Circuit Testing Based on Fuzzy Sampling with BDD Bases

Elena Pinilla Sediles
Universidad Nacional de Educacion a Distancia,
Madrid Spain
epinilla23@alumno.uned.es

David Fernandez-Amoros
Universidad Nacional de Educacion a Distancia,
Madrid Spain
david@issi.uned.es

Ruben Heradio
Universidad Nacional de Educacion a Distancia,
Madrid Spain
rheradio@issi.uned.es

Abstract

Fuzzy testing of integrated circuits is an established technique. Current approaches generate an approximately uniform random sample from a translation of the circuit to Boolean logic. These approaches have serious scalability issues, which become more pressing with the ever-increasing size of circuits. We propose using a base of binary decision diagrams to sample the translations as a soft computing approach. Uniformity is guaranteed by design and scalability is greatly improved. We test our approach against five other state-of-the-art tools and find our tool to outperform all of them, both in terms of performance and scalability.

Keywords: SAT-sampling, Fuzzy Sampling, Integrated Circuits, Random Sampling, Binary Decision Diagrams.

1. Introduction

Testing is the most costly and time-consuming stage of integrated circuits (ICs) development. This is mainly caused by the complexity and the integration scale of the common ICs, which consist of millions of transistors. Accordingly, testing is a key phase for detecting defects and eliminating their causes, which guarantees the reliability and quality of the designed circuits, as well as compliance with their specifications.

There are many different testing techniques. The one covered in this paper is fuzzy testing (also known as fuzzing), which provides random inputs, or samples, to the ICs in order to evaluate their reliability. Fuzzing has a lot of advantages as it is almost completely automated, conceptually simple, does not require any knowledge about the system behaviour (black-box testing), and does not generate false positives. Additionally, it is a broadly used complementary testing technique, as it finds errors that other tools cannot Takanen (2009).

A number of tools to perform sampling of circuits encoded in Conjunctive Normal Form (CNF) have been developed in Achlioptas et al. (2018), Chakraborty and Meel (2019), Dutra et al. (2018), Oh et al. (2019), and Sharma et al. (2018). A sampling tool is said to be uniform if each value can be generated with the same probability (i.e., all the values are equally likely to appear in a sample). These tools claim to achieve a high degree of uniformity; some even claim to be perfectly uniform by design. Independent evaluation found these claims to be overly optimistic Heradio et al. (2022). Uniformity is also hard to achieve because of the underlying problem of counting the number of solutions of a CNF, known as #SAT or model counting.

In any case, computing time rapidly increases with the size of the circuits, resulting in scalability issues.

In this paper, we propose using a base of Binary Decision Diagrams (BDDs) to produce random values for ICs. Each gate is represented by a BDD, which helps alleviate the problems of building big BDDs. In contrast to generating a single BDD for the whole problem, this approach scales very well. The resulting BDD base is then used to produce random samples of the IC in a very efficient fashion. Using several BDDs instead of one generally means that uniformity is lost, which would not be a problem in fuzzy testing as it is a soft computing approach. Instead, we will show that our sampling algorithm is uniform by design in the particular case of IC sampling. The different approaches are then compared in a benchmark of sequential circuits that has been translated to CNF.

The rest of the paper is structured as follows: Section 2 presents related work to this topic. Section 3 explains the translation of the circuits to logic. Section 4 discusses how the sampling is performed. Section 5 shows the experimental results obtained, and lastly, section 6 gathers the conclusions and future work.

2. Related work

Translating circuits to logic for testing is an established technique, as most random samplers expect inputs to be in CNF. This is not an ideal solution for sequential circuits (i.e., circuits with memory) because CNF is stateless. Nevertheless, the general availability of analysis tools that use CNF as input has favored its adoption as a standard for random sampling of circuits.

