Methods for Evaluating the Similarity of Fuzzers for Model Counters

Cristian Soare[1]

Supervisor: Dr. Anna L. D. Latour[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 27, 2025

Name of the student: Cristian Soare
Final project course: CSE3000 Research Project
Thesis committee: Dr. Anna L. D. Latour, Dr. Martin Skrodzki

Abstract

Model Counting solvers are critical in many domains. One way of validating them is through fuzzing. However, current fuzzing approaches lack systematic methods to evaluate how different test generators compare in bug-triggering behavior. This paper proposes three methods for evaluating fuzzer similarity: black-box analysis, differential testing, and white-box analysis.

A similarity metric was defined using differential testing, incorporating solver behavior, solve time, and count differences. A case study was conducted on three fuzzers across numerous configurations and four SOTA model counters, followed by an analysis of correlations between CNF feature distribution similarities and fuzzer behavioral similarities.

The results show moderate correlations between graph structure features (minimum variable node degrees) and fuzzer similarity, while clause balance features show negative correlations. However, the method proves inconclusive for selecting dissimilar fuzzers due to limited incorrect counts in our dataset, uncertainty about crash causes, and the similarity metric's inherent subjectivity. Further research is needed, either by expanding the scope of the experiment with more diverse fuzzers, CNF features, and counters, or by pursuing another method recommended by this study.

1 Introduction

Model Counting or **#SAT** is the counting version of the **Boolean Satisfiability (SAT)** problem and determines the number of solutions to a Boolean formula in **Conjunctive Normal Form (CNF)**. A solution is defined as a set of assignments of truth values to variables such that the formula evaluates to true. In recent years, many important areas of research have relied more and more on the problem of Model Counting, including areas such as quantitative software verification [Teuber and Weigl, 2021, Girol *et al.*, 2021], network reliability [Kabir and Meel, 2023], cryptography [Beck *et al.*, 2020], bayesian networks and probabilistic inference [Darwiche, 2004], and even real-world applications such as reliability of power grids [Latour *et al.*, 2022]. Numerous such algorithmic problems can be effectively modeled to count the number of solutions to a Boolean formula. Researchers have been more focused on proving that their respective problems can be transformed into #SAT rather than building the necessary algorithms [Gomes *et al.*, 2021]. Therefore, they require available third-party model counters (#SAT solvers).

Using third-party solvers comes with one important issue: they must have a high degree of correctness and efficiency. While it is reasonable to expect model counters to accurately indicate the model count and do this predictably in terms of time, this expectation is not always met. Although new #SAT solvers are built employing testing and validation methods, they also suffer from bugs that are difficult to spot: integer overflow, precision errors, memory issues, and other non-deterministic behavior [Brummayer *et al.*, 2010]. For this reason, testing and debugging tools are essential for developers building and working with model counters, as they ensure reliability and robustness [Brummayer *et al.*, 2010].

Brummayer et al. [2010] argue that **fuzzing** can be used to test SAT solvers and to catch the bugs discussed previously. This approach can be naturally extended to #SAT solvers as well. Fuzzing works by generating random inputs for the program under test and then running the program with said inputs to check if the program crashes or behaves in an unexpected manner [Zeller *et al.*, 2024]. In the case of #SAT, the output generated by the fuzzers represents a collection of Boolean formulas in CNF form, and an a set of undesired behaviors resulting from running solvers with this input (often characterized by either a crash or an incorrect model count).

Researchers have created tools designed to generate test cases based on the aforementioned principle. However, there is a critical gap in our ability to evaluate them. Specifically, we lack systematic methods to compare their performance across various solvers and to assess the similarities and differences in the test cases they generate. Subsequently, it is difficult for practitioners to choose the most effective fuzzers to determine the correctness of a specific solver. This paper addresses this research gap through the following contributions:

1. Three distinct methods for evaluating fuzzer similarity in #SAT solvers are introduced, drawing upon methodologies from existing literature. One method is then selected and evaluated through a case study using the SHARPVELVET [Latour and Soos, 2024] project and a collection of instance generators and solvers.
2. Using the selected method:
   (a) Based on experimental results, CNF features that most effectively characterize fuzzer similarity are explored and analyzed.
   (b) The method's overall effectiveness in analyzing fuzzer similarity via bug detection patterns is assessed.

The core research question becomes:

What methods can be employed to evaluate and characterize the similarity between different fuzzers for model counting solvers to ensure comprehensive testing coverage?

This overarching question is divided into several specific research questions:

- **RQ1**: What are the key characteristics and limitations of different methodological approaches for evaluating fuzzer similarity in Model Counting solvers?
- **RQ2**: Which CNF features most effectively characterize fuzzer similarity when using our approach?
- **RQ3**: How effective is our selected method for evaluating fuzzer similarity in Model Counting solvers, and what advantages does it offer compared to alternative approaches?

This paper will proceed with Section 2, which provides more technical definitions for SAT and #SAT. Following that, Section 3 explores related work in the field, offering context and emphasizing the contributions of prior research. Section 4 outlines the proposed methods for evaluating the similarity of fuzzers, while Section 5 details the methodology and experimental setup for the selected case study. Section 6 addresses the transparency, accessibility and scientific integrity part of the study. Section 7 presents the results, limitations, and suggestions for future work. Finally, Section 8 summarizes the contributions of this research.

2 Preliminaries

This section contains definitions on Boolean Satisfiability and Model Counting, and important details about SHARPVELVET

project.

2.1 Boolean Satisfiability

Boolean Satisfiability, or SAT, is the problem of determining whether a given Boolean formula F (usually in Conjunctive Normal Form) has at least one **satisfying assignment**. A formula F in CNF form is defined over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, and is represented as a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable x_i or its negation $\neg x_i$. An assignment $\alpha : X \to \{0, 1\}$ is called a satisfying assignment if α makes F evaluate to True.

2.2 Model Counting

Model Counting, or #SAT, extends the SAT problem by computing the total number of satisfying assignments of a Boolean formula F. Let $\mathrm{Sol}(F)$ denote the set of all satisfying assignments of F. The model counting problem seeks to compute $|\mathrm{Sol}(F)|$, the **cardinality** of $\mathrm{Sol}(F)$.

In Figure 1, an example problem instance in CNF form and a comparison between the result of the SAT problem and the #SAT problem is shown. Note that the assignment shown for the SAT problem is just one of many correct assignments.

