Balancing Scalability and Uniformity in SAT Witness Generator [*]

Supratik Chakraborty
Indian Institute of Technology, Bombay
supratik@cse.iitb.ac.in

Kuldeep S. Meel, Moshe Y. Vardi
Rice University
kuldeep@rice.edu,vardi@cs.rice.edu

ABSTRACT

Constrained-random simulation is the predominant approach used in the industry for functional verification of complex digital designs. The effectiveness of this approach depends on two key factors: the quality of constraints used to generate test vectors, and the randomness of solutions generated from a given set of constraints. In this paper, we focus on the second problem, and present an algorithm that significantly improves the state-of-the-art of (almost-)uniform generation of solutions of large Boolean constraints. Our algorithm provides strong theoretical guarantees on the uniformity of generated solutions and scales to problems involving hundreds of thousands of variables.

1. INTRODUCTION

Functional verification constitutes one of the most challenging and time-consuming steps in the design of modern digital systems. The primary objective of functional verification is to expose design bugs early in the design cycle. Among various techniques available for this purpose, those based on simulation overwhelmingly dominate industrial practice. In a typical simulation-based functional verification exercise, a gate-level or RTL model of the circuit is simulated for a large number of cycles with specific input patterns. The values at observable outputs, as computed by the simulator, are then compared against their expected values, and any discrepancy is flagged as manifestaton of a bug. The state of simulation technology today is mature enough to allow simulation of large designs within reasonable time using modest computational resources. Generating input patterns that exercise diverse corners of the design's behavior space, however, remains a challenging problem [4].

In recent years, constrained-random simulation (also called constrained-random verification, or CRV) [21] has emerged as a practical approach to address the problem of simulating designs with "random enough" input patterns. In CRV, the verification engineer declaratively specifies a set of constraints on the values of circuit inputs. Typically, these constraints are obtained from usage requirements, environmental constraints, constraints on operating conditions and the like. A constraint solver is then used to generate random values for the circuit inputs satisfying the constraints. Since the distribution of errors in the design's behavior space is not known *a priori*, every solution to the set of constraints is as likely to discover a bug as any other solution. It is therefore important to sample the space of all solutions uniformly or almost-uniformly (defined formally below) at random. Unfortunately, guaranteeing uniformity poses significant technical challenges when scaling to large problem sizes. This has been repeatedly noted in the literature (see, for example, [8, 22, 16]) and also confirmed by industry practitioners[1]. The difficulties of generating solutions with guarantees of uniformity have even prompted researchers to propose alternative techniques for generating input patterns [8, 22]. This paper takes a step towards remedying this situation. Specifically, we describe an algorithm for generating solutions to a set of Boolean constraints, with stronger guarantees on uniformity and with higher scalability in practice than that achieved earlier.

Since constraints that arise in CRV of digital circuits are encodable as Boolean formulae, we focus on uniform generation of solutions of Boolean formulae. Henceforth, we call such solutions SAT *witnesses*. Besides its usefulness in CRV and in other applications [2, 23], uniform generation of SAT witnesses has had strong theoretical interest as well [14]. Most prior approaches to solving this problem belong to one of two categories: those that focus on strong guarantees of uniformity but scale poorly in practice (examples being [27, 3, 14]), and those that provide practical heuristics to scale to large problem instances with weak or no guarantees of uniformity (examples being [7, 16, 25])). In [5], Chakraborty, Meel and Vardi attempted to bridge these extremes through an algorithm called UniWit. More recently, Ermon, Gomes, Sabharwal and Selman [9] proposed an algorithm called PAWS for sampling witnesses from discrete distributions over large dimensions. While PAWS is designed to work with any discrete distribution specified through a graphical model, for purposes of this paper, we focus only on distributions that assign equal weight to every assign-

[*] This work was supported in part by NSF grants CNS 1049862 and CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by BSF grant 9800096, by gift from Intel, by a grant from Board of Research in Nuclear Sciences, India, and by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc.

DAC '14, June 01-05 2014, San Francisco, CA, USA

ACM 978-1-4503-2370-5/14/06$15.00.
http://dx.doi.org/10.1145/2593069.2593097.

[1] Private communication: R. Kurshan

ment. For such distributions, both PAWS and UniWit represent alternative (albeit related) approaches to solve the same problem – that of uniform generation of SAT witnesses. Unfortunately, both algorithms suffer from inherent limitations that make it difficult to scale them to Boolean constraints with tens of thousands of variables and beyond. In addition, the guarantees provided by these algorithms (in the context of uniform generation of SAT witnesses) are weaker than what one would desire in practice.

In this paper, we propose an algorithm called UniGen that addresses some of the deficiencies of UniWit and PAWS. This enables us to improve both the theoretical guarantees *and* practical performance vis-a-vis earlier algorithms in the context of uniform generation of SAT witnesses. UniGen is the first algorithm to provide strong two-sided guarantees of almost-uniformity, while scaling to problems involving hundreds of thousands of variables. We also improve upon the success probability of the earlier algorithms significantly, both in theory and as evidenced by our experiments.

2. NOTATION AND PRELIMINARIES

Let F be a Boolean formula in conjunctive normal form (CNF), and let X be the set of variables appearing in F. The set X is called the *support* of F. A *satisfying assignment* or *witness* of F is an assignment of truth values to variables in its support such that F evaluates to true. We denote the set of all witnesses of F as R_F. Let $\mathcal{D} \subseteq X$ be a subset of the support such that there are no two satisfying assignments of F that differ only in the truth values of variables in $\mathcal{D}$. In other words, in every satisfying assignment of F, the truth values of variables in $X \setminus \mathcal{D}$ uniquely determine the truth value of every variable in $\mathcal{D}$. The set $\mathcal{D}$ is called a *dependent* support of F, and $X \setminus \mathcal{D}$ is called an *independent* support of F. Note that there may be more than one independent supports of F. For example, $(a \vee \neg b) \wedge (\neg a \vee b)$ has three independent supports: $\{a\}$, $\{b\}$ and $\{a, b\}$. Clearly, if $\mathcal{I}$ is an independent support of F, so is every superset of $\mathcal{I}$. For notational convenience, whenever the formula F is clear from the context, we omit mentioning it.

We use $\mathsf{Pr}\left[X : \mathcal{P}\right]$ to denote the probability of outcome X when sampling from a probability space $\mathcal{P}$. For notational clarity, we omit $\mathcal{P}$ when it is clear from the context. The expected value of the outcome X is denoted $\mathsf{E}\left[X\right]$. Given a Boolean formula F, a *probabilistic generator* of witnesses of F is a probabilistic algorithm that generates a random witness in R_F. A *uniform generator* $\mathcal{G}^u(\cdot)$ is a probabilistic generator that guarantees $\mathsf{Pr}\left[\mathcal{G}^u(F) = y\right] = 1/|R_F|$, for every $y \in R_F$. An *almost-uniform generator* $\mathcal{G}^{au}(\cdot, \cdot)$ ensures that for every $y \in R_F$, we have $\frac{1}{(1+\varepsilon)|R_F|} \leq \mathsf{Pr}\left[\mathcal{G}^{au}(F, \varepsilon) = y\right] \leq \frac{1+\varepsilon}{|R_F|}$, where $\varepsilon > 0$ is the specified *tolerance*. A *near-uniform generator* $\mathcal{G}^{nu}(\cdot)$ further relaxes the guarantee of uniformity, and ensures that $\mathsf{Pr}\left[\mathcal{G}^{nu}(F) = y\right] \geq c/|R_F|$ for a constant c, where $0 < c \leq 1$. Probabilistic generators are allowed to occasionally "fail" in the sense that no witness may be returned even if R_F is non-empty. The failure probability for such generators must be bounded by a constant strictly less than 1. The algorithm presented in this paper falls in the category of almost-uniform generators. An idea closely related to that of almost-uniform generation, and used in a key manner in our algorithm, is *approximate model counting*. Given a CNF formula F, an *exact model counter* returns the size of R_F. An *approximate model counter* ApproxMC$(\cdot, \cdot, \cdot)$ relaxes this requirement to some extent. Given a CNF formula F, a tolerance $\varepsilon > 0$ and a confidence $1-\delta \in (0, 1]$, and approximate model counter ensures that $\mathsf{Pr}[\frac{|R_F|}{1+\varepsilon} \leq \mathsf{ApproxMC}(F, \varepsilon, 1-\delta) \leq (1+\varepsilon)|R_F|] \geq 1-\delta$.

