT Heuristics for Cost Optimal Planning." In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. Ed. by J. Hoffmann and B. Selman. AAAI Press, 1846–1852. doi:10.1609/AAAI.V26I1.8373.

M. Zhang. 2025. "Program Reduction: Versatility, Insights, and Efficacy." Ph.D. Dissertation. University of Waterloo.

M. Zhang, Z. Xu, Y. Tian, X. Cheng, and C. Sun. May 2025. " Toward a Better Understanding of Probabilistic Delta Debugging." In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, (May 2025), 2024–2035. doi:10.1109/ICSE55347.2025.00117.

Y. Zhang, N. Zhong, W. You, Y. Zou, K. Jian, J. Xu, J. Sun, B. Liu, and W. Huo. 2022. "NDFuzz: a non-intrusive coverage-guided fuzzing framework for virtualized network devices." *Cybersecur.*, 5, 1, 21. doi:10.1186/S42400-022-00120-1.

Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei. 2023. "Fuzzing Configurations of Program Options." *ACM Trans. Softw. Eng. Methodol.*, 32, 2, 53:1–53:21. doi:10.1145/3580597.

X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G. Wang. 2024. "A systematic review of fuzzing." *Soft Comput.*, 28, 6, 5493–5522. doi:10.1007/S00500-023-09306-2.

Y. Zhao, L. Gao, Q. Wan, and L. Zhao. 2024. "Grammar-aware test case trimming for efficient hybrid fuzzing." *J. King Saud Univ. Comput. Inf. Sci.*, 36, 1, 101920. doi:10.1016/J.JKSUCI.2024.101920.

J. Zheng, K. He, M. Jin, Z. Chen, and J. Xue. 2023. "Combining BandMaxSAT and FPS with NuWLS-c." In: *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Vol. B-2023-2. Ed. by J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. University of Helsinki, Helsinki, Finland, 25–26.

X. Zhu, S. Wen, S. Camtepe, and Y. Xiang. 2022. "Fuzzing: A Survey for Roadmap." *ACM Comput. Surv.*, 54, 11s, 230:1–230:36. doi:10.1145/3512345.

Received 30 June 2025; accepted 20 October 2025