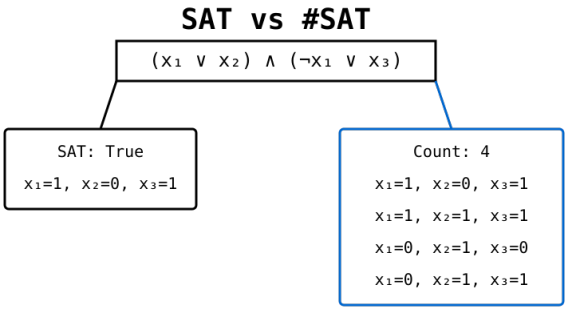

Figure 1: An illustration of a boolean formula in CNF form and the difference between SAT and #SAT.

2.3 The SharpVelvet Project

The SharpVelvet project[1] contains the following separate components:

- **Instance Generation**: The ability to generate CNF instances in the DIMACS format using multiple generators [Latour and Soos, 2024].
- **Fuzz Testing**: Scripts for executing multiple solvers on the generated instances, reporting discrepancies in their outputs and identifying crashes or incorrect results [Latour and Soos, 2024].
- **Verification Tools**: Integration with verifiers like *CPOG* [Bryant *et al.*, 2023] for verifying model counts.

SharpVelvet requires the use of model counters based on the 2024 Model Counting Competition specification[2]. This specification mandates that model counters must accept CNF formulas in DIMACS format and output their results in a standardized manner, in order to be later parsed by SharpVelvet and compiled into a report.

[1] https://github.com/meelgroup/SharpVelvet

[2] https://mccompetition.org/assets/files/mccomp_format_24.pdf

3 Related Work

This section showcases three important areas relevant to this study: general approaches for fuzzer evaluation, fuzzing techniques developed for SAT and #SAT solvers, and methods of CNF instance analysis. First, the literature on general fuzzer evaluation is examined, exploring details about experimental design. Afterward, fuzzing techniques related to SAT and #SAT solvers are addressed, focusing on black-box fuzzing and the problems of determining the ground truth. The section concludes by discussing methods for characterizing CNF instances, including the established SATZilla feature set and contemporary structural metrics that can be applied to fuzzing analysis.

3.1 Evaluating Fuzzers

Given that this study focuses on evaluating fuzzers, it is important to first look at existing general methods for fuzzer evaluation. Klees et al. [2018] present guidelines for analyzing new fuzzer innovations, proposing a framework to determine whether a new fuzzer advances the state-of-the-art. They emphasize the importance of comparing a proposed fuzzer against strong baselines, ensuring experiments account for the inherent randomness of fuzzing, and repeating tests sufficiently (at least 30 trials) to enable statistical validation [Klees *et al.*, 2018]. More recently, Schloegel et al. [2024] expand on these principles by critically reviewing 150 fuzzing papers published between 2018 and 2023. Their analysis revealed widespread deviations from established evaluation guidelines, such as incomplete documentation of setups and insufficient statistical validation. Both studies highlight the need to report experimental setups, including benchmarks, runtime configurations, and seed selection, to support reproducibility and reliable conclusions.

Böhme et al. (2022) challenge the reliability of using code coverage to measure fuzzer effectiveness. The study does indicate a strong correlation between code coverage and the number of bugs found. However, only a moderate correlation between coverage and fuzzer ranking is found. This suggests that while higher code coverage may indicate a greater potential for uncovering bugs, it does not definitively determine the overall effectiveness of a fuzzer.

There are two big differences between the fuzzers described by Klees et al. [2018] and Schloegel et al. [2024] and #SAT fuzzers. Firstly, #SAT fuzzers generate CNF inputs that adhere to a specific format, rather than producing completely random strings. This characteristic complicates the assessment of whether the test set used in the experimental evaluation is sufficiently diverse. Furthermore, the emphasis is not on validating whether a particular fuzzer should be classified as state-of-the-art. Instead, comparing the similarity of fuzzers in terms of the bugs they trigger is more relevant. By doing so, it is possible to achieve higher code coverage on solvers by selecting a collection of dissimilar fuzzers. Nonetheless, it remains essential to apply the findings of Klees et al. [2018] and Schloegel et al. [2024] to enhance the rigor of fuzzer output comparisons, ensuring that the diversity and structural complexity of the generated CNF instances are properly quantified.

3.2 Fuzzing Techniques for SAT and #SAT Solvers

As mentioned in Section 1, Brummayer et al. [2010] is the cornerstone study that defines the fuzzing SAT and #SAT solvers field. Not only is it the first time fuzzing is proposed as a solution for validating different types of SAT solvers, but it also contributes to one of the earliest implementations of a

CNF instance generator. The fuzzing method behind this generator is called *"black-box fuzzing"*. The fuzzers themselves cannot interact with the solver-under-test. Instead, they work by only generating syntactically valid but random CNF inputs, which act as inputs to solvers. Given the nature of this method, an important question is: *How can we determine if the solver-under-test triggered a bug on a specific instance?* This is indeed one important aspect of these fuzzers: **they all require a mechanism for deciding on the "ground truth"**. While crashes are easy to identify, determining whether a model counter has produced an incorrect result presents a more complex challenge.

3.3 The Structure of Instances

After addressing fuzzing techniques and ways of evaluating fuzzers, another relevant part of this study is analyzing CNF features. Nudelman et al. [2004] are the first to describe the *"SATZilla"* set of features, which are a collection of features defined to help measure the "hardness" (computational effort) of SAT solvers. This concept is further developed in Xu et al. [2008], who create a solver portfolio which selects from a collection the best counter for each input instance based on an empirical hardness model trained on a set of SATZilla features. Shavit and Hoos [2024] revisited these features and developed a more efficient feature extractor. This updated version incorporates modern preprocessing techniques to optimize feature computation for more extensive SAT formulas.

The SATZilla set of features has been proven to help predict the computational hardness of SAT solvers. However, an open question remains: how well do these features scale to #SAT solvers? Additionally, do the same features help with fuzzing and bug detection patterns? There are also many other proposed CNF features to consider, such as the treewidth of different graph representations of formulas (primal [Dilkas, 2023], consensus, or conflict [Ganian and Szeider, 2021]), structural entropy, as defined by Zhang et al. [2021], and many others.