A special class of hash functions, called *r-wise independent* hash functions, play a crucial role in our work. Let n, m and r be positive integers, and let $H(n, m, r)$ denote a family of r-wise independent hash functions mapping $\{0, 1\}^n$ to $\{0, 1\}^m$. We use $h \xleftarrow{R} H(n, m, r)$ to denote the probability space obtained by choosing a hash function h uniformly at random from $H(n, m, r)$. The property of r-wise independence guarantees that for all $\alpha_1, \ldots \alpha_r \in \{0, 1\}^m$ and for all distinct $y_1, \ldots y_r \in \{0, 1\}^n$, $\mathsf{Pr}\left[\bigwedge_{i=1}^{r} h(y_i) = \alpha_i : h \xleftarrow{R} H(n, m, r)\right] = 2^{-mr}$. For every $\alpha \in \{0, 1\}^m$ and $h \in H(n, m, r)$, let $h^{-1}(\alpha)$ denote the set $\{y \in \{0, 1\}^n \mid h(y) = \alpha\}$. Given $R_F \subseteq \{0, 1\}^n$ and $h \in H(n, m, r)$, we use $R_{F,h,\alpha}$ to denote the set $R_F \cap h^{-1}(\alpha)$. If we keep h fixed and let α range over $\{0, 1\}^m$, the sets $R_{F,h,\alpha}$ form a partition of R_F. For every $m \in \{1, \ldots |S| - 1\}$, the m^{th} prefix-slice of h, denoted $h^{(m)}$, is a map from $\{0, 1\}^{|S|}$ to $\{0, 1\}^m$, such that $h^{(m)}(y)[i] = h(y)[i]$, for all $y \in \{0, 1\}^{|S|}$ and for all $i \in \{1, \ldots m\}$. Similarly, the m^{th} prefix-slice of α, denoted $\alpha^{(m)}$, is an element of $\{0, 1\}^m$ such that $\alpha^{(m)}[i] = \alpha[i]$ for all $i \in \{1, \ldots m\}$.

3. RELATED WORK

Marrying scalability with strong guarantees of uniformity has been the holy grail of algorithms that sample from solutions of constraint systems. The literature bears testimony to the significant tension between these objectives when designing random generators of SAT witnesses. Earlier work in this area either provide strong theoretical guarantees at the cost of scalability, or remedy the scalability problem at the cost of guarantees of uniformity. More recently, however, there have been efforts to bridge these two extremes.

Bellare, Goldreich and Petrank [3] showed that a provably uniform generator of SAT witnesses can be designed in theory to run in probabilistic polynomial time relative to an NP oracle. Unfortunately, it was shown in [5] that this algorithm does not scale beyond formulae with few tens of variables in practice. Weighted binary decision diagrams (BDD) have been used in [27] to sample uniformly from SAT witnesses. However, BDD-based techniques are known to suffer from scalability problems [16]. Adapted BDD-based techniques with improved performance were proposed in [18]; however, the scalability was achieved at the cost of guarantees of uniformity. Random seeding of DPLL SAT solvers [20] has been shown to offer performance, although the generated distributions of witnesses can be highly skewed [16].

Markov Chain Monte Carlo methods (also called MCMC methods) [16, 26] are widely considered to be a practical way to sample from a distribution of solutions. Several MCMC algorithms, such as those based on simulated annealing, Metropolis-Hastings algorithm and the like, have been studied extensively in the literature [15, 19]. While MCMC methods guarantee eventual convergence to a target distribution under mild requirements, convergence is often impractically slow in practice. The work of [26, 16] proposed several such adaptations for MCMC-based sampling in the context of constrained-random verification. Unfortunately, most of these adaptations are heuristic in nature, and do not preserve theoretical guarantees of uniformity. constraints, thereby increasing constraint-solving time. Sampling techniques based on interval-propagation and belief networks have been proposed in [7, 10, 13]. The simplicity of these approaches lend scalability to the techniques, but the generated distributions can deviate significantly from the uniform distribution, as shown in [17].

Sampling techniques based on hashing were originally pioneered by Sipser [24], and have been used subsequently by several researchers [3, 11, 5]. The core idea in hashing-based sampling is to use r-wise independent hash functions (for a suitable value of r) to randomly partition the space of witnesses into "small cells" of roughly equal size, and then randomly pick a solution from a randomly chosen cell. The algorithm of Bellare et al. referred to above uses this idea with n-wise independent algebraic hash functions (where n denotes the size of the support of F). As noted above, their algorithm scales very poorly in practice. Gomes, Sabharwal and Selman used 3-wise independent linear hash functions in [11] to design XORSample′, a near-uniform generator of SAT witnesses. Nevertheless, to realize the guarantee of near-uniformity, their algorithm requires the user to provide difficult-to-estimate input parameters. Although XORSample′ has been shown to scale to constraints involving a few thousand variables, Gomes et al. acknowledge the difficulty of scaling their algorithm to much larger problem sizes without sacrificing theoretical guarantees [11].

Recently, Chakraborty, Meel and Vardi [5] proposed a new hashing-based SAT witness generator, called UniWit, that represents a small but significant step towards marrying the conflicting goals of scalability and guarantees of uniformity. Like XORSample′, the UniWit algorithm uses 3-wise independent linear hashing functions. Unlike XORSample′, however, the guarantee of near-uniformity of witnesses generated by UniWit does not depend on difficult-to-estimate input parameters. In [5], UniWit has been shown to scale to formulas with several thousand variables. In addition, Chakraborty et al proposed a heuristic called "leap-frogging" that allows UniWit to scale even further – to tens of thousands of variables [5]. Unfortunately, the guarantees of near-uniformity can no longer be established for UniWit with "leap-frogging". More recently, Ermon et al. [9] proposed a hashing-based algorithm called PAWS for sampling from a distribution defined over a discrete set using a graphical model. While the algorithm presented in this paper has some similarities with PAWS, there are significant differences as well. Specifically, our algorithm provides much stronger theoretical guarantees vis-a-vis those offered by PAWS in the context of uniform generation of SAT witness. In addition, our algorithm scales to hundreds of thousands of variables while preserving the theoretical guarantees. PAWS faces the same scalability hurdles as UniWit, and is unlikely to scale beyond a few thousand variables without heuristic adapatations that compromise its guarantees.

4. THE UNIGEN ALGORITHM

The new algorithm, called UniGen, falls in the category of hashing-based almost-uniform generators. UniGen shares some features with earlier hashing-based algorithms such as XORSample′ [11], UniWit [5] and PAWS [9], but there are key differences that allow UniGen to significantly outperform these earlier algorithms, both in terms of theoretical guarantees and measured performance.

Given a CNF formula F, we use a family of 3-independent hash functions to randomly partition the set, R_F, of witnesses of F. Let $h : \{0,1\}^n \rightarrow \{0,1\}^m$ be a hash function in the family, and let y be a vector in $\{0,1\}^n$. Let $h(y)[i]$ denote the i^{th} component of the vector obtained by applying h to y. The family of hash functions of interest is defined as $\{h(y) \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \cdot y[k]), a_{i,j} \in \{0,1\}, 1 \leq i \leq m, 0 \leq j \leq n\}$, where $\oplus$ denotes the xor operation. By choosing values of $a_{i,j}$ randomly and independently, we can effectively choose a random hash function from the family. It has been shown in [11] that this family of hash functions is 3-independent. Following notation introduced in Section 2, we call this family $H_{xor}(n, m, 3)$.

While $H_{xor}(n, m, 3)$ was used earlier in XORSample′, PAWS, and (in a variant of) UniWit, there is a fundamental difference in the way we use it in UniGen. Let $X = \{x_1, x_2, \ldots x_{|X|}\}$ be the set of variables of F. Given $m > 0$, the algorithms XORSample′, PAWS and UniWit partition R_F by randomly choosing $h \in H_{xor}(|X|, m, 3)$ and $\alpha \in \{0,1\}^m$, and by seeking witnesses of F conjoined with $\bigwedge_{i=1}^{m} \big(h(x_1, \ldots x_{|X|})[i] \leftrightarrow \alpha[i]\big)$. By choosing a random $h(x_1, \ldots x_{|X|}) \in H_{xor}(|X|, m, 3)$, the set of *all* assignments to variables in X (regardless of whether they are witnesses of F) is partitioned randomly. This, in turn, ensures that the set of satisfying assignments of F is also partitioned randomly. Each conjunctive constraint of the form $(h(x_1 \ldots x_{|X|})[i] \leftrightarrow \alpha[i])$ is an xor of a subset of variables of X and $\alpha[i]$, and is called an *xor-clause*. Observe that the expected number of variables in each such xor-clause is approximately $|X|/2$. It is well-known (see, for example [12]) that the difficulty of checking satisfiability of a CNF formula with xor-clauses grows significantly with the number of variables per xor-clause. It is therefore extremely difficult to scale XORSample′, PAWS or UniWit to problems involving hundreds of thousands of variables. In [5], an alternative family of linear hash functions is proposed to be used with UniWit. Unfortunately, this also uses $|X|/2$ variables per xor-clause on average, and suffers from the same problem. In [12], a variant of $H_{xor}(|X|, m, 3)$ is used, wherein each variable in X is chosen to be in an xor-clause with a small probability q (< 0.5). This mitigates the performace bottleneck significantly, but theoretical guarantees of (near-)uniformity are lost.

