On Quantitative Testing of Samplers

Mate Soos
National University of Singapore, Singapore

Priyanka Golia
Indian Institute of Technology Kanpur, India
National University of Singapore, Singapore

Sourav Chakraborty
Indian Statistical Institute Kolkata, India

Kuldeep S. Meel
National University of Singapore, Singapore

Abstract

The problem of uniform sampling is, given a formula F, sample solutions of F uniformly at random from the solution space of F. Uniform sampling is a fundamental problem with widespread applications, including configuration testing, bug synthesis, function synthesis, and many more. State-of-the-art approaches for uniform sampling have a trade-off between scalability and theoretical guarantees. Many state of the art uniform samplers do not provide any theoretical guarantees on the distribution of samples generated, however, empirically they have shown promising results. In such cases, the main challenge is to test whether the distribution according to which samples are generated is indeed uniform or not.

Recently, Chakraborty and Meel (2019) designed the first scalable sampling tester, Barbarik, based on a grey-box sampling technique for testing if the distribution, according to which the given sampler is sampling, is close to the uniform or far from uniform. They were able to show that many off-the-self samplers are far from a uniform sampler. The availability of Barbarik increased the test-driven development of samplers. More recently, Golia, Soos, Chakraborty and Meel (2021), designed a uniform like sampler, CMSGen, which was shown to be accepted by Barbarik on all the instances. However, CMSGen does not provide any theoretical analysis of the sampling quality.

CMSGen leads us to observe the need for a tester to provide a quantitative answer to determine the quality of underlying samplers instead of merely a qualitative answer of Accept or Reject. Towards this goal, we design a computational hardness-based tester ScalBarbarik that provides a more nuanced analysis of the quality of a sampler. ScalBarbarik allows more expressive measurement of the quality of the underlying samplers. We empirically show that the state-of-the-art sampler, CMSGen is not accepted as a uniform-like sampler by ScalBarbarik. Furthermore, we show that ScalBarbarik can be used to design a sampler that can achieve balance between scalability and uniformity.

2012 ACM Subject Classification Theory of computation; Computing methodologies → Artificial intelligence

Keywords and phrases SAT Sampling, Testing of Samplers, SAT Solvers

Digital Object Identifier 10.4230/LIPIcs.CP.2022.36

Supplementary Material *Software (Source Code)*: `https://github.com/meelgroup/scalbarbarik`

Funding This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004], Ministry of Education Singapore Tier 2 grant [MOE-T2EP20121-0011], NUS ODPRT grant [R-252-000-685-13], and Amazon Research Award.

Acknowledgements We are grateful to the anonymous reviewers for constructive comments that significantly improved the final version of the paper. We are thankful to Yash Pote for his detailed feedback on the early drafts of the paper. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: `https://www.nscc.sg`.

© Mate Soos, Priyanka Golia, Sourav Chakraborty, and Kuldeep S. Meel;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Given a formula F over the set of variables X, the problem of Boolean satisfiability (SAT) is to determine whether there exists an assignment σ to X such that F evaluates true under σ. The past two decades have witnessed a dramatic improvement in the runtime of SAT solvers owing to the Conflict Driven Clause Learning (CDCL) paradigm, and as a result, SAT solvers find applications in diverse areas ranging from constrained-random verification [19], computational biology [10], and artificial intelligence. The progress in SAT solving has led to development of algorithmic and practical implementations for problems in complexity classes beyond NP. One such problem that has seen a sustained interest over the past decade is that of uniform sampling. The problem of uniform sampling is to sample satisfying assignments of a formula uniformly at random from the space of satisfying assignments of the formula. Like SAT solver, uniform sampling also has wide variety of applications, like in configuration testing [7, 15], constrained-random simulation [19], bug synthesis [21], and function synthesis [12].

The last decade has seen several algorithmic proposals for efficient uniform sampling owing to its diverse applications. The different techniques for uniform sampling can be divided into two categories: (1) techniques that provide theoretical guarantees on the distribution from which the samples are generated, and (2) techniques that do not provide any theoretical guarantees on the samples produced. The hashing-based sampler UniGen, UniGen3 [6, 5, 23], and the knowledge compilation-based sampler KUS [22] fall in the first category, however, experimental evaluation shows that these samplers could not always achieve scalability for real world instances. At the same time, there exist many other sampling techniques, such as the mutation-based QuickSampler [8] and BDD-based techniques [16], or randomized CDCL SAT solvers [13] that can provide empirical scalability, however do not provide guarantees on the distribution of samples generated.

Algorithmic proposals that cannot provide theoretical grantees on the distribution of samples generated often rely on statistical test such as KL-divergence [17] to showcase the quality of the samples generated. These statistical tests are only able to show that samples produced by the samplers for a small set of benchmarks are close to samples produced from a uniform distribution. However, such tests do not generalize over entire benchmark sets. Recently, Chakraborty and Meel proposed the first scalable sampling test framework, Barbarik [3], to test whether a sampler under test (SUT[1]) is close to uniform or not. The tester Barbarik takes an (1) SUT, a (2) base uniform sampler, a (3) tolerance parameter ε, an (4) intolerance parameter η, a (4) confidence parameter δ, and a (5) formula φ and returns Accept if SUT is close to a uniform sampler. Barbarik returns Reject only if the SUT is far from a uniform sampler under *subquery-consistency* assumption, which is to assume that the SUT does not change its sampling behavior during the test, that is, off the shelf samplers would be sub-query consistent[2].

The main idea behind Barbarik is to reduce the input formula φ to $\hat{\varphi}$ using two satisfying assignments of φ chosen uniformly at random from the solution space of φ. One assignment, say σ_1 is drawn using the SUT, and another assignment, say σ_2 is drawn according to uniform distribution using the base sampler. The analysis for Barbarik shows that if the distribution from which the SUT is sampling the assignments is close to uniform distribution, the conditional distribution over $\{\sigma_1, \sigma_2\}$ is also close to uniform. Similarly, if the distribution

[1] The term SUT is from software testing literature, where it is shorthand for System Under Test.
[2] We rename the notion of *non-adversarial* assumption introduced in [3] to subquery-consistency to better capture its intended properties. We formally define the subquery-consistent assumption in Section 2.

from which the SUT is sampling the assignments is far from uniform, the conditional distribution over $\{\sigma_1, \sigma_2\}$ is also far from uniform. It is easy to estimate the distance of conditional distribution over $\{\sigma_1, \sigma_2\}$ to uniform distribution using random samples from $\hat{\varphi}$. Empirically, it was shown that Barbarik accepts UniGen3, which is a sampler with theoretical grantees, however, it rejects the state of the art uniform-like samplers, that is, samplers without theoretical guarantees, such as QuickSampler [8] and STS [9]. Recently, Meel et al. generalize the idea of Barbarik to handle any arbitrary weight function, that is, to test whether a SUT generates samples according to a given distribution [18].