4 Method Study

This section presents and evaluates three methods for quantifying the similarity of fuzzers for model counters in terms of bug-triggering behavior. The methods were envisioned by documenting standard practices in the fuzzing literature, and merits and weaknesses were analyzed. This section ultimately answers **RQ1**.

4.1 Black-box Analysis

A black-box oracle evaluates a program solely based on its inputs and outputs, without relying on knowledge of its internal workings [Zeller *et al.*, 2024]. In evaluating the generators themselves, metrics such as crashes triggered and incorrect results can be used to assess similarities or differences between solvers. By analyzing crash timings, we can determine if two fuzzers trigger the same bug, assuming the timing measurements remain consistent and reliable across different executions. For incorrect counts, the analysis becomes more intricate. The number of incorrect model counts per solver or the difference between incorrect and correct ones can help identify which bugs are triggered by which fuzzers.

However, this process is not always precise. It often relies on the assumption that similar incorrect counts indicate the same bug, which may not always hold true [Klees *et al.*, 2018, Böhme *et al.*, 2022]. Improving result metrics can involve gathering extensive samples and applying statistical or machine-learning methods to identify patterns. For example, clustering the count differences can reveal bugs with similar behavior. Therefore, this method of evaluating fuzzers only based on solver output requires significant time and effort to produce a comprehensive analysis.

4.2 Differential Testing Approach

Differential testing evaluates program behavior by comparing solver outputs across multiple implementations [Zeller *et al.*, 2024]. In the context of fuzzer similarity analysis, this approach involves running instances from different fuzzers through multiple solvers and comparing their behavioral patterns. Like black-box analysis, this method operates solely on solver outputs without requiring internal knowledge of the solvers or fuzzers.

The core idea is to characterize the similarity between two fuzzers by analyzing how pairs of their generated instances behave across multiple solvers. A similarity score can be computed for each pair of instances from different fuzzers based on whether they trigger similar solver behaviors (timeouts, correct count, incorrect count, crashes, or other outcomes). These pairwise similarities can then be aggregated to quantify overall fuzzer similarity. CNF features of the instances can be analyzed to understand which characteristics contribute to similar behavior patterns.

This method is beneficial when traditional black-box analysis is impractical - for example, when bugs producing incorrect counts are rare or when precise crash timing analysis is infeasible due to resource sharing or other environmental constraints. However, the approach has significant limitations. The similarity metric is inherently subjective and based on biased assumptions, making it difficult to establish whether two fuzzers truly pinpoint the same solver weaknesses.

4.3 White-box Analysis

Unlike analyzing the outputs of different fuzzers in isolation, white-box analysis examines program behavior through symbolic execution and program code coverage analysis techniques [Godefroid *et al.*, 2008b]. By exploring execution paths and tracking constraints [Godefroid *et al.*, 2008a], more meaningful insights about test coverage and bug detection capabilities can be gained compared to black-box approaches [Godefroid, 2012].

A method for comparing fuzzers can be implemented by using AFL[3] [Zalewski, 2016] or AFL++ [Fioraldi *et al.*, 2020]. The instances produced by existing instance generators like FuzzSAT [Brummayer *et al.*, 2010] can be fed through an AFL-instrumented #SAT solver, allowing execution paths and crashes to be tracked through AFL's shared memory maps [Zeller *et al.*, 2024].

The gathered data can be used to analyze how different fuzzers' outputs engage the solver's functionality, focusing on instances that cause crashes or incorrect counts and identifying where in the code these issues occur. Using this method, ensuring that two instances trigger the same bug is trivial. Then, CNF features such as SATZilla [Nudelman *et al.*, 2004] can be used to determine which characteristics of the fuzzing input best determine if two generators will trigger the same set of bugs.

While this approach has many advantages compared to the previous two, it also has drawbacks. The central problem is that instrumenting a solver causes overhead, which may drastically impact solver performance. This means that the analysis cannot consider solver timeouts or memory exhaustion errors as issues. Another issue is that the instrumented

[3] `https://lcamtuf.coredump.cx/afl/`

solver setup must first be implemented. Unforeseen bugs related to this implementation may arise later in the analysis and be challenging to identify and address.

5 Methodology and Experimental Setup

Due to resource and time constraints, only the **differential testing approach** was selected for evaluation. This section focuses on this method and presents a case study using the SHARPVELVET project, a selection of fuzzer configurations, and state-of-the-art #SAT solvers from the 2024 Model Counting Competition[4]. This method was chosen because it allows an initial assessment of its effectiveness before exploring more complex approaches. SHARPVELVET already provides a way of doing differential testing and enables analysis of fuzzer behavior patterns even in cases without crashes or incorrect counts, offering insights into similarities between fuzzers beyond traditional bug-triggering methods.

A summary of the steps taken to conduct this case study:

1. Multiple instance generators were configured with various parameter settings (discussed in Section 5.1 and Section 5.3).
2. A set of features to measure the instances generated was selected (addressed in Section 5.2).
3. A collection of diverse state-of-the-art #SAT solvers was chosen (outlined in Section 5.4).
4. All instances were fuzzed using instances from the selected generators through the selected #SAT solvers.
5. The counter responses for each instance was classified as correct, incorrect, timeout, or crash (further details in Section 5.5).
6. Similarity between instances and generators was computed based on the behavioral patterns of the fuzzed solvers (see Section 5.6 and Section 7).
7. CNF features and their correlation with the similarity computed for each generator was analyzed using statistical tests (discussed in Section 7).

5.1 Generator Configurations

This study uses three CNF instance generators: FuzzSAT, PairSAT[5], and HornSAT[6], a Horn clause modifier. FuzzSAT, developed by Brummayer et al. [2010] and extended by Latour and Soos [2024], generates formulas by constructing boolean circuits as Direct Acyclic Graphs, randomly selecting operators (AND, OR, XOR, IFF) to connect nodes until reaching minimum reference counts, then converting to CNF via Tseitin transformation. PairSAT, developed by Vuk Jurišić during a parallel research project, creates bipartite graphs representing CNF formulas with controlled clause-to-variable ratios and balanced variable distributions. HornSAT, also developed by Vuk Jurišić, takes existing CNF formulas and systematically modifies them to contain varying fraction of Horn clauses (clauses with at most one positive literal).

Each base generator (FuzzSAT and PairSAT) is configured across presumed hardness levels and randomness degrees, though their actual hardness for model counting remains to be empirically validated. For each 100% randomness configuration, we select a random instance to create HornSAT instances. All configurations used and their name can be seen in Table 1.