We address the above problem in UniGen by making two important observations: (i) an independent support $\mathcal{I}$ of F is often far smaller (sometimes by a few orders of magnitude) than X, and (ii) since the value of every variable in $X \setminus \mathcal{I}$ in a satisfying assignment of F is uniquely determined by the values of variables in $\mathcal{I}$, the set R_F can be randomly partitioned by randomly partitioning its projection on $\mathcal{I}$. This motivates us to design an almost-uniform generator that accepts a subset S of the support of F as an additional input. We call S the set of *sampling variables* of F, and intend to use an independent support of F (not necessarily a minimal one) as the value of S in any invocation of the generator. Without loss of generality, let $S = \{x_1, \ldots x_{|S|}\}$, where $|S| \leq |X|$. The set R_F can now be partitioned by randomly choosing $h \in H_{xor}(|S|, m, 3)$ and $\alpha \in \{0,1\}^m$, and by seeking solutions of $F \wedge \bigwedge_{i=1}^{m} \big(h(x_1, \ldots x_{|S|})[i] \leftrightarrow \alpha[i]\big)$. If $|S| \ll |X|$ (as is often the case in our experience), the expected number of variables per xor-clause is significantly reduced. This makes satisfiability checking easier, and allows scaling to much larger problem sizes than otherwise possible. It is natural to ask if finding an independent support of a CNF formula F is computationally easy. While an algorithmic solution to this problem is beyond the scope of this paper, our experience indicates that a small, not necessarily minimal, independent support can often be easily determined from the source domain from which the CNF formula F is derived. For example, when a non-CNF formula G is converted to an equisatisfiable CNF formula F using Tseitin encoding, the variables introduced by the encoding form a dependent support of F.

The effectiveness of a hashing-based probabilistic generator depends on its ability to quickly partition the set R_F into "small" and "roughly equal" sized random cells. This, in turn, depends on the parameter m used in the choice of the hash function family $H(n, m, r)$. A high value of m leads to skewed distributions of sizes of cells, while a low value of m leads to cells that are not small enough. The best choice of m depends on $|R_F|$, which is not known *a priori*. Different algorithms therefore use different techniques to estimate a value of m. In XORSample′, this is achieved by requiring the user to provide some difficult-to-estimate input parameters. In UniWit, the algorithm sequentially iterates over values of m until a good enough value is found. The approach of PAWS comes closest to our, although there are crucial differences. In both PAWS and UniGen, an approximate model counter is first used to estimate $|R_F|$ within a specified tolerance and with a specified confidence. This estimate, along with a user-provided parameter, is then used to determine a *unique* value of m in PAWS. Unfortunately, this does not facilitate proving that PAWS is an almost-uniform generator. Instead, Ermon, et al. show that PAWS behaves like an almost-uniform generator with probability greater than $1 - \delta$, for a suitable δ that depends on difficult-to-estimate input parameters. In contrast, we use the estimate of $|R_F|$ to determine a *small range* of candidate values of m. This allows us to prove that UniGen is almost-uniform generator with confidence 1.

Algorithm 1 UniGen(F, ε, S)

```
/*Assume $S = \{x_1, \dots x_{|S|}\}$ is an independent support of $F$,
    and $\varepsilon > 1.71$ */
 1: $(\kappa, \mathrm{pivot}) \leftarrow$ ComputeKappaPivot($\varepsilon$);
 2: $\mathrm{hiThresh} \leftarrow 1 + (1 + \kappa)\mathrm{pivot}$;
 3: $\mathrm{loThresh} \leftarrow \frac{1}{1+\kappa}\mathrm{pivot}$;
 4: $Y \leftarrow$ BSAT($F$, hiThresh);
 5: if ($|Y| \leq$ hiThresh) then
 6:     Let $y_1, \dots y_{|Y|}$ be the elements of $Y$;
 7:     Choose $j$ at random from $\{1, \dots |Y|\}$; return $y_j$;
 8: else
 9:     $C \leftarrow$ ApproxModelCounter($F$, 0.8, 0.8);
10:     $q \leftarrow \lceil \log C + \log 1.8 - \log \mathrm{pivot} \rceil$;
11:     $i \leftarrow q - 4$;
12:     Choose $h$ at random from $H_{xor}(|S|, n, 3)$;
13:     Choose $\alpha$ at random from $\{0, 1\}^n$;
14:     repeat
15:         $i \leftarrow i + 1$;
16:         $Y \leftarrow$ BSAT($F \wedge (h_i(x_1, \dots x_{|S|}) = \alpha_i)$, hiThresh);
17:     until (loThresh $\leq |Y| \leq$ hiThresh) or ($i = q$)
18:     if ($|Y| >$ hiThresh) or ($|Y| <$ loThresh) then
19:         return $\perp$
20:     else
21:         Let $y_1, \dots y_{|Y|}$ be the elements of $Y$;
22:         Choose $j$ at random from $[|Y|]$ and return $y_j$;
```

Algorithm 2 ComputeKappaPivot($t\varepsilon$)

```
Find $\kappa \in [0, 1)$ such that $\varepsilon = (1 + \kappa)(2.23 + \frac{0.48}{(1-\kappa)^2}) - 1$ ;
pivot $\leftarrow \lceil 3e^{1/2}(1 + \frac{1}{\kappa})^2 \rceil$;
return ($\kappa$, pivot)
```

The pseudocode for UniGen is shown in Algorithm 1. UniGen takes as inputs a Boolean CNF formula F, a tolerance ε (> 1.71, for teachnical reasons explained in the Appendix) and a set S of sampling variables. It either returns a random witness of F or $\perp$ (indicating failure). The algorithm assumes access to a source of random binary numbers, and to two subroutines: (i) BSAT(F, N), which, for every $N > 0$, returns $\min(|R_F|, N)$ distinct witnesses of F, and (ii) an approximate model counter ApproxModelCounter($F, \varepsilon', 1 - \delta'$).

UniGen first computes two quantities, "pivot" and κ, that represent the expected size of a "small" cell and the tolerance of this size, respectively. The specific choices of expressions used to compute κ and "pivot" in ComputeKappaPivot are motivated by technical reasons explained in the Appendix. The values of κ and "pivot" are used to determine high and low thresholds (denoted "hiThresh" and "loThresh" respectively) for the size of each cell. Lines 5–7 handle the easy case when F has no more than "hiThresh" witnesses. Otherwise, UniGen invokes ApproxModelCounter to obtain an estimate, C, of $|R_F|$ to within a tolerance of 0.8 and with a confidence of 0.8. Once again, the specific choices of the tolerance and confidence parameters used in computing C are motivated by technical reasons explained in the Appendix. The estimate C is then used to determine a range of candidate values for m. Specifically, this range is $\{q - 4, \dots q\}$, where q is determined in line 10 of the pseudocode. The loop in lines 12–17 checks whether some value in this range is good enough for m, i.e., whether the number of witnesses in a cell chosen randomly after partitioning R_F using $H_{xor}(|S|, m, 3)$, lies within "hiThresh" and "loThresh". If so, lines 21–22 return a random witness from the chosen cell. Otherwise, the algorithm reports a failure in line 19.

An probabilistic generator is likely to be invoked multiple times with the same input constraint in constrained-random verification. Towards this end, note than lines 1–11 of the pseudocode need to executed only once for every formula F. Generating a new random witness requires executing afresh only lines 12–22. While this optimization appears similar to "leapfrogging" [5, 6], it is fundamentally different since it does not sacrifice any theoretical guarantees, unlike "leapfrogging".

Implementation issues: In our implementation of UniGen, BSAT is implemented using CryptoMiniSAT [1] – a SAT solver that handles xor clauses efficiently. CryptoMiniSAT uses *blocking clauses* to prevent already generated witnesses from being generated again. Since the independent support of F determines every satisfying assignment of F, blocking clauses can be restricted to only variables in the set S. We implemented this optimization in CryptoMiniSAT, leading to significant improvements in performance. ApproxModelCounter is implemented using ApproxMC [6]. Although the authors of [6] used "leapfrogging" in their experiments, we disable this optimization since it nullifies the theoretical guarantees of [6]. We use "random_device" implemented in C++ as the source of pseudo-random numbers in lines 7, 14, 15 and 22 of the pseudocode, and also as the source of random numbers in ApproxMC.

Guarantees: The following theorem shows that UniGen is an almost-uniform generator with a high success probability.

Theorem 1. *If S is an independent support of F and if $\varepsilon > 1.71$, then for every $y \in R_F$, we have*

$$\frac{1}{(1+\varepsilon)(|R_F|-1)} \leq \Pr\left[\mathsf{UniGen}(F,\varepsilon,S) = y\right] \leq (1+\varepsilon)\frac{1}{|R_F|-1}.$$

In addition, $\Pr\left[\mathsf{UniGen}(F,\varepsilon,S) \neq \bot\right] \geq 0.62$.

For lack of space, we defer the proof to the Appendix. It can be shown that UniGen runs in time polynomial in ε^{-1} and in the size of F, relative to an NP-oracle.

The guarantees provided by Theorem 1 are significantly stronger than those provided by earlier generators that scale to large problem instances. Specifically, neither XORSample′ [11] nor UniWit [5] provide strong upper bounds for the probability of generation of a witness. PAWS [9] offers a *probabilistic* guarantee that the probability of generation of a witness lies within a tolerance factor of the uniform probability, while the guarantee of Theorem 1 is not prbabilistic. The success probability of PAWS, like that of XORSample′, is bounded below by an expression that depends on difficult-to-estimate input parameters. Interestingly, the same parameters also directly affect the tolerance of distribution of the generated witnesses. The success probability of UniWit is bounded below by 0.125, which is significantly smaller than the lower bound of 0.62 guaranteed by Theorem 1.