Recently, Golia, et al. used Barbarik in a test-driven development fashion to create the uniform-like sampler CMSGen [13] from the state-of-art SAT solver CryptoMiniSat [24]. CMSGen is based on randomization of the conflict-driven-clause-learning (CDCL) framework inside CryptoMiniSat and most modern SAT solvers. Based on the feedback from Barbarik, the authors iteratively changed the hyper-parameters of CryptoMiniSat such as restart intervals, restart types, polarity picking heuristics and the like, until they arrived at a point where it was able to pass all tests. Analyzing the CDCL itself is a hard problem, and so the resulting uniform-like sampler, CMSGen, could not provide theoretical guarantees on the distribution of samples produced. However, it was shown in [13] that Barbarik returns Accept for CMSGen.

The development of samplers such as CMSGen poses an interesting question regarding test frameworks such as Barbarik: is it possible that uniform-like samplers such as CMSGen pass the test, but they are not uniform? If so, how can one demonstrate that they are not? These questions point towards revisiting the design of sampler test frameworks such as Barbarik. We need a tester that provides a quantitative analysis instead of qualitative answer of Accept or Reject to measure the quality of samplers.

The above stated goal to improve sampling testers requires new insights about the workings of samplers. The improvement of Barbarik that we are envisioning is to generate input formulas that are specifically crafted to highlight non-uniformity in the samples produced by the samplers. Towards this goal, we propose the framework ScalBarbarik.

Contributions

The success of CMSGen and the current lack of theoretical analysis leads us to hypothesize that CMSGen may not be uniform for all the formulas but is not necessarily far from uniform for a large class of formulas. The current framework of Barbarik provides too coarse grained analysis to allow users to determine the quality of distributions generated by a sampler such as CMSGen. To achieve such a fine-grained analysis, we need a parameterized generation of $\hat{\varphi}$. To this end, we design an improved algorithm, Shakuni, for construction of $\hat{\varphi}$ such that $\hat{\varphi}$ is composed of two sub-formulas with varying computational hardness.

We augment Barbarik with Shakuni to obtain ScalBarbarik that can provide fine-grained analysis with respect to hardness dial provided by Shakuni. ScalBarbarik allows us to view that the distribution quality of CMSGen is better than samplers such as QuickSampler but falls short of samplers with rigorous guarantees such as UniGen. ScalBarbarik can then be used to fine-tune a heuristic-based uniform-like sampler such as CMSGen to achieve a different balance between scalability and uniformity. Towards this, we empirically analyze the distribution of samples generated by CMSGen with different restart intervals. We then show that CMSGen could generate samples from a close to uniform distribution with increased restart intervals, sacrificing speed for better uniformity.

It is worth remarking that an important strength of ScalBarbarik is its simplicity. Based on our empirical analysis, ScalBarbarik with varying computational hardness is able to show that CMSGen is not a uniform sampler. The availability of ScalBarbarik has the potential to spur a virtuous cycle of development of samplers and testing techniques: the developers

can design sampling methods that can be accepted by testers such as Barbarik/ScalBarbarik and consequently improve testers so that such samplers are rejected in the following version of it. With the help of ScalBarbarik, we can tune a sampler to achieve the balance between scalability and uniformly. Our experimental evaluation demonstrates that as we increase the restart intervals of CMSGen, we need to increase the computational hardness of ScalBarbarik to reject CMSGen, that is, with increased restart intervals CMSGen is able to generate samples from a close to uniform distribution; however, it takes longer time to generate the samples. The availability of ScalBarbarik allows us to improve to samplers such as CMSGen.

The rest of the paper is organized as follows: In Section 2, we present the formal definitions and also present a brief description of state-of-the-art tester Barbarik. In Section 3, we present the improved test framework ScalBarbarik based on a cryptographically hard function. We provide a detailed algorithmic description in Section 4, and we present the experimental evaluation in Section 5. Finally, we conclude in Section 6.

2 Notation and Background

A literal is a Boolean variable or its negation. A formula is considered to be in conjunctive normal form (CNF) if the formula is conjunction of clauses. A clause is a disjunction of literals. Let φ be the formula in CNF, and let $Supp(\varphi)$ represent the set of variables in φ. A satisfying assignment to φ is an assignment of truth values to $Supp(\varphi)$ under which the formula φ evaluates to True. Let σ be a satisfying assignment of φ, and let $S \subseteq Supp(\varphi)$, $\sigma_{\downarrow S}$ represents the projection of σ over S. Let R_φ be the set of all satisfying assignments of formula φ. We used $L[n:m]$ to represent the substring of L, starting with position n to m.

Chain Formulas. Chain formulas were introduced in [4]. Given positive integers k and m, chain formulas are Boolean formulas with exactly k satisfying solutions with $\lceil \log(k) \rceil \leq m$ variables.

▶ **Definition 1** ([4]). *Let $c_1, c_2, \ldots, c_m$ be the m-bit binary representation of k, where c_m is the least significant bit. A chain formula $\varphi_{k,m}(.)$ on m Boolean variables $v_1, v_2, \ldots, v_m$ is as follows:*
For every j in $\{1, \ldots, m-1\}$, let C_j be the connector "$\vee$" if $c_j = 1$, and the connector "$\wedge$" if $c_j = 0$, and the formula $\varphi_{k,m}(v_1, v_2, \ldots, v_m) = v_1 C_1 (v_2 C_2 (\ldots (v_{m-1} C_{m-1} v_m)))$

A Sampler. A CNF sampler or simply a sampler takes a formula φ, a number of required satisfying assignments N, $S \subseteq Supp(\varphi)$, and returns satisfying assignments $\sigma_{1_{\downarrow S}}, \sigma_{2_{\downarrow S}}, \ldots, \sigma_{N_{\downarrow S}}$. A uniform sampler, say $\mathcal{G}$ takes $\varphi, N, S \subseteq Supp(\varphi)$ that generates a satisfying assignment σ_i for all $i \in \{1, N\}$ with probability $\dfrac{1}{|R_\varphi|}$. Similarly, a sampler is considered to be an additive almost-uniform sampler, if the following holds with $0 \leq \varepsilon \leq 1$:

$$\forall \sigma \in R_\varphi, \frac{1-\varepsilon}{|R_\varphi|} \leq Pr[\mathcal{G}(\varphi, N) = \sigma] \leq \frac{1+\varepsilon}{|R_\varphi|}$$

We use a sampler $\mathcal{G}(.,.,.)$ or $\mathcal{G}(.,.)$ when S is $Supp(\varphi)$, or simply $\mathcal{G}$ when N and S are clear from context. We use $p_{\mathcal{G}(.,.,.)}$ to denote the probability with that $\mathcal{G}$ samples a satisfying assignment σ, and $D_{\mathcal{G}(\varphi,.,.)}$ to denote the distribution induced by sampler G over solution space of φ.