[4] https://zenodo.org/records/14249109
[5] https://github.com/Chevuu/PairSAT
[6] https://github.com/Chevuu/HornSAT

Generator	Hardness	Variables	Clause Length	Randomness
FuzzSAT	Easy	18-30	1-3	0%: Purely structural 50%: 45-55% random clauses 100%: 90-100% random
	Hard	30-50	4-6	Same as above
PairSAT	Easy	30-50	1-3	0%: Balanced + fixed (4.0) 100%: Unbalanced + variable
	Hard	55-90	4-6	0%: Balanced + fixed (4.5) 100%: Unbalanced + variable
FuzzSATHORN	Easy	FuzzSAT random easy instance varied to 0-100% Horn clauses		
	Hard	FuzzSAT random hard instance, varied to 0-100% Horn clauses		
PairSATHORN	Easy	PairSAT instance with same variations		
	Hard	PairSAT hard instance with same variations		

Table 1: Generator configurations and parameters

The choice of fuzzers and configurations was motivated by the need for a diverse range of instances to evaluate instance similarity. The combination of different generators and randomness parameters enables the analysis of structural patterns across varied formula types, independent of the generators' effectiveness as fuzzers.

5.2 CNF Features

The feature set used in the analysis is based on the foundational SATZilla features defined by Nudelman et al, and later reduced by Xu et al. [2008] into a *"base"* set of features. As mentioned in Section 3, no direct connection has been found between this set of features and the hardness of instances for #SAT. These features were selected because they capture important structural patterns within CNF data, which can substitute CNF properties, and they are also easy and quick to extract [Shavit and Hoos, 2024]. While other novel CNF features were analyzed and considered, they were excluded from this particular experiment due to the lack of available tools for their extraction in SHARPVELVET.

Table 2 describes the feature categories from the original SATzilla paper [Nudelman *et al.*, 2004].

Category	Description and Features
Problem Size	Basic formula statistics: variables and clauses before (nvarsOrig, nclausesOrig) and after preprocessing (nvars, nclauses), their ratio (vars-clauses-ratio), and preprocessing reductions (reducedVars, reducedClauses).
Variable-Clause Graph	Statistics from bipartite graph connecting variables to clauses they appear in. Includes degree distributions for variables (VCG-VAR-mean/min/max/entropy) and clauses (VCG-CLAUSE-mean/min/max/entropy).
Variable Graph	Derived from variable co-occurrence in clauses. Tracks node degree statistics (VG-mean, VG-coeff-variation, VG-min, VG-max).
Clause Graph	Graph where clauses sharing variables are connected. Includes degree statistics (CG-mean/min/max/entropy) and clustering coefficients measuring local density (cluster-coeff-mean/min/max/entropy).
Balance	Distribution of positive/negative literals in clauses (POSNEG-RATIO-CLAUSE-*) and per variable (POSNEG-RATIO-VAR-*). Also includes clause size proportions (UNARY, BINARY+, TRINARY+).
Horn	Measures Horn formula characteristics through horn-clauses-fraction and variable statistics in Horn clauses (HORNY-VAR-mean/min/max/entropy).

Table 2: Description of SATzilla feature categories used in this analysis.

5.3 Instance Reduction Strategy

Initially, 1000 instances per fuzzer configuration were generated, resulting in $10,000$ total CNF instances (+ 404 Horn-SAT instances). The reasoning for using a larger number of instances is that as the quantity of instances generated increases, the impact of randomness on instance distribution per generator decreases. However, due to time and resource limitations, 100 instances from each generator configuration were used in the fuzzing process.

The selection/downsampling process aimed to preserve the distribution of CNF features from the original set. This was achieved through stratified sampling, where the feature space was divided into adaptive bins, and instances were selected proportionally from each bin. Statistical validation confirmed that the reduced dataset maintained the essential characteristics of the original, as proven by three example feature distributions in Figure 2, while reducing computational requirements by 90%.

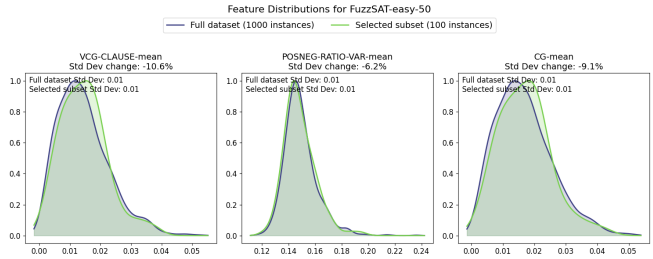

Figure 2: Comparison of three example feature distributions before and after downsampling.

5.4 Selected Solvers

The choice of solvers was limited to those accepting DIMACS format instances from the 2024 Model Counting Competition, focusing on traditional, unweighted model counting. Four state-of-the-art solvers were selected based on their diverse technical approaches and strong performance. This selection was motivated by the need to analyze fuzzing outcomes across different solver implementations while keeping the experimental scope computationally tractable.

- D4 [Lagniez and Marquis, 2017]: A knowledge compilation-based counter using dynamic decomposition with hypergraph partitioning. Achieved strong performance across all sections in the 2024 competition[7].
- GANAK [Sharma *et al.*, 2019]: Builds on sharpSAT's [Thurley, 2006] component caching architecture with probabilistic component caching and improved heuristics. Currently state-of-the-art in unweighted model counting performance.
- ExactMC [Lai *et al.*, 2021]: Introduces CCDD, a Decision-DNNF generalization capturing literal equivalences.
- GPMC [Suzuki *et al.*, 2017]: Extends Glucose 3.0 [Audemard and Simon, 2009] and sharpSAT with preprocessing and tree decomposition-based decision heuristics.

All model counters were run with a timeout of twelve minutes, or 720 seconds. This was chosen because of the limited time and resources available for the study; however, it does balance fuzzing hard instances and the ability to fuzz at least 120 instances in 24 hours.

[7] https://mccompetition.org/assets/files/2024/MC2024_awards.pdf

5.5 Result Classification

Fuzzing results were characterized by distinct classifications, as shown in Table 3. The framework distinguishes between instances with correct counts (including zero for unsatisfiable instances) and incorrect counts, while also tracking solver-specific issues like timeouts and crashes.