Figure 1. BDD of a circuit, whose nodes are annotated with their probabilities

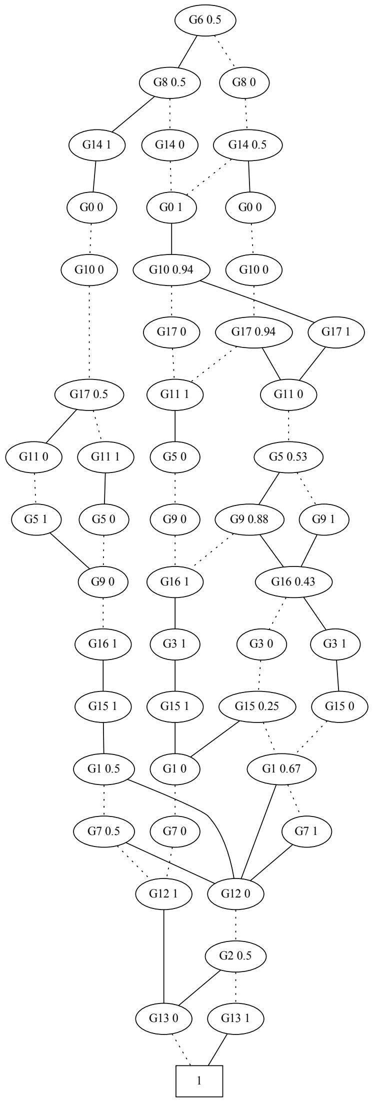


Figure 2. Snippet of bench language

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs +
# 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)
G15 = OR(G12, G8)
G16 = OR(G3, G8)
G9 = NAND(G16, G15)
G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

In general, random samplers can be classified into two families: those that use graph structures like BDDs and generalizations thereof, like deterministic Decomposable Negation Normal Form (d-DNNF), and those relying on search trees, typically improvements over Davis-Putnam-Logemann-Loveland (DPLL) algorithms Davis et al. (1962) used as model counters.

On the one hand, a BDD Bryant (1986) is a top-down rooted acyclic graph, which represents logic values as an alternative to logical formulae. To do so, variables are ordered, each node has an associated variable and two children: the low child, connected with a dotted line, corresponds to setting the variable to false, and the high child, connected via a straight line, corresponds to setting the variable to true. The variable associated with the parent precedes those of the children. It is similar to a binary decision tree, using a graph instead of a tree to reduce the number of nodes by merging any isomorphic subgraphs into one. Also, the parent is omitted when both children are the same. The input formula to build a BDD is not required to be in any logical form, which is an advantage over DPLL methods that dictate the use of CNF. Nevertheless, the number of nodes in a BDD is very sensitive to the variable ordering. Figure 1 shows an example BDD expressing the translation of the circuit in Figure 2 with node 0 and the paths leading to it omitted for clarity. Heuristic reordering of the variables has reduced the number of nodes from 145 to 49.

On the other hand, DPLL Davis et al. (1962) is based on partially building and traversing a binary search tree, and is the foundation of many SAT solvers and model counters.

Algorithm 1 Knuth's node annotation algorithm

```
1: function ANNOTATE($bdd$)
2:     for $t \in ReverseTopologicalOrder(bdd)$ do
3:         if $t = 0$ then
4:             $count(t) \leftarrow 0$
5:         else if t = 1 then
6:             $count(t) \leftarrow 1$
7:         else
8:             $thenCount \leftarrow count(t.high)2^{level(t.high)-level(t)}$
9:             $elseCount \leftarrow count(t.low)2^{level(t.low)-level(t)}$
10:            $count(t) \leftarrow thenCount + elseCount$
11:            $Pr(t) \leftarrow thenCount/count(t)$
12:    return $bdd$
```

The first Uniform Random Sampling (URS) algorithm for logical formulae was presented in Knuth (2009). This algorithm uses a BDD as input, although some properties of the BDD are computed before the generation takes place: We reproduce this step as pseudocode in Algorithm 1. The nodes in the BDD are traversed in reverse topological order, that is, from bottom to top, and for each node, the number of valid derived solutions is computed using that same information from its children. This information is then used to decorate each node with the probability of reaching node 1 through the high child. Figure 1 shows such an annotated BDD. Once the BDD is annotated, the generation step is performed using Algorithm 2: The graph is traversed from the root by generating random numbers and comparing them to the node probability to decide whether the low or high child are visited next, which decides the value of the corresponding variable, until node 1 is reached.