Given a formula φ, and an intolerance parameter η, a sampler $\mathcal{G}$ is considered to be η-far from a uniform sampler if ℓ_1 distance between the distribution induced by $\mathcal{G}$ over solution space of φ to the uniform distribution is at least η, that is,

$$\sum_{x \in sol(\varphi)} \left| p_{\mathcal{G}(\varphi,x)} - \frac{1}{|sol(\varphi)|} \right| \geq \eta$$

A Sampler Tester. Given a uniform sampler, a sampler tester tests if the sampler is sampling an assignment from the solution space R_φ, and the samples are generated from a close to uniform distribution. A sampler test framework is defined as follows:

▶ **Definition 2.** *Given a Boolean formula φ, a sampler $\mathcal{G}$, a tolerance parameter ε, an intolerance parameter η, a confidence parameter δ, a sampler tester $\mathcal{T}(\cdot,\cdot,\cdot,\cdot,\cdot)$ returns* Accept *or* Reject *(with a witness) with the following guarantees:*

1. *If the sampler $\mathcal{G}(\varphi,.,.)$ is an additive almost-uniform generator, then $\mathcal{T}(G,\varphi,\varepsilon,\eta,\delta)$ returns* Accept *with probability at least $1-\delta$*
2. *If the sampler $\mathcal{G}(\varphi,.,.)$ is η-far from uniform generator, then $\mathcal{T}(G,\varphi,\varepsilon,\eta,\delta)$ returns* Reject *with probability at least $1-\delta$*

Barbarik

Chakraborty and Meel [3] designed the tester Barbarik that takes a base uniform sampler $\mathcal{U}$, a Sampler Under Test (SUT) $\mathcal{G}$, a tolerance parameter ϵ, an non-tolerance parameter η, and a confidence parameter δ. ϵ, η, δ take values between 0 to 1. The problem under consideration is to distinguish between the case where $\mathcal{G}$ is close to $\mathcal{U}$, and the case when $\mathcal{G}$ is far from $\mathcal{U}$. We know the probability of each assignment in the support for uniform sampler $\mathcal{U}$, that is, $Pr[\mathcal{U}(.,.,.)] = \frac{1}{|R_\varphi|}$. However, distribution for $\mathcal{G}$ is unknown, we only have access to samples from $\mathcal{G}$. Given access to a uniform sampler $\mathcal{U}$, Barbarik provides guarantees described in Definition 2. Furthermore, in case Barbarik rejects the SUT, it also provides a CNF formula $\hat{\varphi}$ as a witness. The formula φ is reduced to $\hat{\varphi}$ such that $\hat{\varphi}$ has exactly two assignments for the variables in the support S, and the distribution $D_{\mathcal{G}(\hat{\varphi})}$ from which samples are generated for $\hat{\varphi}$ is η far from uniform.

To achieve the aforementioned guarantees, Barbarik uses the idea of conditional sampling. Barbarik samples a satisfying assignment σ_1 from the SUT $\mathcal{G}$, and another satisfying assignment σ_2 from the base uniform sampler $\mathcal{U}$. Let T be $\{\sigma_1, \sigma_2\}$. If the distribution $D_{\mathcal{G}(\varphi)}$ from which SUT is sampling is close to uniform distribution, then the conditional distribution $D_{\mathcal{G}(\varphi)|T}$ is also close to uniform distribution. Similarly, if the distribution $D_{\mathcal{G}(\varphi)}$ is far from uniform distribution, then the conditional distribution $D_{\mathcal{G}(\varphi)|T}$ is also far from uniform distribution. Therefore, instead of focusing on the distribution $D_{\mathcal{G}(\varphi)}$, Barbarik considers the distribution $D_{\mathcal{G}(\varphi)|T}$ as it is easier to test.

In order to consider the distribution $D_{\mathcal{G}(\varphi)|T}$, Barbarik constructs a formula $\hat{\varphi}$ from φ with the help of the subroutine Kernel. The subroutine Kernel takes a formula φ, two satisfying assignments σ_1 and σ_2, and an integer N which represents the number of assignments $\hat{\varphi}$ and returns a formula $\hat{\varphi}$. The subroutine Kernel ensures that $\hat{\varphi}$ and φ have the similar structure, and $Supp(\varphi) \subset Supp(\hat{\varphi})$. Furthermore, $|\forall \sigma \in R_{\hat{\varphi}} | \sigma_{\downarrow S} = \sigma_1| = |\forall \sigma \in R_{\hat{\varphi}} | \sigma_{\downarrow S} = \sigma_2|$.

The formula $\hat{\varphi}$ should satisfy the two conditions: (i) If the SUT $\mathcal{G}(\varphi)$ is ϵ-additive almost-uniform generator, the distribution from which sampler is generating samples, say $D_{\mathcal{G}(\hat{\varphi},S)}$ is *close* to uniform distribution over the set $\{\sigma_1, \sigma_2\}$, and (ii) If the SUT $\mathcal{G}(\varphi)$ is η-far from uniform sampler, then the distribution $\mathcal{U}$, the distribution $D_{\mathcal{G}(\hat{\varphi},S)}$ is *far* from uniform distribution over the set $\{\sigma_1, \sigma_2\}$.

If the sampler $\mathcal{G}$ is an additive almost-uniform generator on any input formula φ, the first condition would be satisfied. However, to satisfy the second condition, we need *subquery-consistent assumption* as per [3]:

▶ **Definition 3.** *The* ***subquery-consistent sampler assumption*** *states that if* $(\hat{\varphi}, \hat{S})$ *is the output obtained from* Kernel$(\varphi, S, \sigma_1, \sigma_2, N)$ *then*

- $S \subseteq \hat{S}$
- *the output of* $\mathcal{G}(\hat{\varphi}, S, N)$ *is* N *independent samples from the conditional distribution* $\mathcal{D}_{\mathcal{G}(\varphi,S)} \mid_T$*, where* $T = \{\sigma_1, \sigma_2\}$.

Thus, if for any formula φ the sampler $\mathcal{G}(\varphi)$ is η-far from the uniform sampler in the ℓ_1 distance and the sampler satisfies the **subquery-consistent sampler assumption** then Barbarik will Reject with probability $(1 - \delta)$.

3 A Quantitative Tester

The behavior of Barbarik shows that while Barbarik is able to return Reject for samplers without guarantees such as STS or QuickSampler, it returns Accept for CMSGen. It is important to note that the theoretical analysis of soundness of Barbarik is unconditional but the analysis of completeness is conditional, i.e., when Barbarik returns Reject, then the sampler is non-uniform, but the output Accept from Barbarik needs to be interpreted through the lens of *subquery-consistent* assumption.

It is worth emphasizing that the existence of strong lower bounds on the black-box approach necessitates introduction of a grey-box approach, and in turn subroutines such as Kernel along with *subquery-consistent* assumption are likely unavoidable. Therefore, in order to improve Barbarik, we focus on extending Kernel via parameterization to allow a nuanced analysis of the quality of distributions. To this end, we first focus on identifying properties of formulas that may make it hard for algorithms without rigorous guarantees to sample well.

3.1 Computational Hardness

As discussed in Section 1, there are a number of decisions taken by CMSGen, as in all samplers and solvers, for increasing efficiency. Many of these decisions/heuristics are inherited from CryptoMiniSat. One of the crucial components of CDCL-based SAT solvers is the usage of restarts [2]. While theoretical understanding of the power and need for restarts in CDCL SAT solvers is limited, a predominant view among practitioners is that frequent restarts help the solver avoid being stuck in a part of assignment space.