Trading scalability with uniformity: The tolerance parameter ε provides a knob to balance scalability and uniformity in UniGen. Smaller values of ε lead to stronger guarantees of uniformity (by Theorem 1). Note, however, that the value of "hiThresh" increases with decreasing values of ε, requiring BSAT to find more witnesses. Thus, each invocation of BSAT is likely to take longer as ε is reduced.

5. EXPERIMENTAL RESULTS

To evaluate the performance of UniGen, we built a prototype implementation and conducted an extensive set of experiments. Industrial constrained-random verification problem instances are typically proprietary and unavailable for published research. Therefore, we conducted experiments on CNF SAT constraints arising from several problems available in the public-domain. These included bit-blasted versions of constraints arising in bounded model checking of circuits and used in [5], bit-blasted versions of SMTLib benchmarks, constraints arising from automated program synthesis, and constraints arising from ISCAS89 circuits with parity conditions on randomly chosen subsets of outputs and next-state variables.

To facilitate running multiple experiments in parallel, we used a high-performance cluster and ran each experiment on a node of the cluster. Each node had two quad-core Intel Xeon processors with 4 GB of main memory. Recalling the terminology used in the pseudocode of UniGen (see Section 4), we set the tolerance ε to 6, and the sampling set S to an independent support of F in all our experiments. Independent supports (not necessarily minimal ones) for all benchmarks were easily obtained from the providers of the benchmarks on request. We used 2,500 seconds as the timeout for each invocation of BSAT and 20 hours as the overall timeout for UniGen, for each problem instance. If an invocation of BSAT timed out in line 16 of the pseudocode of UniGen, we repeated the execution of lines 14–16 without incrementing i. With this set-up, UniGen was able to successfully generate random witnesses for formulas having up to 486,193 variables.

For performance comparisons, we also implemented and conducted experiments with UniWit – a state-of-art near-uniform generator [5]. Our choice of UniWit as a reference for comparison is motivated by several factors. First, UniGen and UniWit share some commonalities, and UniGen can be viewed as an improvement of UniWit. Second, XORSample′ is known to perform poorly vis-a-vis UniWit [5]; hence, comparing with XORSample′ is not meaningful. Third, the implementation of PAWS made available by the authors of [9] currently does not accept CNF formulae as inputs. It accepts only a graphical model of a discrete distribution as input, making a direct comparison with UniGen difficult. Since PAWS and UniWit share the same scalability problem related to large random xor-clauses, we chose to focus only on UniWit. Since the "leapfrogging" heuristic used in [5] nullifies the guarantees of UniWit, we disabled this optimization. For fairness of comparison, we used the same timeouts in UniWit as used in UniGen, i.e. 2,500 seconds for every invocation of BSAT, and 20 hours overall for every invocation of UniWit.

Table 1 presents the results of our performance-comparison experiments. Column 1 lists the CNF benchmark, and columns 2 and 3 give the count of variables and size of independent support used, respectively. The results of experiments with UniGen are presented in the next 3 columns. Column 4 gives the observed probability of success of UniGen when generating 1,000 random witnesses. Column 5 gives the average time taken by UniGen to generate one witness (averaged over a large number of runs), while column 6 gives the average number of variables per xor-clause used for randomly partitioning R_F. The next two columns give results of our experiments with UniWit. Column 7 lists the average time taken by UniWit to generate a random witness, and column 8 gives the average number of variables per xor-clause used to partition R_F. A "–" in any column means that the corresponding experiment failed to generate any witness in 20 hours.

It is clear from Table 1 that the average run-time for generating a random witness by UniWit can be two to three orders of magnitude larger than the corresponding run-time for UniGen. This is attributable to two reasons. The first stems from fewer variables in xor-clauses and blocking clauses when small independent supports are used. Benchmark "tutorial3" exemplifies this case. Here, UniWit failed to generate any witness because all calls to BSAT in UniWit, with xor-clauses and blocking clauses containing numbers of variables, timed out. In contrast, the calls to BSAT in UniGen took much less time, due to short xor-clauses and blocking clauses using only variables from the independent support. The other reason for UniGen's improved efficiency is that the computationally expensive step of identifying a a good range of values for m (see Section 4 for details) needs to be executed only once per benchmark. Subsequently, whenever a random witness is needed, UniGen simply iterates over this narrow range of m. In contrast, generating every witness in UniWit (without leapfrogging) requires sequentially searching over all values afresh to find a good choice for m. Referring to Table 1, UniWit requires more than 20,000 seconds on average to find a good value for m and generate a random witness for benchmark "s953a_3_2". Unlike in UniGen, there is no way to amortize this large time over multiple runs in UniWit, while preserving the guarantee of near-uniformity.

Table 1 also shows that the observed success probability of UniGen is almost always 1, much higher than what Theorem 1 guarantees and better than those from UniWit. It is clear from our experiments that UniGen can scale to problems involving almost 500K variables, while preserving guarantees of almost uniformity. This goes much beyond the reach of any other random-witness generator that gives strong guarantees on the distribution of witnesses.

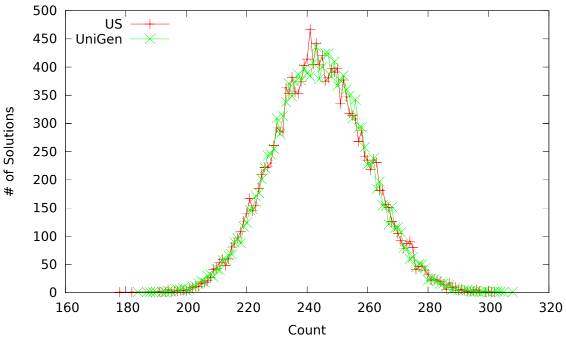

Figure 1: Uniformity comparison for case110

Theorem 1 guarantees that the probability of generation of every witness lies within a specified tolerance of the uniform probability. In practice, however, the distribution of witnesses generated by UniGen is much more closer to a uniform distribution. To illustrate this, we implemented a *uniform sampler*, henceforth called US, and compared the distributions of witnesses generated by UniGen and by US for some representative benchmarks. Given a CNF formula F, US first determines $|R_F|$ using an exact model counter (such as sharpSAT). To mimic generating a random witness, US simply generates a random number i in $\{1 \ldots |R_F|\}$. To ensure fair comparison, we used the same source of randomness in both UniGen and US. For every problem instance on which the comparison was done, we generated a large number N $(= 4 \times 10^6)$ of sample witnesses using each of US and UniGen. In each case, the number of times various witnesses were generated was recorded, yielding a distribution of the counts. Figure 1 shows the distributions of counts generated by UniGen and by US for one of our benchmarks (case110) with $16,384$ witnesses. The horizontal axis represents counts and the vertical axis represents the number of witnesses appearing a specified number of times. Thus, the point $(242, 450)$ represents the fact that each of 450 distinct witnesses were generated 242 times in 4×10^6 runs. Observe that the distributions resulting from UniGen and US can hardly be distinguished in practice. This holds not only for this benchmark, but for all other benchmarks we experimented with.

Overall, our experiments confirm that UniGen is two to three orders of magnitude more efficient than state-of-the-art random witness generators, has probability of success almost 1, and preserves strong guarantees about the uniformity of generated witnesses. Furthermore, the distribution of generated witnesses can hardly be distinguished from that of a uniform sampler in practice.

6. CONCLUSION

Striking a balance between scalability and uniformity is a difficult challenge when designing random witness generators for constrained-random verification. UniGen is the first such generator for Boolean CNF formulae that scales to hundreds of thousands of variables and still preserves strong guarantees of uniformity. In future, we wish to investigate the design of scalable generators with similar guarantees for SMT constraints, leveraging recent progress in satisfiability modulo theories.

Acknowledgments

We profusely thank Mate Soos for implementing required APIs in CryptoMiniSAT without which the experimentation section would have been incomplete. Mate was generous with his suggestions to improve our implementation. We thank Ajith John for his help in experimental setup.

7. REFERENCES

[1] CryptoMiniSAT.
`http://www.msoos.org/cryptominisat2/`.

[2] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. of FOCS*, pages 340–351, 2003.

[3] M. Bellare, O. Goldreich, and E. Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 1998.

[4] L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes.* Springer, 2001.

[5] S. Chakraborty, K. Meel, and M. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *Proc. of CAV*, 2013.

[6] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, 2013.

[7] R. Dechter, K. Kask, E. Bin, and R. Emek. Generating random solutions for constraint satisfaction problems. In *AAAI*, 2002.

[8] S. Deng, Z. Kong, J. Bian, and Y. Zhao. Self-adjusting constrained random stimulus generation using splitting evenness evaluation and xor constraints. In *Proc. of ASP-DAC*, pages 769–774. IEEE, 2009.

[9] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman. Embed and project: Discrete sampling with universal hashing. In *Proc. of NIPS*, 2013.

[10] V. Gogate and R. Dechter. A new algorithm for sampling csp solutions uniformly at random. In *CP*, pages 711–715, 2006.