Classification	Details
Correct Count	Solution matches expected count (zero for unsatisfiable instances)
Incorrect Count	Solution differs from expected count
Timeout	Solver exceeded time limit
Crash	Solving process terminated unexpectedly

Table 3: Classification of instances in the testing framework

5.6 Similarity Evaluation

Two fuzzers are evaluated for similarity by analyzing their behavior patterns across multiple solvers. For each instance generated by a fuzzer, we classify the solver's response into four categories: correct count, incorrect count, timeout, or crash. We then compute a similarity score between two fuzzers by comparing these behavioral patterns and their quantitative measures across all their instance pairs and solvers.

For two instances i_1, i_2 with count c_1, c_2 and solve time t_1, t_2, the similarity score $s(i_1, i_2)$ is calculated as:

$$S(i_1, i_2) = \begin{cases} 0 & \text{if behaviors differ} \\ 0.5 + 0.5e^{-0.1|t_1 - t_2|} & \text{if both crashed} \\ 0.5 + 0.25e^{-0.01|c_1 - c_2|} + & \text{if both correct/} \\ \quad 0.25e^{-0.1|t_1 - t_2|} & \text{incorrect} \\ P(\text{same behavior}) & \text{if both timed out} \end{cases}$$

The exponential decay function provides bounded outputs between 0 and 1 with diminishing rates of change as differences increase. This matches the intuition that the likelihood of similar algorithmic paths decreases exponentially with growing differences in count or solve time. A baseline similarity of 0.5 is assigned for matching behaviors, with the remaining 0.5 weighted equally between count and time differences. The steeper -0.1 decay rate for time reflects how variations of tens of seconds typically indicate distinct algorithmic paths, while the gentler -0.01 decay for count differences accommodates moderate variations that may still represent similar underlying behavior.

For timeout cases, without having access to the actual count and solve time differences, the exact similarity score cannot be computed. However, rather than assuming that both instances exhibit the same behavior, we compute $P(\text{same behavior})$ as the probability that both instances would exhibit the same behavior (correct or incorrect) based on the sample distributions of their respective generators.

The final similarity is calculated between generators as an aggregation of all pairwise similarity scores calculated previously:

$$S = \lambda \mu_S + (1 - \lambda)(1 - (s_{max} - s_{min}))$$

where μ_S is the mean similarity across all instance pairs and solvers, and $\lambda = 0.7$ balances between mean similarity and consistency. The second term uses range rather than standard deviation as it directly measures maximum variability

in the $[0, 1]$-bounded similarity scores, where higher values indicate more consistent behavior patterns. This formulation rewards fuzzers that maintain reliable algorithmic patterns across all instances rather than just matching on average, while the range's simple interpretation makes the consistency penalty transparent.

5.7 Software Environment

The experiments were conducted using an enhanced version of SHARPVELVET [Latour and Soos, 2024], augmented with some additional features[8]. These tools were developed collaboratively by the Research Project Peer Group. The modular architecture preserves the original SHARPVELVET codebase by Latour and Soos [2024] without direct modifications.

5.8 Hardware Configuration

All experiments were run on the DelftBlue supercomputer [Delft High Performance Computing Centre (DHPC), 2024]. The experiments utilized Phase 1 compute nodes, each equipped with:

- 48-core Intel Xeon processors
- 185 GB RAM
- High-performance interconnect network for parallel computing

6 Responsible Research

Responsible, accessible and ethical research practices are essential in any scientific study. Therefore, this section discusses such considerations, acknowledging some of the challenges regarding fuzzing model-counting solvers and their impact on the broader research community.

6.1 Transparency and Reproducibility

An important aspect of conducting experimental research is ensuring that the resulting data is described transparently and that readers of the study can reproduce the experiment in its totality. Given the nature of fuzzing, which employs randomness throughout the process, including the seed used in the random number generators is essential. Numerous tools (generators, counters) were used and can be configured differently. Knowledge of said configuration is required to ensure another person can use these tools similarly. A companion GitHub repository[8] is therefore available, which includes not only any of the aforementioned configurations but also a guide on how to use this information in order to repeat the same experiment. We also include the Jupyter Notebooks used to aggregate the data and generate the figures shown in Section 7.

Nevertheless, it is important to acknowledge that the nature of the fuzzing results is non-deterministic and that hardware also plays a role in the possible bugs triggered. For example, running the solvers on slower hardware results in more timeouts. Moreover, the operating system and hardware architecture dictate arithmetic logic and memory access behavior. Therefore, even with all the information available, a subsequent identical experiment might show different fuzzing behavior.

[8] https://github.com/Krat-OS/cse3000-how-to-break-a-solver/tree/csoare-cse3000-clean

6.2 Accessibility

Accessibility is an important consideration, as it directly impacts the study's broader utility and reach. A step taken to achieve this is using colorblind-friendly visualization principles. All figures, tables, and plots are either in black-and-white or employ a carefully selected colorblind-safe palette called *"Viridis"*. This way, regardless of color impairment, any reader can visualize the figures.

A notable limitation to the study's accessibility is its computational requirements. While all experiments were conducted using High-Performance Computing resources [Delft High Performance Computing Centre (DHPC), 2024], these facilities may not be available to researchers aiming to replicate our findings, potentially restricting broader validation of our results.

6.3 Ethical Considerations and Scientific Integrity

Running fuzzers on model-counters has limited ethical implications. The nature of both areas of research is theoretical and does not have direct effects on data privacy or real-world ethical applications. Even if model-counting solvers may be used for unethical purposes, this is derivative and unrelated to the subject of this study: evaluating fuzzers and improving the testing of model counters.

Regarding scientific integrity, all software tools used in this paper were utilized under license or with developer permission. They were comprehensively cited, and their licenses are available in their respective repositories and in the main repository for this project.

6.4 Generative AI

This research utilized Large Language Models and generative AI tools to support various aspects of the study. The applications encompassed:

- Research assistance for brainstorming, source identification, and summarization. For instance, the work by Klees et al. [2018] was identified using ChatGPT's web search functionality. All AI-generated summaries were comprehensively verified against original source materials.
- Help with LaTeX formatting. Example prompt used in Claude 3.5 Sonnet: *"help me make this table smaller in size while keeping the font large. maybe we could also shorten the text in the cells a little to facilitate this"*.
- Software development support via GitHub Copilot, powered by Claude Sonnet 3.5, for implementing shell scripts, integrating feature extraction and verification with generator output in Python, and developing the data aggregation and visualization code.
- Grammarly AI provided writing support through grammar checks and sentence improvements.