The usage of heuristics that seek to avoid a sampler being stuck in a part of assignment space may have implications on its ability to sample uniformly. In particular, one can argue that usage of frequent restarts may lead CMSGen to not sample uniformly for a certain class of formulas, where the solution space of the formula can be categorized into *easy* and *hard* – such that solutions belong to the *easy* set are easier to find without the need for excessively large number of conflicts while the solutions belonging to the *hard* set require significantly more conflicts. In such a scenario, CMSGen may find it harder to sample uniformly as the restarts will push CMSGen towards the easier side while it may almost never end up finding an assignment from the harder side. At this point, one may ask if this observation can be used to inform the design of the sampler tester.

To design a test framework to Reject a sampler such as CMSGen, we need to formalize our observation. To this end, we seek to define the notion of *computational hardness* for our case formally. At the onset, it is worth accepting that our limited understanding of the

workings of CDCL solvers in the context of classical complexity-theoretic notions imply that we need to use constructs based on practical aspects of SAT solvers. Roughly speaking, the computational hardness of a CNF-formula should indicate how hard it is for a SAT solver to find a satisfying assignment. It is well known that while modern SAT solvers are extremely efficient at solving many problems, there are entire classes of problems that pose significant challenges. One such class of problem is cryptographic challenges, which are designed to be hard to be solved by any tool. The consumption of resources such as a memory by an algorithm varies with time, and we seek to capture the peak resource consumption as follows:

▶ **Definition 4.** *Given an algorithm $\mathcal{A}$, input $\mathcal{I}$, and time t, for a particular run of the algorithm $\mathcal{A}$ on input $\mathcal{I}$ the* $\mathsf{PeakCost}(\mathcal{A}, \mathcal{I}, t)$ *measures the maximum resource consumption by $\mathcal{A}$ at execution step t on that particular run. This function is a non-decreasing function in t and stops to increase from the moment the run of the algorithm stops.*

Given a set of solvers/samplers $\mathcal{G}$, a CNF-formula φ is said to have computational hardness κ with respect to $\mathcal{G}$ if for $\mathcal{A} \in \mathcal{G}$

$$\Pr[\lim_{t\to\infty}\{\mathsf{PeakCost}(\mathcal{A}, \varphi, t)\} \geq \kappa] \geq 1 - o(1),$$

where the probability is taken over the internal randomness of $\mathcal{A}$, and $o(1)$ refers to "little-o" notation.

To capture the behavior of samplers that employ cutoff parameters, we define the notion of intractable formulas for cutoff κ as the set of formulas whose computational hardness is at least κ with respect to $\mathcal{G}$, i.e.,

▶ **Definition 5.** $\mathsf{Intractable}(\kappa, \mathcal{G}) = \{\varphi \mid \varphi \text{ has computational hardness} \geq \kappa \text{ w. r. t. } \mathcal{G}\}$

In the next section, we seek to use the notion of $\mathsf{Intractable}(\kappa, \mathcal{G})$ to improve Barbarik.

3.2 From Kernel to Shakuni

In this section, we turn to the design of an improved version of Barbarik, called ScalBarbarik, that can employ the set of formulas belonging to Intractable so as to distinguish samplers that were beyond the reach of Barbarik. ScalBarbarik takes as input an SUT $\mathcal{G}$, a uniform sampler $\mathcal{U}$, tolerance parameter ε, intolerance parameter η, accuracy parameter δ, a CNF-formula φ, a set $S \subseteq Supp(\varphi)$ and a computational hardness parameter κ. It outputs Accept or Reject depending on whether the SUT is ε-additive close to a uniform sampler or whether it is η-far from the uniform sampler. It is supposed to output the correct answer with probability at least $(1 - \delta)$. The computational hardness parameter is passed onto the subroutine Shakuni.

Shakuni takes in a CNF-formula φ, a set $S \subseteq Supp(\varphi)$, two assignments σ_1 and σ_2 from $sol(\varphi)_{\downarrow S}$ and a positive integer N and returns a new formula $\hat{\varphi}$ such that the following conditions are satisfied:

- $\hat{\varphi}$ has at least N satisfying assignments
- Every satisfying assignment of $\hat{\varphi}$ restricted to the set S is either σ_1 or σ_2
- If R_{σ_1} and R_{σ_2} are the set of assignments of $\hat{\varphi}$ that when restricted to the set S is σ_1 and σ_2 respectively, then $|R_{\sigma_1}| = |R_{\sigma_2}|$

In contrast to Kernel [3], Shakuni constructs $\hat{\varphi}$ such that the set R_{σ_1} is significantly different from the set R_{σ_2} in a structure such that finding assignments from one is easier than finding assignments from the other. More precisely, Shakuni assumes access to a subroutine GenHard that takes in the computational hardness parameter κ and estimated count parameter τ as inputs and returns $(\psi, \hat{\tau})$ such that $\hat{\tau} = |sol(\psi)|$ and $\psi \in \mathsf{Intractable}(\kappa, \mathsf{C_{CDCL}})$ where $\mathsf{C_{CDCL}}$ refers to the set of all the *efficient* CDCL solvers. As discussed above, given our lack of

understanding of CDCL solvers, we do not seek to define $\mathsf{C}_{\mathsf{CDCL}}$ formally, but we discuss the approach to construct formulas that seem to exhibit desired properties in practice in Section 3.3.

Assuming existence to GenHard, Shakuni starts by first finding a formula ψ with computational hardness parameter κ. Then, it uses ψ to construct the CNF-formula $\hat{\varphi}$ such that the assignments in R_{σ_1} correspond to solutions of ψ while the assignments of R_{σ_2} corresponds to solutions of a Chain Formula obtained according to [4] and having a much smaller computational hardness measure.

3.3 Formulas with Computational Hardness Measure

As discussed above, Shakuni (and in turn, ScalBarbarik) assumes access to a subroutine GenHard that takes in a counting parameter τ and hardness parameter κ and returns a formula $(\psi, \hat{\tau})$ such that (1) $|sol(\psi)| = \hat{\tau}$, where $\hat{\tau} \approx \tau$, and (2) the hardness of finding a solution of ψ using a CDCL-based SAT solver is proportional to κ.

To this end, we employ the construct of cryptographic hash functions, widely studied in cryptography. A cryptographic hash family, $\mathcal{H}_{crypto} := \{h : \{0,1\}^* \to \{0,1\}^m\}$ is a family of hash functions that compute a fixed-length hash value, also known as *fingerprint*, for arbitrarily long message msg. In the context of this work, we are interested in a collection of such families, $\{\mathcal{H}^1_{crypto}, \mathcal{H}^2_{crypto}, \ldots, \mathcal{H}^\kappa_{crypto} \ldots\}$ that satisfy the following two properties:

Pre-Image Resistance. For all $h \in \mathcal{H}^\kappa_{crypto}$, given y, the computational hardness of the task of finding msg such that $h(\mathsf{msg}) = y$ is a monotonically non-decreasing function of the hardness[3] parameter κ [11]. In our context, we are interested in the hardness measured as runtime of a CDCL SAT solver to find msg such that $h(\mathsf{msg}) = y$.

(Weak) Collision Resistance. For $x, y \in \{0,1\}^*$ we have $\Pr[h(x) = h(y)] \approx \frac{1}{2^m}$, where probability is defined over random choice of x and y.