[11] C. Gomes, A. Sabharwal, and B. Selman. Near uniform sampling of combinatorial spaces using XOR constraints. In *Proc. of NIPS*, pages 670–676, 2007.

[12] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. Short XORs for model counting; from theory to practice. In *SAT*, pages 100–106, 2007.

[13] M. A. Iyer. Race: A word-level atpg-based constraints solver system for smart random simulation. In *ITC*, pages 299–308. Citeseer, 2003.

[14] M. Jerrum, L. Valiant, and V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *TCS*, 43(2-3):169–188, 1986.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

Table 1: Runtime performance comparison of UniGen and UniWit

			UniGen			UniWit		
Benchmark	|X|	|S|	Succ Prob	Avg Run Time (s)	Avg XOR leng	Avg Run Time (s)	Avg XOR len	Succ Prob
Squaring7	1628	72	1.0	2.44	36	2937.5	813	0.87
squaring8	1101	72	1.0	1.77	36	5212.19	550	1.0
Squaring10	1099	72	1.0	1.83	36	4521.11	550	0.5
s1196a_7_4	708	32	1.0	6.9	16	833.1	353	0.37
s1238a_7_4	704	32	1.0	7.26	16	1570.27	352	0.35
s953a_3_2	515	45	0.99	12.48	23	22414.86	257	*
EnqueueSeqSK	16466	42	1.0	32.39	21	–	–	–
LoginService2	11511	36	0.98	6.14	18	–	–	–
LLReverse	63797	25	1.0	33.92	13	3460.58	31888	0.63
Sort	12125	52	0.99	79.44	26	–	–	–
Karatsuba	19594	41	1.0	85.64	21	–	–	–
tutorial3	486193	31	0.98	782.85	16	–	–	–

A "*" entry indicates insufficient data for estimating success probability

[16] N. Kitchen. *Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation*. PhD thesis, University of California, Berkeley, 2010.

[17] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *Proc. of ICCAD*, pages 258–265, 2007.

[18] J. H. Kukula and T. R. Shiple. Building circuits from relations. In *Proc. of CAV*, pages 113–123, 2000.

[19] N. Madras. Lectures on monte carlo methods, fields institute monographs 16. *AMS*, 2002.

[20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of DAC*, pages 530–535, 2001.

[21] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek. Constraint-based random stimuli generation for hardware verification. In *Proc of IAAI*, pages 1720–1727, 2006.

[22] S. M. Plaza, I. L. Markov, and V. Bertacco. Random stimulus generation using entropy and xor constraints. In *Proc. of DAC*, pages 664–669, 2008.

[23] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273–302, 1996.

[24] M. Sipser. A complexity theoretic approach to randomness. In *Proc. of STOC*, pages 330–335, 1983.

[25] W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proc. of AAAI*, pages 670–676, 2004.

[26] W. Wei and B. Selman. A new approach to model counting. In *Proc. of SAT*, pages 2293–2299, 2005.

[27] J. Yuan, A. Aziz, C. Pixley, and K. Albin. Simplifying boolean constraint solving for random simulation vector generation. *TCAD*, 23(3):412–420, 2004.

APPENDIX

In this section, we present a proof of Theorem 1, originally stated in Section 4, and also present an extended table of performance comparison results.

Recall that UniGen is a probabilistic algorithm that takes as inputs a Boolean CNF formula F, a tolerance ε and a subset S of the support of F. We first show that if X is the support of F, and if $S \subsetneq X$ is an independent support of F, then $\mathsf{UniGen}(F,\ \varepsilon,\ S)$ behaves *identically* (in a probabilistic sense) to $\mathsf{UniGen}(F,\ \varepsilon,\ X)$. Once this is established, the remainder of the proof proceeds by making the simplifying assumption $S = X$.

Clearly, the above claim holds trivially if $X = S$. Therefore, we focus only on the case when $S \subsetneq X$. For notational convenience, we assume $X = \{x_1, \ldots x_n\}$, $0 \le k < n$, $S = \{x_1, \ldots x_k\}$ and $D = \{x_{k+1}, \ldots x_n\}$ in all the statements and proofs in this section. We also use $\vec{X}$ to denote the vector $(x_1, \ldots x_n)$, and similarly for $\vec{S}$ and $\vec{D}$.

Lemma 1. *Let $F(\vec{X})$ be a Boolean function with support X, and let S be an independent support of F. Then there exist Boolean functions $g_0, g_1, \ldots g_{n-k}$, each with support S such that*

$$F(\vec{X}) \leftrightarrow \left(g_0(\vec{S}) \wedge \bigwedge_{j=1}^{n-k} (x_{k+j} \leftrightarrow g_j(\vec{S}))\right)$$

Proof. Since S is an independent support of F, we have $D = X \setminus S$ is a dependent support of F. From the definition of a dependent support, there exist Boolean functions $g_1, \ldots g_k$, each with support S, such that $F(\vec{X}) \rightarrow \bigwedge_{j=1}^{n-k} (x_{k+j} \leftrightarrow g_j(\vec{S}))$.

Let $g_0(\vec{S})$ be the characteristic function of the projection of R_F on S. More formally, $g_0(\vec{S}) \equiv \bigvee_{(x_{k+1}, \ldots x_n) \in \{0,1\}^{n-k}} F(\vec{X})$. It follows that $F(\vec{X}) \rightarrow g_0(\vec{S})$. Combining this with the result from the previous paragraph, we get the implication $F(\vec{X}) \rightarrow \left(g_0(\vec{S}) \wedge \bigwedge_{j=1}^{n-k} (x_{k+j} \leftrightarrow g_j(\vec{S}))\right)$

From the definition of $g_0(\vec{S})$ given above, we have $g_0(\vec{S}) \rightarrow F(\vec{S}, x_{k+1}, \ldots x_n)$, for some values of $x_{k+1}, \ldots x_n$. However, we also know that $F(\vec{X}) \rightarrow \bigwedge_{j=1}^{n-k} (x_{k+j} \leftrightarrow g_j(\vec{S}))$. It follows that $\left(g(\vec{S}) \wedge \bigwedge_{j=1}^{n-k} (x_{k+j} \leftrightarrow g_j(\vec{S}))\right) \rightarrow F(\vec{X})$. □

Referring to the pseudocode of UniGen in Section 4, we observe that the only steps that depend directly on S are those in line 14, where h is chosen randomly from $H_{xor}(|S|, i, 3)$, and line 16, where the set Y is computed by calling $\mathsf{BSAT}(F \wedge (h(x_1, \ldots x_{|S|}) = \alpha), \text{hiThresh})$. Since all subsequent steps of the algorithm depend only on Y, it suffices to show that if S is an independent support of F, the probability distribution of Y obtained at line 16 is *identical* to what we would obtain if S was set equal to the entire support, X, of F.

The following lemma formalizes the above statement. As before, we assume $X = \{x_1, \ldots x_n\}$ and $S = \{x_1, \ldots x_k\}$.

Lemma 2. *Let S be an independent support of $F(\vec{X})$. Let h and h' be hash functions chosen uniformly at random from $H_{xor}(k, i, 3)$ and $H_{xor}(n, i, 3)$, respectively. Let α and α' be tuples chosen uniformly at random from $\{0,1\}^i$. Then, for every $Y \in \{0,1\}^n$ and for every $t > 0$, we have*

$$\mathsf{Pr}\left[\mathsf{BSAT}\left(F(\vec{X}) \wedge (h(\vec{S}) = \alpha), t\right) = Y\right] = \mathsf{Pr}\left[\mathsf{BSAT}\left(F(\vec{X}) \wedge (h'(\vec{X}) = \alpha'), t\right) = Y\right]$$

Proof. Since h' is chosen uniformly at random from $H_{xor}(n, i, 3)$, recalling the definition of $H_{xor}(n, i, 3)$, we have $F(\vec{X}) \wedge (h'(\vec{X}) = \alpha') \equiv F(\vec{X}) \wedge \bigwedge_{l=1}^{i} \left((a_{l,0} \oplus \bigoplus_{j=1}^{n} a_{l,j} \cdot x[j]) \leftrightarrow \alpha'[l]\right)$, where the $a_{l,j}$s are chosen independently and identically randomly from $\{0,1\}$.

Since S is an independent support of F, from Lemma 1, there exist Boolean functions $g_1, \ldots g_{n-k}$, each with support S, such that $F(\vec{X}) \rightarrow \bigwedge_{j=1}^{n-k} (x_{k+j} \leftrightarrow g_j(\vec{S}))$. Therefore, $F(\vec{X}) \wedge (h'(\vec{X}) = \alpha')$ is semantically equivalent to $F(\vec{X}) \wedge \bigwedge_{l=1}^{i} \left((a_{l,0} \oplus \bigoplus_{j=1}^{k} a_{l,j} \cdot x[j] \oplus B) \leftrightarrow \alpha'[l]\right)$, where $B \equiv \bigoplus_{j=k+1}^{n} a_{l,j} \cdot g_{j-k}(\vec{S})$. Rearranging terms, we get $F(\vec{X}) \wedge \bigwedge_{l=1}^{i} \left((a_{l,0} \oplus \bigoplus_{j=1}^{k} a_{l,j} \cdot x[j]) \leftrightarrow (\alpha'[l] \oplus B)\right)$.