7 Results, Analysis and Future Work

This section presents our experimental results, analyzes the findings, and discusses limitations and suggestions for future research. The experiment was conducted according to the steps described in section 5.

7.1 Results and Analysis

The analysis was initiated by examining the initial generators and the characteristics of the generated instances. The input data was categorized into the following groups:

- **Feature Category** (Variable Graph Features, Clause Graph Features, etc.)

- **Base Generator** (FuzzSAT, PairSAT, FuzzSATHORN, PairSATHORN)
- **Presumed Hardness** (easy, hard)
- **Randomness Control** (0%, 50%, 100%)

While analyzing the distribution of features generated by fuzzers is not the primary focus of this study, it remains important to understand the data under evaluation. Figure 3 illustrates the distribution of Clause Graph Features across generators for hard instances with 100% randomness, highlighting the variations in feature coverage among different generation approaches.

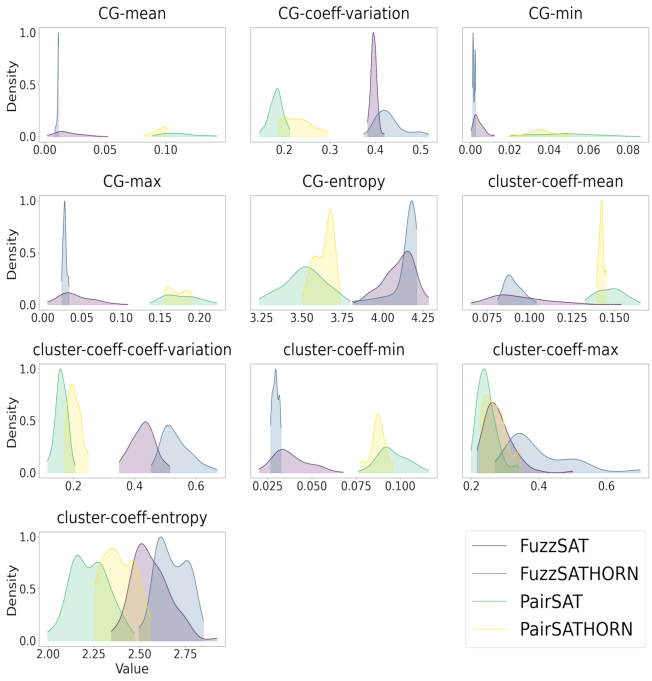

Figure 3: Distribution of Clause Graph Features for hard instances with 100% randomness.

Due to space constraints and visualization clarity, only representative distributions are presented, with Figure 3 as an illustrative example of the feature analysis methodology. Key observations from the feature analysis include:

1. PairSAT and PairSATHORN easy configurations produce instances where the feature extractor can reduce all of their original number of clauses and variables to 0. This might be because these instances exhibit duplicate unary clauses.
2. FuzzSAT shows expected behavior where increased randomness and hardness correlate with smaller variable-to-clause ratios [Nudelman *et al.*, 2004].
3. The HornSAT generator maintains consistent Variable Graph and Variable-Clause Graph features with the original input instance while showing more variation in Clause Graph features.

After fuzzing all of the solvers on the instances described previously, the results in Table 4 were obtained.

Counter	Correct	Incorrect	Crash	Timeout
D4	1042	0	102	260
ExactMC	1043	0	0	361
GPMC	1063	0	2	339
Ganak	1086	0	137	181

Table 4: Fuzzing results by model counter.

The incorrect count is determined by comparing each computed count across all solvers and doing majority voting. As shown in Table 4, the four selected state-of-the-art solvers demonstrated the following behaviors:

- All solvers achieved similar correct count rates.
- No incorrect counts were observed across any solver.
- Crashes and timeouts significantly varied, indicating different performance characteristics across solver implementation.

Table 5 presents the fuzzing results categorized by instance generator rather than by solver.

Generator	Hardness	Random	Correct	Incorrect	Crash	Timeout
FuzzSAT	easy	0%	397	0	0	3
		50%	400	0	0	0
		100%	400	0	0	0
	hard	0%	157	0	79	164
		50%	400	0	0	0
		100%	400	0	0	0
PairSAT	easy	0%	400	0	0	0
		100%	400	0	0	0
	hard	0%	39	0	55	306
		100%	267	0	71	62
FuzzSAT-HORN	easy	100%	404	0	0	0
	hard	100%	68	0	3	333
PairSAT-HORN	easy	100%	404	0	0	0
	hard	100%	98	0	33	273

Table 5: Fuzzing results by generator.

Already, some interesting patterns regarding instance hardness emerge. Contrary to our initial assumptions, generators configured for presumably hard instances, which would normally result in more timeouts and crashes, did not consistently produce these results. For example, with FuzzSAT, increasing the proportion of random clauses in hard instances made them easier to solve and less likely to trigger bugs. This pattern is visible in Figure 4, which shows the average solve time across different generator configurations. Generally, instances with higher randomness control parameters (either more random clauses in FuzzSAT or unbalanced variable distribution with variable clause-to-variable ratios in PairSAT) were solved more quickly. They produced fewer timeouts and crashes than their more structured counterparts.

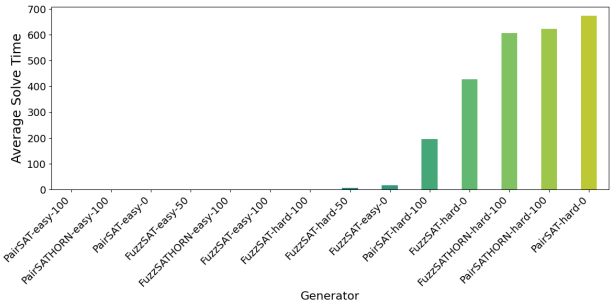

Figure 4: Average solve time by generator.

Also evident from the fuzzing results is that applying the HornSAT modifier to either FuzzSAT or PairSAT hard instances consistently increases their computational difficulty. The instances produced by each HornSAT modification show higher average solve times and generate more timeouts and crashes than their original counterparts. They also produce instances with a much higher count or number of solutions, as visible in Figure 5.