The understanding in the cryptographic community is that most of the widely used hash families satisfy the above properties. In this work, we work with one of the widely studied hash families, SHA-1, whose hardness parameter can be varied by changing the number of so-called *rounds* of the algorithm [14]. We exploit the above properties of SHA-1 to be able to generate formulas that are similar but have tunable complexity and number of solutions. We use the SHA-1 preimage CNF instance generator[4] by Nossum [20], which generates the function $\mathcal{H}_{\text{SHA-1}} := \{h : \{0,1\}^{512} \mapsto \{0,1\}^{160}\}$. The generator allows us to set any number of randomly fixed input bits, any number of output bits, and to vary the number of rounds κ. For example, using 10 rounds, fixing 0 bits of input and 160 bits of output, the generator takes a random 512 bits input msg, runs SHA-1 on msg to obtain y, then generates a formula to encode the problem $h^{-1}(y)$, where $h \in \mathcal{H}^{10}_{\text{SHA-1}}$.

We need to construct a formula ψ with predefined number of satisfying assignments. Therefore, in order to be able to decide the number of satisfying assignments of the generated formula, and to have adjustable complexity, we change the problem slightly. We consider a random 512 bits input, msg, and we calculate $y = h_\kappa(\mathsf{msg})$, where κ is the number of rounds. We generate the formula ψ using the generator as above, encoding the function $y = h_\kappa(\mathsf{msg})$. We then fix the first e bits of msg and the first f bits of y in ψ. Hence, our formula has the following parameters: κ, e, f. We use these parameters to allow us to generate any number of problems of approximate complexity and of approximate number of solutions.

[3] A formal characterization from complexity theoretic viewpoint along with the standard cryptographic assumptions is beyond the scope of this work.

[4] Available at `https://github.com/vegard/sha1-sat`

Generating hard problems with multiple solutions

Due to the collision resistance effect, with $\kappa = 80, e = 500, f = 160$, it is most likely that there is only one solution to the generated formula: there are only 12 bits to vary for msg and there is at least one solution given the way the problem is generated. Checking the actual number of solution is easy given an optimized SHA-1 implementation, as it only needs 2^{12} executions of SHA-1. Now, to create a formula with multiple solutions, let us consider the parameters $\kappa = 80, e = 500, f = 0$. Here, there are almost certainly 2^{12} solutions, as any lower than 2^{12} would mean a collision on SHA-1, which is extremely unlikely. However, this formula is very easy to solve, as any of the 12 bits can be varied and a solution obtained.

Putting the above two cases together, one might use the parameters $\kappa = 80, e = 500, f = 5$ to get the number of solutions to be approximately $s = 2^{512-e-f} = 2^7$. There are 12 bits that are unset in the input and there are 5 bits set in the output, leading to a difference of 7 bits combined with the weak collision effect, leads to approximate 2^7 solutions. If we generate with the same parameters but $f = 6$ the number of solutions halves, and the complexity of finding a solution approximately doubles, as now there is one more fingerprint bit that must match. To change the complexity with a finer grain than doubling or halving it, one can also change the number of rounds, κ. Therefore, we can vary κ, e and f to generate a formula ψ with varying complexity that can have solution $\hat{\tau}$, where $\hat{\tau}$ approximate the τ.

4 Algorithmic Description

We augment Barbarik with Shakuni to obtain ScalBarbarik. We now provide the detailed algorithm description of Shakuni.

Algorithm 1 presents the pseudocode of the Shakuni subroutine. Shakuni takes a formula φ, two satisfying assignments of φ, σ_1 and σ_2, the desired number of samples τ, and the hardness parameter κ. Shakuni assumes access to following two subroutines:

- GenHard: Takes a counting parameter τ and hardness parameter κ and returns a formula $(\psi, \hat{\tau})$.
- ConstructChain: Takes $\hat{\tau}$ and variables of ψ as input and constructs a chain formula $\hat{\psi}$ as discussed in Section 2.

Shakuni first finds a lit that is the first literal that appears in σ_1, but not in σ_2. On line 2, Shakuni conditions the formula φ over σ_1 and σ_2, and considers the new formula as φ'. Then, on line 3, Shakuni calls GenHard subroutine with τ and κ. GenHard returns a formula ψ and $\hat{\tau}$. On lines 4 and 5 Shakuni constructs the formula $\hat{\varphi}$. $\hat{\varphi}$ is the formula φ' conjuncted with positive literal lit implies ψ, and literal $\neg$lit implies the formula returned by ConstructChain. Finally, Shakuni adds the variables of $\hat{\varphi}$ in S, and stores them as $\hat{S}$ on line 6. Finally, Shakuni returns the formula $\hat{\varphi}$ and $\hat{S}$.

As discussed, Shakuni assumes access to the subroutine GenHard. Algorithm 2 presents GenHard. GenHard takes a integer τ, and a hardness parameter κ as inputs. GenHard further assumes access to following two subroutines:

- Compute: Takes an integer τ and returns two positive integers m and f such that $m - f$ is equal to $\lceil \log \tau \rceil$.
- NossumFormulaGen: Takes the SHA-1 number of rounds κ, integers m and f, and strings over $\{0,1\}$ M and F. It considers a random 512 bits msg and fixes the first m bits of msg to M. It runs SHA-1 with κ rounds on msg to obtain y, whose first f bits are fixed to F. NossumFormulaGen returns a formula ψ which encodes the problem $h_\kappa^{-1}(y)$.

Algorithm 1 Shakuni$(\varphi, S, \sigma_1, \sigma_2, \tau, \kappa)$.

1 lit $\leftarrow (\sigma_1 \setminus \sigma_2)[0]$ /* Choose first literal lit s.t. lit $\in \sigma_1$, and lit $\notin \sigma_2$ */
2 $\varphi' = \varphi \wedge (\sigma_1 \vee \sigma_2)$
3 $(\psi, \hat{\tau}) \leftarrow$ GenHard(τ, κ)
4 $\hat{\varphi} \leftarrow \varphi' \wedge (\text{lit} \rightarrow \psi)$
5 $\hat{\varphi} \leftarrow \hat{\varphi} \wedge (\neg\text{lit} \rightarrow$ ConstructChain$(\hat{\tau}, Supp(\psi)))$
6 $\hat{S} \leftarrow S \cup Supp(\hat{\varphi})$
7 **return** $(\hat{\varphi}, \hat{S})$.

Algorithm 2 GenHard(τ, κ).

1 $(m, f) \leftarrow$ Compute(τ) /* Compute m, f such that $m, f \geq 0, m - f = \lceil \log \tau \rceil$ */
2 $M \leftarrow_r \{0,1\}^{512-m}$
3 $F \leftarrow_r \{0,1\}^{f}$
4 $\psi \leftarrow$ NossumFormulaGen(κ, m, M, f, F)
5 $\hat{\tau} \leftarrow 0$
6 **for** $value \in \{0,1\}^m$ **do**
7 **if** $h_\kappa(M + value)[1:f] = F$ /* h_κ is a hash-function, $h_\kappa \in \mathcal{H}^{\kappa}_{\texttt{SHA-1}}$ */
8 **then**
9 $\hat{\tau} \leftarrow \hat{\tau} + 1$
10 **return** $\psi, \hat{\tau}$

GenHard first computes the value of m and f by calling subroutine Compute. On line 2 GenHard generates a random string M of length $512 - m$ over $\{0,1\}$ from all possible sets of such strings. Similarly, on line 3, GenHard generates a string F of length f over $\{0,1\}$ randomly from all possible such strings. On line 4, GenHard calls NossumFormulaGen subroutine that returns a formula ψ. Finally, to calculate the exact number of satisfying assignments of ψ, on lines 6-9, GenHard iterates over all possible strings, denoted as *value*, of $\{0,1\}$ of size m. If first f bits of $h_\kappa(M + value)$ matches with F, then the count of $\hat{\tau}$ is increased by 1. At the end, GenHard returns the formula ψ and $\hat{\tau}$.