Unigen algorithms resulted from later iterations of a tool presented by Chakraborty et al. (2015), Chakraborty et al. (2013, 2014), called *UniWit*, that used universal hashing functions to partition the search space into roughly equivalent cells. Once a cell has been decided upon, the *CryptoMiniSAT*[1] solver is applied. Further running time improvements, such as allowing partial solution extraction and intelligent reuse of solutions, were implemented in *Unigen2* and *Unigen3* Soos et al. (2020). These algorithms hinge on the concept of *independent support*: A subset of variables

[1]https://www.msoos.org/cryptominisat2

Algorithm 2 Knuth's random sampling algorithm

```
1: function GENERATE($bdd$)
2:     $state \leftarrow$ sequence of size level(1) initialized to false
3:     $pos \leftarrow 0$
4:     $trav \leftarrow root(bdd)$
5:     if $trav = 0$ then                          ▷ bdd represents false
6:         return $state$
7:     while $trav \neq 1$ do
8:         while $pos < level(trav)$ do           ▷ Level jump
9:                                                 ▷ Equally likely
10:            $state[pos] \leftarrow randomBoolean()$
11:            $pos \leftarrow pos + 1$
12:                                    ▷ $0 \leq random() \leq 1$
13:        if $random() \leq Pr(trav)$ then
14:            $trav \leftarrow trav.high$
15:            $state[pos] \leftarrow true$
16:        else
17:            $trav \leftarrow trav.low$
18:        $pos = pos + 1$
19:    while $pos < level(1)$ do                  ▷ Level jump
20:        $state[pos] \leftarrow randomBoolean()$
21:        $pos \leftarrow pos + 1$
22:    return $state$
```

such that the rest of the variables are a function of these. The independent support set corresponds to the inputs to the circuit, meaning that only the independent variables need to be sampled first, and then the rest of the values simply have to be computed. The algorithms are supposed to perform very well if an independent support set is provided, and very poorly if it is not.

Oh et al. (2017) used *counting BDDs*, to generate random samples to guide the search for near-optimal configurations. The variable set size was considered a limiting experimental factor to build the BDDs. Afterwards, in Oh et al. (2019), the tool named *SMARCH* was proposed, which is built on top of the #SAT solver *sharpSAT* Thurley (2006). Specifically, the number of solutions is counted with this tool, and then a random number is generated to select one of those solutions. Then, a binary search is applied to get the solution in question by determining the value of each variable successively with the aid of more calls to the solver. The algorithm is quite straightforward, but does not include any optimizations, so performance is expected to be poor. Scalability depends on what sharpSAT can handle.

Table 1. Key characteristics of the compared tools

Characteristics	Smarch	Unigen3	Quick Sampler	SPUR	KUS	Genrandom
Based on graphs					●	●
Based on DPLL	●	●	●	●		
Independent support		●	●			
Limited input set size	●					
SAT solver used	sharpSAT	CryptoMiniSAT	MAX-SAT	sharpSAT		

QuickSampler is another sampling tool conceived by Dutra et al. (2018). It works by first generating a candidate solution randomly, and then applying a *MAX-SAT* solver to find a solution similar to the candidate. From there, a series of mutations are applied (i.e., flipping variables values) to generate more candidates in following calls to the solver. *QuickSampler* was designed to be used for fuzzy testing, so it does not matter if some candidates are not real solutions to the constraints. This tool also benefits greatly from using an independent support set. Without it, the performance of the tool is expected to degrade considerably.

Achlioptas et al. (2018) developed *SPUR*, a modification of *sharpSAT*. *SPUR* performs a DPLL counting search like *sharpSAT*, catching components for efficiency. It also stores partial solutions to produce combined solutions to the global problem, a technique called reservoir sampling.

The last sampling tool in this review, named *KUS*, was presented by Sharma et al. (2018). *KUS* uses a generalization of BDDs, called d-DNNF, which is a strict superset of BDDs, to perform URS. A d-DNNF is a graph consisting of AND nodes and OR nodes. Knuth's algorithms are generalized for this structure. The OR nodes represent disjunctions over disjoint variables, so the probabilities of the children can be added after some adjustment to get the probability of the parent. AND nodes also feature disjoint variable sets, so probabilities can be computed by multiplying the adjusted probabilities of the children. *KUS* relies on d4, a d-DNNF compiler. An advantage of *KUS* is that the whole sample is generated with a single traversal of the graph. *KUS* evaluation showed it was faster than *SPUR* and *Unigen3*. It is a very fast algorithm, provided building the d-DNNF is viable.