Since α' is chosen uniformly at random from $\{0,1\}^i$ and since B is independent of α', it is easy to see that $\alpha'[l] \oplus B$ is a random binary variable with equal probability of being 0 and 1. It follows that $\mathsf{Pr}\left[\mathsf{BSAT}(F(\vec{X}) \wedge (h'(\vec{X}) = \alpha'), t) = Y\right] = \mathsf{Pr}\left[\mathsf{BSAT}(F(\vec{X}) \wedge (h(\vec{S}) = \alpha), t) = Y\right]$. □

Lemma 2 allows us to continue with the remainder of the proof assuming $S = X$. It has already been shown in [11] that $H_{xor}(n, m, 3)$ is a 3-independent family of hash functions. We use this fact in a key way in the remainder of our analysis. The following result about Chernoff-Hoeffding bounds, proved in [6], plays an important role in our discussion.

Theorem 1. *Let Γ be the sum of r-wise independent random variables, each of which is confined to the interval $[0,1]$, and suppose $\mathsf{E}[\Gamma] = \mu$. For $0 < \beta \le 1$, if $2 \le r \le \left\lfloor \beta^2 \mu e^{-1/2} \right\rfloor \le 4$, then $\mathsf{Pr}\left[\,|\Gamma - \mu| \ge \beta\mu\,\right] \le e^{-r/2}$.*

Using notation introduced in Section 2, let R_F denote the set of witnesses of the Boolean formula F. For convenience of analysis, we assume that $\log(|R_F| - 1) - \log \mathit{pivot}$ is an integer, where *pivot* is the quantity computed by algorithm $\mathsf{ComputeKappaPivot}$ (see Section 4). A more careful analysis removes this assumption by scaling the probabilities by constant factors. Let us denote $\log(|R_F| - 1) - \log \mathit{pivot}$ by m. The expression used for computing *pivot* in algorithm $\mathsf{ComputeKappaPivot}$ ensures that pivot ≥ 17. Therefore, if an invocation of UniGen does not return from line 7 of the pseudocode, then $|R_F| \ge 18$. Note also that the expression for computing κ in algorithm $\mathsf{ComputeKappaPivot}$ requires $\varepsilon \ge 1.71$ in order to ensure that $\kappa \in [0,1)$ can always be found.

The following lemma shows that q, computed in line 10 of the pseudocode, is a good estimator of m.

Lemma 3. $\mathsf{Pr}[q - 3 \le m \le q] \ge 0.8$

Proof. Recall that in line 9 of the pseudocode, an approximate model counter is invoked to obtain an estimate, C, of $|R_F|$ with tolerance 0.8 and confidence 0.8. By the definition of approximate model counting, we have $\mathsf{Pr}[\frac{C}{1.8} \le |R_F| \le (1.8)C] \ge 0.8$. Thus, $\mathsf{Pr}[\log C - \log(1.8) \le \log |R_F| \le \log C + \log(1.8)] \ge 0.8$. It follows that $\mathsf{Pr}[\log C - \log(1.8) -$

$\log pivot - \log(\frac{1}{1-1/|R_F|}) \leq \log(|R_F|-1) - \log pivot \leq \log C - \log pivot + \log(1.8) - \log(\frac{1}{1-1/|R_F|})] \geq 0.8$. Substituting $q = \lceil \log C + \log 1.8 - \log pivot \rceil$, $m = \log(|R_F|-1) - \log pivot$, $log(1.8) = 0.85$ and $\log(\frac{1}{1-1/|R_F|}) \leq 0.12$ (since $|R_F| \geq 18$ on reaching line 10 of the pseudocode), we get $\mathsf{Pr}[q-3 \leq m \leq q] \geq 0.8$. □

The next lemma provides a lower bound on the probability of generation of a witness. Let $w_{i,y,\alpha}$ denote the probability $\mathsf{Pr}\left[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)\text{pivot and } h(y) = \alpha : h \xleftarrow{R} H_{xor}(n,i,3)\right]$. The proof of the lemma also provides a lower bound on $w_{m,y,\alpha}$.

Lemma 4. *For every witness y of F, $\mathsf{Pr}[y \text{ is output}] \geq \frac{0.8(1-e^{-1})}{(1.06+\kappa)(|R_F|-1)}$*

Proof. If $|R_F| \leq 1 + (1+\kappa)\text{pivot}$, the lemma holds trivially (see lines 5–7 of the pseudocode). Suppose $|R_F| \geq 1 + (1+\kappa)\text{pivot}$ and let U denote the event that witness $y \in R_F$ is output by UniGen on inputs F, ε and X. Let $p_{i,y}$ denote the probability that we return from line 17 for a particular value of i with y in $R_{F,h,\alpha}$, where $\alpha \in \{0,1\}^i$ is the value chosen in line 15. Then, $\mathsf{Pr}[U] = \cup_{i=q-3}^{q} \frac{1}{|Y|} p_{i,y}$, where Y is the set of witnesses returned by BSAT in line 16 of the pseudocode. Let $f_m = \mathsf{Pr}[q-3 \leq m \leq q]$. From Lemma 3, we know that $f_m \geq 0.8$. From the design of the algorithm, we also know that $\frac{1}{1+\kappa}\text{pivot} \leq |Y| \leq 1 + (1+\kappa)\text{pivot}$. Therefore, $\mathsf{Pr}[U] \geq \frac{1}{1+(1+\kappa)\text{pivot}} \cdot p_{m,y} \cdot f_m$. The proof is now completed by showing $p_{m,y} \geq \frac{1}{2^m}(1-e^{-1})$. This gives $\mathsf{Pr}[U] \geq \frac{0.8(1-e^{-1})}{(1+(1+\kappa)pivot)2^m} \geq \frac{0.8(1-e^{-1})}{(1.06+\kappa)(|R_F|-1)}$. The last inequality uses the observation that $1/\text{pivot} \leq 0.06$.

To calculate $p_{m,y}$, we first note that since $y \in R_F$, the requirement "$y \in R_{F,h,\alpha}$" reduces to "$y \in h^{-1}(\alpha)$". For $\alpha \in \{0,1\}^n$, we define $w_{m,y,\alpha}$ as $\mathsf{Pr}\left[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)\right.$ $\left. pivot \text{ and } h(y) = \alpha : h \xleftarrow{R} H_{xor}(n,m,3)\right]$. Therefore, $p_{m,y} = \Sigma_{\alpha \in \{0,1\}^m}\left(w_{m,y,\alpha}.2^{-m}\right)$. The proof is now completed by showing that $w_{m,y,\alpha} \geq (1-e^{-1})/2^m$ for every $\alpha \in \{0,1\}^m$ and $y \in \{0,1\}^n$.

Towards this end, let us first fix a random y. Now we define an indicator variable $\gamma_{z,\alpha}$ for every $z \in R_F \setminus \{y\}$ such that $\gamma_{z,\alpha} = 1$ if $h(z) = \alpha$, and $\gamma_{z,\alpha} = 0$ otherwise. Let us fix α and choose h uniformly at random from $H_{xor}(n,m,3)$. The random choice of h induces a probability distribution on $\gamma_{z,\alpha}$ such that $E[\gamma_{z,\alpha}] = \mathsf{Pr}[\gamma_{z,\alpha} = 1] = 2^{-m}$. Since we have fixed y, and since hash functions chosen from $H_{xor}(n,m,3)$ are 3-wise independent, it follows that for every distinct $z_a, z_b \in R_F \setminus \{y\}$, the random variables $\gamma_{z_a,\alpha}, \gamma_{z_b,\alpha}$ are 2-wise independent. Let $\Gamma_\alpha = \sum_{z \in R_F \setminus \{y\}} \gamma_{z,\alpha}$ and $\mu_\alpha = E[\Gamma_\alpha]$. Clearly, $\Gamma_\alpha = |R_{F,h,\alpha}| - 1$ and $\mu_\alpha = \sum_{z \in R_F \setminus \{y\}} E[\gamma_{z,\alpha}] = \frac{|R_F|-1}{2^m}$. Also, $\mathsf{Pr}[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)\text{pivot}] = \mathsf{Pr}[\frac{\text{pivot}}{1+\kappa} - 1 \leq |R_{F,h,\alpha}| - 1 \leq (1+\kappa)\text{pivot}] \geq \mathsf{Pr}[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| - 1 \leq (1+\kappa)\text{pivot}]$. Using the expression for pivot, we get $2 \leq \lfloor e^{-1/2}(1+1/\epsilon)^2 \cdot \frac{|R_F|-1}{2^m} \rfloor$. Therefore using Theorem 1 and substituting $\text{pivot} = (|R_F|-1)/2^m$, we get $\mathsf{Pr}[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| - 1 \leq (1+\kappa)\text{pivot}] \geq 1 - e^{-1}$. Therefore, $\mathsf{Pr}[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)\text{pivot}] \geq 1 - e^{-1}$ Since h is chosen at random from $H_{xor}(n,m,3)$, we also have $\mathsf{Pr}[h(y) = \alpha] = 1/2^m$. It follows that $w_{m,y,\alpha} \geq (1-e^{-1})/2^m$. □

The next lemma provides an upper bound of $w_{i,y,\alpha}$ and $p_{i,y}$.