The algorithmic description of ScalBarbarik is almost identical to the Barbarik except for a notable difference of replacement of Kernel subroutine with Shakuni and the argument of hardRange. For completeness, we now provide the detailed algorithmic description of ScalBarbarik in Algorithm 3. Note that the expressions for t_j, β_j, N_j in Algorithm 3 have been revised after fixing minor errors in [3].

Algorithm 3 represents ScalBarbarik. ScalBarbarik has three loops, the outermost loop, lines 2-19 varies the computational hardness parameter κ as per the given range. The second loop, lines 3-19 makes $\log(\frac{4}{2\varepsilon + \eta})$ many rounds. And, in each round, first ScalBarbarik on line 9 computes the number of satisfying assignments, called N_j to be sampled from SUT. The inner loop of ScalBarbarik iterates t_j many times, which is computed in each round on line 4. In the inner loop, ScalBarbarik first samples a satisfying assignment σ_1 from the ideal distribution using the base sampler on line 12, and then it samples a satisfying assignment σ_2 from the SUT on line 13. Then, on line 14 ScalBarbarik calls subroutine Shakuni with formula φ, sampling set S, σ_1, and σ_2,N_j and κ. Subroutine Shakuni returns a new formula $\hat{\varphi}$ and a sampling set $\hat{S}$. On line 15, ScalBarbarik asks the SUT to sample N_j many satisfying assignments of $\hat{\varphi}$, which is stored in list L_3. On line 16, ScalBarbarik calls subroutine $Bias$.

Algorithm 3 ScalBarbarik($\mathcal{G}$, $\mathcal{U}$, S, ε, η, δ, φ, hardRange).

```
1  S ← Supp(φ)
2  for κ ∈ hardRange do
3      for j ← 1 to ⌈log(4/(2ε+η))⌉ do
           /* constants required to compute the # of samples */
4          t_j ← ⌈2^j (η+2ε)/(η−2ε)^2 log(4(2ε+η)^{-1})(4e/(e−1)) ln(δ^{-1})⌉
5          β_j ← ((2^{j−1}+1)(2ε+η)) / (4+(2ε+η)(2^{j−1}−1))
6          BoundFactor ← log( (2^4 e)/(e−1) · δ^{-1}/(η−2ε)^2 log(4/(2ε+η)) ln(1/δ) )⌉
7          γ ← (β_j − 2 × ε)/4
8          ConstantFactor ← ⌈ 1/(8.79 × γ × γ) ⌉
9          N_j ← ⌈(ConstantFactor × BoundFactor )⌉
10         for i ← 1 to t_j do
11             while L_1 = L_2 do
12                 L_1 ← G(φ, S, 1); σ_1 ← L_1[0]      /* G samples σ_1 from Sol(φ) */
13                 L_2 ← U(φ, S, 1); σ_2 ← L_2[0]      /* U samples σ_2 from Sol(φ) */
14             (φ̂, Ŝ) ← Shakuni(φ, S, σ_1, σ_2, N_j, κ)
15             L_3 ← G(φ̂, S, N_j)        /* G samples N_j solutions from Sol(φ̂) */
16             b ← Bias(σ_1, L_3, S)
17             if b < 1/2 (1 − c_j) or b > 1/2 (1 + c_j) then
18                 return REJECT
19     return ACCEPT
```

The subroutine *Bias* takes σ_1, L_3, and S as input and returns the cardinality of intersection of the σ_1 and L_3 over the sampling set S. The returned cardinality from *Bias* is stored in b. Finally, ScalBarbarik checks if the value of b is either lower than the low threshold or higher than the high threshold on line 17. If that is the case, ScalBarbarik rejects the SUT on line 18, otherwise, it continues with the inner loop on line 10.

4.1 Theoretical Analysis

First we need to prove the correctness of GenHard. From the code of GenHard (also, refer to Section 3.3) the following theorem follows:

▶ **Theorem 6.** *If* GenHard(τ, κ) *returns* $\psi, \hat{\tau}$ *then* $|R_\psi| = \hat{\tau}$, *where* $\hat{\tau} \geq \tau$.

Now, the correctness of Shakuni is almost identical to that of Kernel from [3]. We can prove the following theorem:

▶ **Theorem 7.** *If* $\hat{\varphi}$ *is the output of* Shakuni$(\varphi, S, \sigma_1, \sigma_2, \tau, \kappa)$ *then* $R_{\hat{\varphi}}$ *can be written as a disjoint union of two sets* Z_1 *and* Z_2 *such that for* $|Z_1| = |Z_2|$ *and for all* $\sigma \in Z_1$, $\sigma|_S = \sigma_1$ *and for all* $\sigma \in Z_2$, $\sigma|_S = \sigma_2$.

Proof. On line 2 it is ensured that φ' has only two satisfying assignments – namely σ_1 and σ_2. From Theorem 6 we see that GenHard (on line 3) returns a formula $(\psi, \hat{\tau})$ where $R_\psi = \hat{\tau}$ and at the same time ConstructChain (on line 5) returns a formula $\hat{\psi}$ with $R_{\hat{\psi}} = \hat{\tau}$ and

$Supp(\psi) = Supp(\hat{\psi})$. Thus by the construction of $\hat{\varphi}$ on lines 4 and 5, if σ is a satisfying assignment of $\hat{\varphi}$ then firstly $\sigma|_S$ is either σ_1 or σ_2. Also if $\sigma|_S$ is σ_1 then $\sigma|_{Supp(\hat{\varphi}\setminus S}$ is a satisfying assignment of ψ. Moreover, there is a one-to-one correspondence between the satisfying assignments of $\hat{\varphi}$, that satisfy $\sigma|_S = \sigma_1$, with R_ψ. Similarly, if $\sigma|_S$ is σ_2 then $\sigma|_{Supp(\hat{\varphi}\setminus S}$ is a satisfying assignment of $\hat{\psi}$. and there is a one-to-one correspondence between the satisfying assignments of $\hat{\varphi}$, that satisfy $\sigma|_S = \sigma_2$, with $R_{\psi'}$. Thus we have the theorem. ◀

Given the correctness of Shakuni, we observe that the theoretical analysis and query complexity of ScalBarbarik are almost identical to that of Barbarik from [3]. That is, if SUT $\mathcal{G}$ is ε-additive close to the uniform sampler then with probability $(1-\delta)$, ScalBarbarik outputs Accept. If the SUT is η far from uniform and it abides by the *subquery-consistent assumption*, ScalBarbarik outputs Reject with probability $(1-\delta)$. In case ScalBarbarik outputs Reject for sampler $\mathcal{G}$ on input φ, the assignments σ_1 and σ_2 can be seen as a certificate because the sampler $\mathcal{G}$ samples them with significantly different probabilities. Therefore, the output of ScalBarbarik is a list of tuples of the values of κ and the corresponding output.