Table 1 summarises the key characteristics to distinguish the aforementioned tools.

The original benchmark circuits are generally not available, often mainly due to intellectual property issues. Therefore, reverse-engineering methods have been applied to try to recover the original files, although there is always some information loss when encoding a circuit to CNF. Li (2000) presents an effort to extract equivalences. Simple AND and OR gates matches were found in Ostrowski et al. (2002). Roy et al. were the first to extract logic gates from CNF using a generic graph matcher Roy et al. (2004). Later, Fu and Malik (2007) not only extracted logic gates but also guaranteed to extract the biggest acyclic circuit possible using an SAT solver. This approach is based on a gate library that describes the gates to extract, which makes it more flexible but less efficient than pattern matching.

Seltner (2014) and Biere developed a program called *cnf2aig*[2] that can reconstruct circuits from CNFs and outputs them as and-inverter graphs. It consists of algorithms for detecting the most common hardware gates in CNF. It also implemented a solution for the partial MAX-SAT problem that guarantees that the reconstructed circuit is maximal with respect to the gates it detects.

In Section 5, we will perform our own comparison of *Unigen3*, *SMARCH*, *Quicksampler*, *SPUR* and *KUS* to asses their scalability against our proposed tool *genrandom*. These tools have been chosen among others for being state-of-the-art. *SMARCH* is useful mainly as a baseline. *Unigen3* and *QuickSampler* are supposed to perform very well with a support set. *SPUR* will show the limits of optimized DPLL search and we will push the ability of the d4 compiler, which *KUS* depends upon, to obtain the d-DNNF graphs.

3. Translating circuits to logic

The benchmark used in this paper to test the different random samplers comes from the IEEE International Symposium on Circuits and Systems (ISCAS) in 1989. It is composed of the models shown in Table 2 with their corresponding characteristics. Whereas most benchmarks solely include the CNF representation of the circuits, ISCAS'89 data set provides the complete

[2]http://fmv.jku.at/cnf2aig/

original circuits, which are the inputs our algorithm requires.

Table 2. Tested circuits

Model	#Variables	#Support	#Clauses
s344	184	9	429
s499	175	1	491
s635	320	2	762
s938	512	34	1,233
s967	439	16	1,157
s991	603	65	1,337
s1196	561	14	1,538
s1269	624	18	1,616
s1512	866	29	2,044
s3271	1,714	26	4,269
s3330	1,961	40	4,605
s3384	1,911	43	4,440
s4863	2,495	49	6,434
s6669	3,402	83	8,423

Accordingly, the original circuits are encoded in *bench*, which is a hierarchical description language widely used in the benchmark circuit description, such as ISCAS'85, ISCAS'89, and ISCAS'99. This type of language provides a mechanism to define a system and to reuse the system to build another one. The different elements incorporated are the system's name, the ports (inputs, outputs, or bidirectional inputs), the function, and an instantiation of a system in another one. An example of its contents is shown in Figure 2.

The translation of logic gates to propositional logic is relatively straightforward. D-type flip-flops are circuit elements that delay the change of state of its output signal until the next rising edge of a clock timing input signal. They are used as storage in sequential circuits. If we take, for instance, A = DFF(B), A takes its value from B *before* it is evaluated, which, in principle, could have any value. Usually, B will have a definition later on which corresponds to the value *after* evaluation. Our translation will pick A as an input variable, and B's definition is translated as is. That way, the value of the D flip flop before B's evaluation goes to A, and the value after evaluation goes to B.

The grammar and semantic actions for the translation can be seen in Table 3. A sequence of input variables is compiled, together with a sequence with the rest of the variables and another sequence of Boolean logic expressions. This information can be stored in a file or be translated further into CNF. We also order the variables in such a way that the variables used in a gate definition precede it.