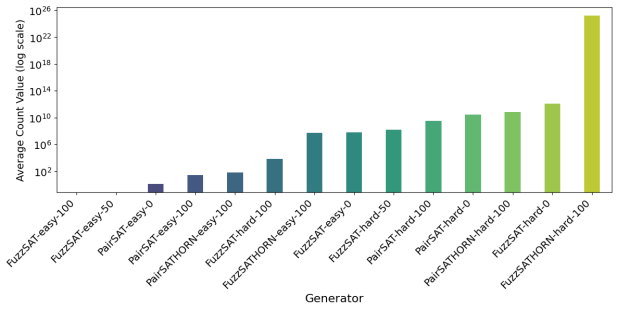

Figure 5: Average count value by generator.

Following the similarity evaluation strategy defined in Section 5.6, every instance from each generator was compared pairwise with all instances from other generators. Figure 6 shows the resulting similarity matrix, defining the similarity score between every two generators.

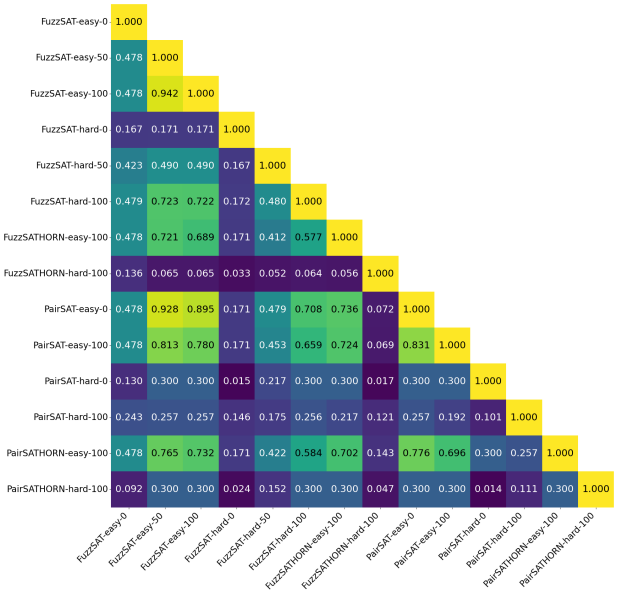

Figure 6: Pairwise similarity scores between generators, displayed as a triangular matrix to avoid redundant entries.

The matrix highlights distinct yet predictable patterns. Instances generated by presumably easy configurations consistently achieve high similarity scores (0.7 to 0.9), indicating similar solver behavior and count values. In contrast, instances generated by harder configurations show lower scores (0.3 to 0.5), reflecting more diverse outcomes, including timeouts and crashes. HornSAT-modified hard instances and PairSAT-hard-0 instances have the lowest similarity scores (< 0.1), which is to be expected because of the longer solve times, higher model counts, and increased timeouts (see Figures 4 and 5).

In order to answer **RQ2**, we investigated how instance feature distributions predict behavioral similarity between generators. Spearman correlation, which captures monotonic relationships without assuming linearity, was used between feature distribution similarities and behavioral similarity scores. Since our behavioral similarity metric already incorporates solve time and model count differences through specific weights, no additional accounting for these confounding variables was needed.

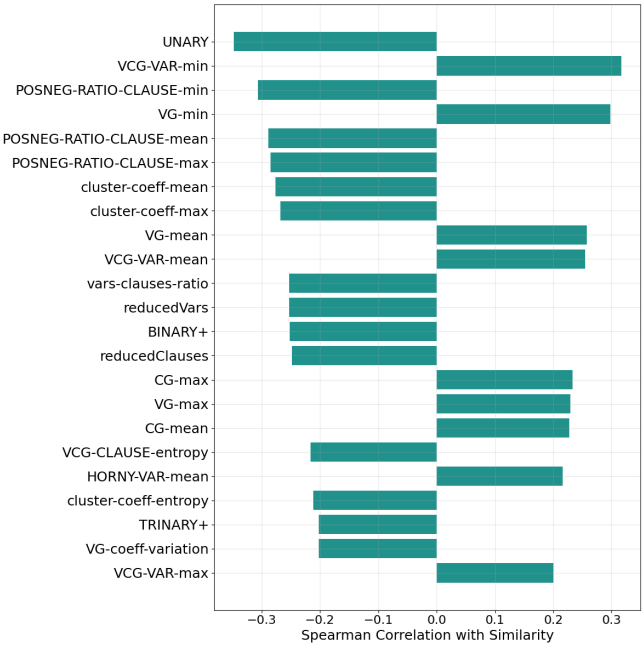

Figure 7: Spearman correlation coefficients between feature distribution similarities and generator behavioral similarity scores. Features are sorted by absolute correlation strength ($|\rho| > 0.3$), showing only statistically significant correlations ($p < 0.05$).

As shown in Figure 7, only four features showed moderate absolute Spearman correlations ($|\rho| > 0.3$). Variable graph minimum degrees (VCG-VAR-min, VG-min) showed positive correlations, indicating that generators producing instances with similar basic graph structures tend to behave similarly. Conversely, minimum positive-to-negative literal ratio (POSNEG-RATIO-CLAUSE-min) and proportion of unary clauses (UNARY) showed negative correlations.

These moderate correlations suggest that examining the distribution of these four features could help researchers select which fuzzers to run by identifying potentially redundant behavior without requiring expensive fuzzing campaigns (such as the one conducted in this study). Particularly, fuzzers producing similar VCG-VAR-min and VG-min distributions might be candidates for elimination due to likely behavioral overlap. However, this method proves inconclusive in answering **RQ3** for two key reasons: first, our dataset could not generate incorrect model counts, weakening the behavioral analysis; second, only a handful of features showed moderate correlations, and these correlations rely on a subjective similarity metric that weighs different behavioral aspects (timeouts, crashes, model counts) based on chosen parameters rather than empirically derived weights.

7.2 Limitations and Future Work

Several limitations of this study suggest directions for future research. Our analysis was constrained by limited computational resources, leading to relatively short timeouts (12 minutes) and small sample sizes (100 instances per generator). Additionally, the study was limited by the few avail-

able model counters and fuzzers, requiring us to use different configurations of the same generators to obtain a statistically representative dataset.

A significant limitation in our crash analysis is that many reported crashes could have been caused by system-level process termination rather than actual solver bugs. Out-of-memory handlers and infrastructure-related issues in the fuzzing setup may have terminated solver processes prematurely, making distinguishing between genuine solver crashes and external termination events difficult.