5 Experimental Evaluation

To analyze the behavior of ScalBarbarik, we built a prototype implementation in Python and performed empirical evaluation on the 50 benchmarks that were used for the evaluation of Barbarik so as to situate our results with prior context [3]. For our evaluation, we used SPUR [1] as a base uniform sampler.

Test Hardware. All our experiments were conducted on a high-performance computing cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit of 4GB/core.

Test Parameters. We considered tolerance parameter ϵ, intolerance parameter η, and confidence δ to be 0.2, 1.6, and 0.1, respectively for experimentation evaluation using ScalBarbarik. For our chosen parameters, the number of samples required to return Accept for a given SUT is 2.173×10^3. We considered the following hardness parameters for ScalBarbarik: $\kappa = 10$, 11, 12, and 13. In the implementation of GenHard, we used $m = 14, f = 4$.

Samplers Tested. We performed empirical evaluation with four state-of-the-art samplers, QuickSampler [8], STS [9] CMSGen [13], and UniGen3 [23]. Of these, STS, QuickSampler, and CMSGen cannot provide theoretical guarantees on the distribution of samples generated, whereas UniGen provides guarantees. Furthermore, we experimented with different restart intervals for CMSGen. We set the parameter restart intervals to 300 and 500, that is, restarts at every 300 or 500 conflicts. We used CMSGen_{300} and CMSGen_{500} to refer to our prototype of CMSGen, respectively. The default version of CMSGen restarts at 100 conflicts.

The objective of our experimental evaluation is to analyze the impact of different computational hardness levels on the ability of ScalBarbarik to distinguish between state-of-the-art samplers. Furthermore, we seek to use ScalBarbarik to establish the balance between scalability and uniformity in order to tune the sampler to the application at hand. Towards this, we analyses the impact of different restart intervals of CMSGen on the quality of samples generated through ScalBarbarik. In particular, we seek to answer the following questions:

(1) Can ScalBarbarik distinguish between distributions generated by the various state-of-the-samplers? (2) Can we use ScalBarbarik to design a sampler that can balance scalability and uniformity?

Summary of results. In summary, we observe that ScalBarbarik Rejects STS and QuickSampler, and returns Accept for UniGen for all the benchmarks. Moreover, as we increase the hardness parameter for ScalBarbarik, it could Reject CMSGen. These experiments show that the quality of distribution for the samples generated by CMSGen is between the distribution generated by samplers without guarantees such as STS and QuickSampler, and by samplers with guarantees, such as UniGen3. Furthermore, with the help of ScalBarbarik, we are able to show that the quality of distribution generated improves with increased restart intervals for CMSGen, however, it takes more time to generate the samples.

5.1 Performance of ScalBarbarik

The first column of Table 1 shows the value of κ and rest of the table consists of two columns for each of the samplers. The columns with Accept/Reject represent the number of instances for which ScalBarbarik outputs Accept or Reject, respectively.

Table 1 Analysis of different samplers with ScalBarbarik over 50 benchmarks. Parameters used were $\epsilon : 0.2, \eta : 1.6, \delta : 0.1$, and samples required to output Accept: 2.173×10^3.

ScalBarbarik	QuickSampler		STS		CMSGen		UniGen3	
(κ)	Accept	Reject	Accept	Reject	Accept	Reject	Accept	Reject
10	0	50	0	50	50	0	50	0
11	0	50	0	50	41	9	50	0
12	0	50	0	50	19	31	50	0
13	0	50	0	50	0	50	50	0

Note that for $\kappa = 10$, ScalBarbarik outputs Accept on all instances for CMSGen, whereas it Rejects QuickSampler and STS. Upon increasing the value of κ to 11 and 12, ScalBarbarik outputs Reject on 9 and 31 instances, respectively. Finally, ScalBarbarik outputs Reject for CMSGen on all 50 instances with $\kappa = 13$. On the other hand, ScalBarbarik outputs Accept for all values of κ on all instances for UniGen3.

It is worth emphasizing that in comparison to Barbarik, ScalBarbarik returns a fine-grained analysis of the quality of distributions generated by the given sampler. Such a fine-grained analysis allows one to observe that the quality of distributions generated by CMSGen lie between QuickSampler, STS and UniGen3.

5.2 Achieving Balance between Scalability and Uniformity

Based on the discussion in Section 3.1, we can hypothesize that the quality of samples produced increase with an increase in restart interval for SAT solver based sampler such as CMSGen. To put our hypothesis to test, and to understand the behavior of CMSGen with different restart intervals, we performed evaluation using ScalBarbarik on CMSGen, CMSGen_{300}, and CMSGen_{500}. To provide a prospective, we also considered a sampler with theoretical guarantees, UniGen. We set the computation hardness parameter $\kappa = 11$, 15, 18, and 22. In Table 2, we list the number of instances for which ScalBarbarik returned Accept and Reject corresponding to the aforementioned samplers.

Table 2 # of benchmarks for which CMSGen, CMSGen$_{300}$, CMSGen$_{500}$, and UniGen3 are Accepted or Rejected by ScalBarbarik. Total of 50 benchmarks. Parameters $\epsilon : 0.2, \eta : 1.6, \delta : 0.1$, and samples required to return Accept 2.173×10^3. The default version of CMSGen used restart at 100 conflicts.

ScalBarbarik (κ)	CMSGen		CMSGen$_{300}$		CMSGen$_{500}$		UniGen3	
	Accept	Reject	Accept	Reject	Accept	Reject	Accept	Reject
11	41	9	47	3	47	3	50	0
15	0	50	37	13	42	8	50	0
18	0	50	0	50	36	14	50	0
22	0	50	0	50	0	50	50	0

Table 3 CMSGen, CMSGen$_{300}$, CMSGen$_{500}$ with ScalBarbarik with different hardness parameters.

(κ)	Benchmarks	CMSGen		CMSGen$_{300}$		CMSGen$_{500}$	
		Result	Samples	Result	Samples	Result	Samples
15	GuidanceService	Reject	742	Accept	2.173×10^3	Accept	2.173×10^3
	70.sk-310	Reject	265	Accept	2.173×10^3	Accept	2.173×10^3
	BlastedSpring24	Reject	318	Accept	2.173×10^3	Accept	2.173×10^3
	ActivityService	Reject	106	Reject	848	Accept	2.173×10^3
	IterationService	Reject	265	Reject	742	Reject	1.802×10^3
18	GuidanceService	Reject	159	Reject	265	Accept	2.173×10^3
	70.sk-310	Reject	53	Reject	848	Accept	2.173×10^3
	BlastedSpring24	Reject	159	Reject	742	Reject	849
	ActivityService	Reject	106	Reject	689	Accept	2.173×10^3
	IterationService	Reject	53	Reject	265	Reject	1.961×10^3

We observe that ScalBarbarik needs to increase the computation hardness in order to Reject CMSGen$_{500}$ for all the benchmarks – it Rejects CMSGen, CMSGen$_{300}$, and CMSGen$_{500}$ at κ values 13, 18, and 22 respectively.