Lemma 5. *For $i < m$, both $w_{i,y,\alpha}$ and $p_{i,y}$ are bounded above by $\frac{1}{|R_F|-1} \frac{1}{\left(1-\frac{1+\kappa}{2^{m-i}}\right)^2}$.*

Proof. We will use the terminology introduced in the proof of Lemma 4. Clearly, $\mu_\alpha = \frac{|R_F|-1}{2^i}$. Since each $\gamma_{z,\alpha}$ is a 0-1 variable, $\mathsf{V}[\gamma_{z,\alpha}] \leq \mathsf{E}[\gamma_{z,\alpha}]$. Therefore, $\sigma^2_{z,\alpha} \leq \sum_{z \neq y, z \in R_F} \mathsf{E}[\gamma_{z,\alpha}]$ $\leq \sum_{z \in R_F} \mathsf{E}[\gamma_{z,\alpha}] = \mathsf{E}[\Gamma_\alpha] = 2^{-m}(|R_F|-1)$. So $\mathsf{Pr}[\frac{pivot}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)\text{pivot}] \leq \mathsf{Pr}[|R_{F,h,\alpha}| - 1 \leq (1+\kappa)\text{pivot}]$. From Chebyshev's inequality, we know that $\mathsf{Pr}[|\Gamma_\alpha - \mu_{z,\alpha}| \geq \kappa\sigma_{z,\alpha}] \leq 1/\kappa^2$ for every $\kappa > 0$. By choosing $\kappa = (1 - \frac{1+\kappa}{2^{m-i}})\frac{\mu_{z,\alpha}}{\sigma_{z,\alpha}}$, we have $\mathsf{Pr}[|R_{F,h,\alpha}| - 1 \leq (1+\kappa)\text{pivot}] \leq \mathsf{Pr}$ $\left[|(|R_{F,h,\alpha}| - 1) - \frac{|R_F|-1}{2^i}| \geq (1 - \frac{1+\kappa}{2^{m-i}})\frac{|R_F|-1}{2^i}\right] \leq \frac{1}{\left(1-\frac{(1+\kappa)}{2^{m-i}}\right)^2}$. $\frac{2^i}{|R_F|-1}$. Since h is chosen at random from $H_{xor}(n,m,3)$, we also have $\mathsf{Pr}[h(y) = \alpha] = 1/2^i$. It follows that $w_{i,y,\alpha} \leq \frac{1}{|R_F|-1}\frac{1}{\left(1-\frac{1+\kappa}{2^{m-i}}\right)^2}$. The bound for $p_{i,y}$ is easily obtained by noting that $p_{i,y} = \Sigma_{\alpha \in \{0,1\}^i}\left(w_{i,y,\alpha}.2^{-i}\right)$. □

Lemma 6. *For every witness y of F, $\mathsf{Pr}[y \text{ is output}] \leq \frac{1+\kappa}{|R_F|-1}(2.23 + \frac{0.48}{(1-\kappa)^2})$*

Proof. We will use the terminology introduced in the proof of Lemma 4. $\mathsf{Pr}[U] = \cup_{i=q-3}^{q} \frac{1}{|Y|} p_{i,y} \leq \frac{1+\kappa}{\text{pivot}} \sum_{i=q-3}^{q} p_{i,y}$. We can sub-divide the calculation of $\mathsf{Pr}[U]$ into three cases based on the range of the values m can take.

Case 1 : $q - 3 \leq m \leq q$.

Now there are four values that m can take.

1. $m = q-3$. We know that $p_{i,y} \leq \mathsf{Pr}[h(y) = \alpha] = \frac{1}{2^i}$. $\mathsf{Pr}[U|m = q-3] \leq \frac{1+\kappa}{\text{pivot}} \cdot \frac{1}{2^{q-3}}\frac{15}{8}$. Substituting the value of pivot and m, we get $\mathsf{Pr}[U|m = q-3] \leq \frac{15(1+\kappa)}{8(|R_F|-1)}$.
2. $m = q-2$. For $i \in [q-2, q]$ $p_{i,y} \leq \mathsf{Pr}[h(y) = \alpha] = \frac{1}{2^i}$ Using Lemma 5, we get $p_{q-3,y} \leq \frac{1}{|R_F|-1}\frac{1}{\left(1-\frac{1+\kappa}{2}\right)^2}$. Therefore, $\mathsf{Pr}[U|m = q-2] \leq \frac{1+\kappa}{\text{pivot}}\frac{1}{|R_F|-1}(\frac{1}{1-\frac{1+\kappa}{2}}) + \frac{1+\kappa}{\text{pivot}}\frac{1}{2^{q-2}}\frac{7}{4}$. Noting that $\text{pivot} = \frac{|R_F|-1}{2^m} > 10$, $\mathsf{Pr}[U|m = q-2] \leq \frac{1+\kappa}{|R_F|-1}(\frac{7}{4} + \frac{0.4}{(1-\kappa)^2})$
3. $m = q-1$. For $i \in [q-1, q]$, $p_{i,y} \leq \mathsf{Pr}[h(y) = \alpha] = \frac{1}{2^i}$. Using Lemma 5, we get $p_{q-3,y} + p_{q-2,y} \leq \frac{1}{|R_F|-1}\left(\frac{1}{\left(1-\frac{1+\kappa}{2^2}\right)} + \frac{1}{\left(1-\frac{1+\kappa}{2}\right)}\right.$ Therefore, $\mathsf{Pr}[U|m = q-1] \leq \frac{1+\kappa}{\text{pivot}}\left(\frac{1}{|R_F|-1}\left(\frac{1}{\left(1-\frac{1+\kappa}{2^2}\right)^2} + \frac{1}{\left(1-\frac{1+\kappa}{2}\right)^2}\right)\right.$ Noting that $\text{pivot} = \frac{|R_F|-1}{2^m} > 10$ and $\kappa \leq 1$, $\mathsf{Pr}[U|m = q-1] \leq \frac{1+\kappa}{|R_F|-1}(1.9 + \frac{0.4}{(1-\kappa)^2})$.
4. $m = q$, $p_{q,y} \leq \mathsf{Pr}[h(y) = \alpha] = \frac{1}{2^q}$. Using Lemma 5, we get $p_{q-3,y} + p_{q-2,y} + p_{q-1,y} \leq \frac{1}{|R_F|-1}\left(\frac{1}{\left(1-\frac{1+\kappa}{2^3}\right)^2}\frac{1}{\left(1-\frac{1+\kappa}{2^2}\right)^2} + \frac{1}{\left(1-\frac{1+\kappa}{2}\right)^2}\right)$. Therefore, $\mathsf{Pr}[U|m = q] \leq \frac{1+\kappa}{\text{pivot}}\left(\frac{1}{|R_F|-1}\left(\frac{1}{\left(1-\frac{1+\kappa}{2^3}\right)^2} + \frac{1}{\left(1-\frac{1+\kappa}{2^2}\right)^2} + \frac{1}{\left(1-\frac{1+\kappa}{2}\right)^2}\right) + 1\right)$. Noting that $\text{pivot} = \frac{|R_F|-1}{2^m} > 10$, $\mathsf{Pr}[U|m = q] \leq \frac{1+\kappa}{|R_F|-1}(1.58 + \frac{0.4}{(1-\kappa)^2})$.

$\mathsf{Pr}[U|q-3 \leq m \leq q] \leq \max_i(\mathsf{Pr}[U|m=i])$. Therefore, $\mathsf{Pr}[U|q-3 \leq m \leq q] \leq \mathsf{Pr}[U|m=q-1] \leq \frac{1+\kappa}{|R_F|-1}(1.9 + \frac{0.4}{(1-\kappa)^2})$.

Case 2 : $m < q-3$. $\mathsf{Pr}[U|m < q-3] \leq \frac{1+\kappa}{\text{pivot}} \cdot \frac{1}{2^{q-3}}\frac{15}{8}$. Substituting the value of pivot and maximizing $m-q+3$, we get $\mathsf{Pr}[U|m<q-3] \leq \frac{15(1+\kappa)}{16(|R_F|-1)}$.