The absence of incorrect counts in our results indicates the need for more sophisticated instance generators that better explore corner cases. Future work should investigate alternative CNF features beyond the base SATzilla set, such as treewidth measures [Dilkas, 2023] and structural entropy [Zhang *et al.*, 2021].

Our similarity metric, while functionally able to compare fuzzing results, relies on subjective weights for different behavioral aspects, and its black-box nature makes it challenging to identify definitively when different errors stem from the same underlying bug. Future work should scale up this experiment with more diverse fuzzers and model counters, develop empirically validated similarity metrics that better capture error relationships, and explore the other proposed methodologies in Section 4.

8 Conclusion

Although evaluating fuzzers is essential for improving the validation of #SAT solvers, this evaluation is no easy feat. The complex implementation of model counters, combined with limited computational resources and time constraints, makes evaluating fuzzers based on the similarity of bug-triggering behavior an essential approach. This study proposed three methods for achieving this goal while conducting a comprehensive case study using differential testing on three available fuzzers and four state-of-the-art #SAT solvers.

The results revealed moderate correlations between certain CNF features and fuzzer similarity, particularly in graph structure metrics like variable node degrees and clause polarity ratios. However, the limited number of features showing significant correlations, combined with the absence of incorrect model counts in our results and uncertainty about crash behaviors, suggests that feature-based similarity alone may not be sufficient for selecting a dissimilar subset of fuzzers that maximizes coverage.

While this approach demonstrated promise, the findings indicate the need for more advanced fuzzer evaluation techniques, particularly those incorporating white-box analysis with AFL instrumentation. The proposed methods and results contribute to a growing understanding of effectively evaluating fuzzers for #SAT solvers while working within practical resource constraints.

References

[Audemard and Simon, 2009] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404. Morgan Kaufmann Publishers Inc., 2009.

[Beck *et al.*, 2020] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Automating the Development of Chosen Ciphertext Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1821–1837. USENIX Association, 2020.

[Böhme *et al.*, 2022] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1621–1633. Association for Computing Machinery, July 2022.

[Brummayer *et al.*, 2010] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated Testing and Debugging of SAT and QBF Solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57. Springer Berlin Heidelberg, 2010.

[Bryant *et al.*, 2023] Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified Knowledge Compilation with Application to Verified Model Counting. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[Darwiche, 2004] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2004.

[Delft High Performance Computing Centre (DHPC), 2024] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2`, 2024.

[Dilkas, 2023] Paulius Dilkas. Generating Random Instances of Weighted Model Counting. In Andre A. Cire, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, CPAIOR 2023, pages 395–416. Springer Berlin Heidelberg, 2023.

[Fioraldi *et al.*, 2020] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies*, WOOT 20. USENIX Association, 2020.

[Ganian and Szeider, 2021] Robert Ganian and Stefan Szeider. New width parameters for SAT and #SAT. *Artificial Intelligence*, 295, June 2021.

[Girol *et al.*, 2021] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In A. Silva and K. R. M. Leino, editors, *33rd International Conference on Computer Aided Verification*, CAV 2021, pages 669–693. Cham: Springer International Publishing, 2021.

[Godefroid *et al.*, 2008a] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing.

In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215. Association for Computing Machinery, 2008.

[Godefroid *et al.*, 2008b] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium (NDSS)*, volume 8, pages 151–166. The Internet Society, 2008.

[Godefroid, 2012] Patrice Godefroid. SAGE: Whitebox Fuzzing for Security Testing. *SAGE*, 10(1), 2012.

[Gomes *et al.*, 2021] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In Armin Biere, Marijn Heule, and Hans van Maaren, editors, *Handbook of Satisfiability: Second Edition*, Frontiers in Artificial Intelligence and Applications, chapter 25. IOS Press, 2021.

[Kabir and Meel, 2023] Mohimenul Kabir and Kuldeep S. Meel. A Fast and Accurate ASP Counting Based Network Reliability Estimator. In Ruzica Piskac and Andrei Voronkov, editors, *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning,*, volume 94, pages 270–287, 2023.

[Klees *et al.*, 2018] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. Association for Computing Machinery, 2018.

[Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 667–673, 2017.

[Lai *et al.*, 2021] Yong Lai, Kuldeep S. Meel, and Roland H. C. Yap. The Power of Literal Equivalence in Model Counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35(5), pages 3851–3859, 2021.

[Latour and Soos, 2024] Anna L.D. Latour and Mate Soos. SharpVelvet, 2024.

[Latour *et al.*, 2022] Anna L. D. Latour, Behrouz Babaki, Daniël Fokkinga, Marie Anastacio, Holger H. Hoos, and Siegfried Nijssen. Exact Stochastic Constraint Optimization with Applications in Network Analysis. *Artificial Intelligence*, 304:103650, 2022.

[Nudelman *et al.*, 2004] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*, CP 2004, pages 438–452. Springer Berlin Heidelberg, 2004.

[Schloegel *et al.*, 2024] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993, 2024.

[Sharma *et al.*, 2019] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, IJCAI'19, pages 1169–1176, 2019.

[Shavit and Hoos, 2024] Hadar Shavit and Holger H. Hoos. Revisiting SATZilla Features in 2024. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 27:1–27:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[Suzuki *et al.*, 2017] Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. Improvement of Projected Model Counting Solver with Component Decomposition using SAT Solving in Components. Technical report, JSAI Technical Report, SIG-FPAI-506-07, 2017.

[Teuber and Weigl, 2021] Samuel Teuber and Alexander Weigl. Quantifying Software Reliability via Model-Counting. In Alessandro Abate and Andrea Marin, editors, *18th International Conference on Quantitative Evaluation of Systems*, QEST 2021, pages 59–79. Springer International Publishing, 2021.

[Thurley, 2006] Marc Thurley. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In Armin Biere and Carla P. Gomes, editors, *9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, pages 424–429. Springer Berlin Heidelberg, 2006.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-Based Algorithm Selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

[Zalewski, 2016] Michał Zalewski. American Fuzzy Lop - Whitepaper. `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2016.

[Zeller *et al.*, 2024] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-07-01 16:50:18+02:00.

[Zhang *et al.*, 2021] Zaijun Zhang, Daoyun Xu, and Jincheng Zhou. A Structural Entropy Measurement Principle of Propositional Formulas in Conjunctive Normal Form. *Entropy*, 23(3):303, 2021.