Table 3 presents the result of ScalBarbarik with κ set to 15 and 18 over a subset of representative benchmarks. The first column in Table 3 presents the hardness parameter κ used with ScalBarbarik. The second column has the benchmarks details and the following columns indicate the outcome of ScalBarbarik for samplers CMSGen, CMSGen$_{300}$ and CMSGen$_{500}$. There are two columns for each of the samplers: (i) the first column shows whether the sampler is accepted by ScalBarbarik as a uniform sampler, and (ii) the second column shows the number of samples required by ScalBarbarik to decide Accept/Reject. Table 3 shows that ScalBarbarik needs less samples to reject CMSGen as compared to CMSGen$_{300}$ and CMSGen$_{500}$. Furthermore, as the hardness parameter κ is increased, ScalBarbarik rejects more instances with less number of samples for all three SUTs.

The results in Table 2 and Table 3 strongly support that as we increase the restart intervals, the distribution of samples generated are more likely to be uniform.

At this point, one may wonder whether there are costs associated with the improved quality of sampling in terms of runtime efficiency. To this end, we conducted a study of runtimes over 70 benchmarks used in prior studies [13]. We present the runtime comparison of CMSGen, CMSGen$_{300}$, and CMSGen$_{500}$ to generate 1000 samples in Figure 1. To put the runtimes in perspective, we also plot the curve corresponding to UniGen3. Figure 1 represents a cactus plot – a point $\langle x, y \rangle$ represents that a sampler took less than or equal to y seconds to sample 1000 satisfying assignments for x many benchmarks. With a timeout of 7200 seconds, CMSGen, CMSGen$_{300}$, CMSGen$_{500}$, were all able to generate 1000 samples for 52 benchmarks, and we see a significant increase in the runtime for those instances with CMSGen$_{500}$ and CMSGen$_{300}$ as compared to CMSGen.

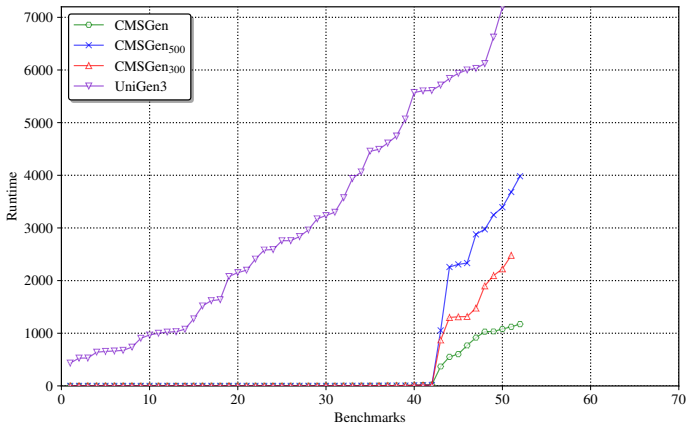

Figure 1 Cactusplot showing runtime performance of CMSGen, CMSGen$_{300}$, CMSGen$_{500}$, and UniGen3 to generate 1000 samples within a timeout of 7200s.

The gain of uniformity at the loss of runtime efficiency in the case of CMSGen$_{500}$ illustrates the trade-off between uniformity and runtime performance, and highlights opportunities for design of large number of samplers based on the needs of the underlying applications. While ideally, one would perform in-depth theoretical analysis to characterize the distribution generated by different samplers, modern CDCL solvers have not been shown to be amenable to such analysis. In this regard, having access to test frameworks such as ScalBarbarik to test uniformity is crucial.

6 Conclusion

Uniform sampling is a fundamental problem in computer science with widespread applications. This variety of applications has led to the design of many samplers with varying theoretical guarantees. There exists many uniform-like samplers that do not provide any guarantees on the distribution from which the samples are generated. The existence of such samplers led to the design of the first tester, Barbarik to test whether the distribution generated is ε-close or η-far from the uniform distribution. Barbarik was used in a test-driven development manner to create a uniform-like sampler CMSGen that cannot provide theoretical guarantees on the sampling distribution but is accepted as a ε-close uniform sampler by Barbarik.

The development of such a sampler led us to improve the testing framework Barbarik. In this work, we propose the sampler tester ScalBarbarik that provides quantitative answers to measure the quality of samplers, that is, it provides a hardness dial to achieve a fine-grained analysis of quality of samples. We showed that that the quality of samples generated by CMSGen are better than the other state-of-the-art samples such as STS and QuickSampler that do not provide theoretical guarantee; however, it is not as good as the samplers that provide guarantees on the distribution generated, such as UniGen3. Furthermore, the availability of ScalBarbarik can be used to achieve a balance between scalability and uniformity of samplers. We hope the demonstration of virtuosity of the cycle between testing and design will encourage other developers to design their own samplers while using ScalBarbarik as the underlying testing engine.

References

1 Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In *Proc. of SAT*, 2018.

2 Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In *Proc. of Pragmatics of SAT 2015*, 2018.

3 Sourav Chakraborty and Kuldeep S. Meel. On testing of uniform samplers. In *Proc. of AAAI*, 2019.

4 Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Proc. of AAAI*, 2015.

5 Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable and nearly uniform generator of sat witnesses. In *Proc. of CAV*, 2013.

6 Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, 2014.

7 Lori A Clarke. A program testing system. In *Proc. of ACM*, 1976.

8 Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of SAT solutions for testing. In *Proc. of ICSE*, 2018.

9 Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. In *Proc. of UAI*, 2012.

10 Vijay Ganesh, Charles W O'donnell, Mate Soos, Srinivas Devadas, Martin C Rinard, and Armando Solar-Lezama. Lynx: A programmatic SAT solver for the RNA-folding problem. In *Proc. of SAT*, 2012.

11 Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

12 Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data-driven approach for Boolean function synthesis. In *Proc. of CAV*, 2020.

13 Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing samplers is easy: The boon of testers. In *Proc. of FMCAD*, 2021.

14 Evgeny A. Grechnikov. Collisions for 72-step and 73-step SHA-1: improvements in the method of characteristics. *Proc. of IACR*, 2010.

15 James C King. Symbolic execution and program testing. *Comm. ACM*, 1976.

16 James H Kukula and Thomas R Shiple. Building circuits from relations. In *Proc. of CAV*, 2000.

17 S. Kullback and R. A. Leibler. On information and sufficiency. *Proc. of Ann. Math. Statist.*, 1951.

18 Kuldeep S. Meel, Yash Pote, and Sourav Chakraborty. On testing of samplers. In *Proc. of NeurIPS*, 2020.

19 Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *Proc. of AI magazine*, 2007.

20 Vegard Nossum. SAT-based preimage attacks on SHA-1. Master's thesis, University of Oslo, 2012. URL: `https://www.duo.uio.no/handle/10852/34912`.

21 Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proc. of ESEC/FSE*, 2018.

22 Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. Knowledge compilation meets uniform sampling. In *Proc. of LPAR*, 2018.

23 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*, 2020.

24 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, 2009.