Case 3 : $m > q$. Using Lemma 5, we know that $\mathsf{Pr}[U|m > q] \leq \frac{1+\kappa}{|R_F|-1}\frac{2^m}{|R_F|-1} \sum_{i=q-3}^{q} \frac{1}{1-\frac{1+\kappa}{2^{m-i}}}$. The R.H.S. is maximized when $m = q+1$. Hence $\mathsf{Pr}[U|m > q] \leq \frac{1+\kappa}{|R_F|-1} \frac{2^m}{|R_F|-1}\sum_{i=q-3}^{q} \frac{1}{1-\frac{1+\kappa}{2^{q+1-i}}}$. Noting that pivot $= \frac{|R_F|-1}{2^m} > 10$ and expanding the above summation $\mathsf{Pr}[U|m > q] \leq \frac{1+\kappa}{|R_F|-1}\frac{1}{10}\left(\frac{1}{(1-\frac{1+\kappa}{2^4})^2} + \frac{1}{(1-\frac{1+\kappa}{2^3})^2} + \frac{1}{(1-\frac{1+\kappa}{2^2})^2} + \frac{1}{(1-\frac{1+\kappa}{2^1})^2}\right)$. Using $\kappa \leq 1$ for the first two summation terms, $\mathsf{Pr}[U|m > q] \leq \frac{1+\kappa}{|R_F|-1} \cdot \frac{1}{10} \cdot (7.1 + \frac{4}{(1-\kappa)^2})$

Summing up all the above cases, $\mathsf{Pr}[U] = \mathsf{Pr}[U|m < q-3] \times \mathsf{Pr}[m < q-3] + \mathsf{Pr}[U|q-3 \leq m \leq q] \times \mathsf{Pr}[q-3 \leq m \leq q] + \mathsf{Pr}[U|m > q] \times \mathsf{Pr}[m > q]$. Using $\mathsf{Pr}[m < q-1] \leq 0.2$, $\mathsf{Pr}[m > q] \leq 0.2$ and $\mathsf{Pr}[q-3 \leq m \leq q] \leq 1$. Therefore, $\mathsf{Pr}[U] \leq \frac{1+\kappa}{|R_F|-1}(2.23 + \frac{0.48}{(1-\kappa)^2})$

□

Combining Lemma 4 and 6, the following theorem is obtained.

Theorem 2. *For every witness y of F, if $\varepsilon > 1.71$,*

$$\frac{1}{(1+\varepsilon)(|R_F|-1)} \leq \mathsf{Pr}\left[\mathsf{UniGen}(F,\varepsilon,X) = y\right] \leq (1+\varepsilon)\frac{1}{|R_F|-1}.$$

Proof. The proof is completed by using Lemmas 4 and 6 and substituting $(1+\varepsilon) = (1+\kappa)(2.23 + \frac{0.48}{(1-\kappa)^2})$. To arrive at the results, we use the inequality $\frac{1.06+\kappa}{0.8(1-e^{-1})} \leq (1+\kappa)(2.23 + \frac{0.48}{(1-\kappa)^2})$. □

Theorem 3. *Algorithm* UniGen *succeeds (i.e. does not return $\bot$) with probability at least* 0.62.

Proof. If $|R_F| \leq 1 + (1+\kappa)$pivot, the theorem holds trivially. Suppose $|R_F| > 1 + (1+\kappa)$pivot and let P_{succ} denote the probability that a run of the algorithm UniGen succeeds. Let p_i, such that $(q-3 \leq i \leq q)$ denote the conditional probability that UniGen (F, ε, X) terminates in iteration i of the repeat-until loop (line 11-16) with $\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)$pivot, given $|R_F| > 1 + (1+\kappa)$pivot. Therefore, $P_{\text{succ}} = \sum_{i=q-3}^{q} p_i \prod_{j=q-3}^{i}(1-p_j)$. Let $f_m = \mathsf{Pr}[q-3 \leq m \leq q]$. Therefore, $P_{\text{succ}} \geq p_m f_m \geq 0.8 p_m$. The theorem is now proved by using Theorem 1 to show that $p_m \geq 1 - e^{-3/2} \geq 0.77$.

For every $y \in \{0,1\}^n$ and for every $\alpha \in \{0,1\}^m$, define an indicator variable $\nu_{y,\alpha}$ as follows: $\nu_{y,\alpha} = 1$ if $h(y) = \alpha$, and $\nu_{y,\alpha} = 0$ otherwise. Let us fix α and y and choose h uniformly at random from $H_{xor}(n,m,3)$. The random choice of h induces a probability distribution on $\nu_{y,\alpha}$, such that $\mathsf{Pr}[\nu_{y,\alpha} = 1] = \mathsf{Pr}[h(y) = \alpha] = 2^{-m}$ and $\mathsf{E}[\nu_{y,\alpha}] = \mathsf{Pr}[\nu_{y,\alpha} = 1] = 2^{-m}$. In addition 3-wise independence of hash functions chosen from $H_{xor}(n,m,3)$ implies that for every distinct $y_a, y_b, y_c \in R_F$, the random variables $\nu_{y_a,\alpha}, \nu_{y_b,\alpha}$ and $\nu_{y_c,\alpha}$ are 3-wise independent.

Let $\Gamma_\alpha = \sum_{y \in R_F} \nu_{y,\alpha}$ and $\mu_\alpha = \mathsf{E}\left[\Gamma_\alpha\right]$. Clearly, $\Gamma_\alpha = |R_{F,h,\alpha}|$ and $\mu_\alpha = \sum_{y \in R_F} \mathsf{E}\left[\nu_{y,\alpha}\right] = 2^{-m}|R_F|$. Since $|R_F| >$ *pivot* and $i - l > 0$, using the expression for *pivot*, we get $3 \leq \left\lfloor e^{-1/2}(1+\frac{1}{\varepsilon})^{-2} \cdot \frac{|R_F|}{2^m} \right\rfloor$. Therefore, using Theorem 1, $\mathsf{Pr}\left[\frac{|R_F|}{2^m}.\left(1 - \frac{\kappa}{1+\kappa}\right) \leq |R_{F,h,\alpha}| \leq (1+\kappa)\frac{|R_F|}{2^m}\right] > 1 - e^{-3/2}$. Simplifying and noting that $\frac{\kappa}{1+\kappa} < \kappa$ for all $\kappa > 0$, we obtain $\mathsf{Pr}\left[(1+\kappa)^{-1} \cdot \frac{|R_F|}{2^m} \leq |R_{F,h,\alpha}| \leq (1+\kappa) \cdot \frac{|R_F|}{2^m}\right] > 1 - e^{-3/2}$. Also, $\frac{\text{pivot}}{1+\kappa} = \frac{1}{1+\kappa}\frac{|R_F|-1}{2^m} \leq \frac{|R_F|}{(1+\kappa)2^m}$ and $1 + (1+\kappa)\text{pivot} = 1 + \frac{(1+\kappa)(|R_F|-1)}{2^m} \geq \frac{(1+\kappa)|R_F|}{2^m}$. Therefore, $p_m = \mathsf{Pr}[\frac{\text{pivot}}{1+\kappa} \leq |R_{F,h,\alpha}| \leq 1 + (1+\kappa)\text{pivot}] \geq \mathsf{Pr}\left[(1+\kappa)^{-1} \cdot \frac{|R_F|}{2^m} \leq |R_{F,h,\alpha}| \leq (1+\kappa) \cdot \frac{|R_F|}{2^m}\right] \geq 1 - e^{-3/2}$. □

Table 2 presents an extended version of Table 1. We observe that UniGen is two to three orders of magnitude more efficient than state-of-the-art random witness generators, has probability of success almost 1 over a large set of benchmarks arising from different domains.

Table 2: Extended Table of Runtime performance comparison of UniGen and UniWit

			UniGen			UniWit	
Benchmark	#Variables	|S|	Succ Prob	Avg Run Time (s)	Avg XOR len	Avg Run Time (s)	Avg XOR len
Case121	291	48	1.0	0.19	24	56.09	145
Case1_b11_1	340	48	1.0	0.2	24	755.97	170
Case2_b12_2	827	45	1.0	0.33	22	–	–
Case35	400	46	0.99	11.23	23	666.14	199
Squaring1	891	72	1.0	0.38	36	–	–
Squaring8	1101	72	1.0	1.77	36	5212.19	550
Squaring10	1099	72	1.0	1.83	36	4521.11	550
Squaring7	1628	72	1.0	2.44	36	2937.5	813
Squaring9	1434	72	1.0	4.43	36	4054.42	718
Squaring14	1458	72	1.0	24.34	36	2697.42	728
Squaring12	1507	72	1.0	31.88	36	3421.83	752
Squaring16	1627	72	1.0	41.08	36	2852.17	812
s526_3_2	365	24	0.98	0.68	12	51.77	181
s526a_3_2	366	24	1.0	0.97	12	84.04	182
s526_15_7	452	24	0.99	1.68	12	23.04	225
s1196a_7_4	708	32	1.0	6.9	16	833.1	353
s1196a_3_2	690	32	1.0	7.12	16	451.03	345
s1238a_7_4	704	32	1.0	7.26	16	1570.27	352
s1238a_15_7	773	32	1.0	7.94	16	136.7	385
s1196a_15_7	777	32	0.97	8.98	16	133.45	388
s1238a_3_2	686	32	0.99	10.85	16	1416.28	342
s953a_3_2	515	45	0.99	12.48	23	22414.86	257
TreeMax	24859	19	1.0	0.52	10	49.78	12423
LLReverse	63797	25	1.0	33.92	13	3460.58	31888
LoginService2	11511	36	0.98	6.14	18	–	–
EnqueueSeqSK	16466	42	1.0	32.39	21	–	–
ProjectService3	3175	55	1.0	71.74	28	–	–
Sort	12125	52	0.99	79.44	26	–	–
Karatsuba	19594	41	1.0	85.64	21	–	–
ProcessBean	4768	64	0.98	123.52	32	–	–
tutorial3_4_31	486193	31	0.98	782.85	16	–	–