4. Circuit sampling

Knuth's sampling algorithm for BDDs is very fast. However, the size of a BDD may grow exponentially with the number of variables. Sadly, this worst-case scenario is very close to common practice. To avoid this problem, we propose using an ensemble or base of BDDs, which is essentially a sequence of several BDDs, with the particularity that nodes may be shared between BDDs. One BDD is created for the translation of each gate. Individual BDDs are easy to compute because there are usually only a few variables involved.

The translation obtained in section 3 is used to build a BDD base: All the variables, with the exception of input variables, have a definition, which is preceded by a comment with the name of the variable being defined to mark the beginning of a gate translation. We build a BDD for each variable definition. The variables are ordered in such a way that the variables in a definition are either input variables or have already been defined. Figure 3 shows a BDD base of the example circuit annotated with probabilities. The roots are colored in orange and the other non-terminal nodes in cyan. We have omitted the probabilities in read-only nodes and the paths leading to node 0.

Our approach to sampling is based on Knuth's URS algorithm for BDDs Knuth (2009). It consists of a two-step strategy: The first step is to annotate the nodes using Algorithm 1. The difference is that this time we apply it to a BDD base instead of a single BDD.

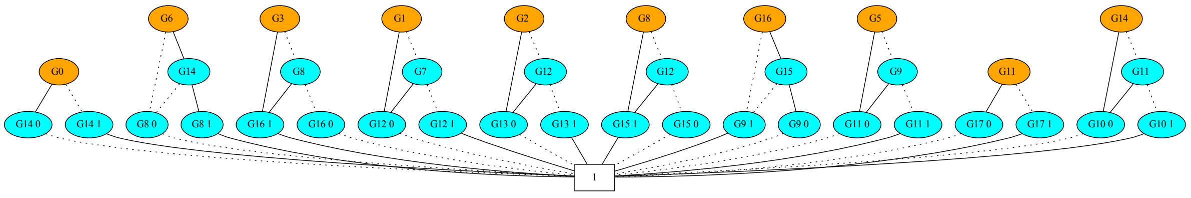
In the second step, we generate as many random states as needed. For that, we use Algorithm 3. It is very similar to Knuth's GENERATE algorithm 2 except it is performed in sequence, once for each BDD: First, the inputs are assigned random values (their BDDs point to 1). Then each non-input variable is assigned a value by traversing its BDD from the root: The variables appearing in the definition already have a value, so the next nodes to traverse are selected according to that value. Finally, the variable being defined is assigned a value according to the node probability. After this, node 1 must have been reached, another difference with Knuth's algorithm which may keep on assigning variables.

Following the algorithm and Figure 3, we see that to get a random value for G14, we assign the value opposite to G0. After that, we can get a value for G8: If G6 is false, so is G8, otherwise, the value for G8 is the same as the value for G14, and so on.

Table 3. Circuit syntax and associated semantic actions for translation

non-terminal	production	semantic action
file	ϵ	
file	line	
file	file line	
line	INPUT (NAME)	addInput(NAME)
line	OUTPUT (NAME)	addOutput(NAME)
line	output = type (inputs)	addVar(output)
		addExp(Comment(output)
		switch(type)
		case or : addExp(NAME $\rightarrow$ OR(inputs))
		addExp(OR(inputs) $\rightarrow$ NAME)
		break;
		case nor : addExp(NAME $\rightarrow$ NOT(OR(inputs)))
		addExp(NOT(OR(inputs)) $\rightarrow$ NAME)
		break;
		case not : addExp(NAME $\rightarrow$ NOT(inputs))
		addExp(NOT(inputs) $\rightarrow$ NAME)
		break;
		case and : addExp(NAME $\rightarrow$ AND(inputs))
		addExp(AND(inputs) $\rightarrow$ NAME)
		break;
		case nand : addExp(NAME $\rightarrow$ NOT(AND(inputs)))
		addExp(NOT(AND(inputs))$\rightarrow$ NAME)
		break;
		case nor : addExp(NAME $\rightarrow$ NOT(OR(inputs)))
		addExp(NOT(OR(inputs))$\rightarrow$ NAME)
		break;
		case dff : addInput(inputs)
inputs	NAME	return NAME)
inputs_1	inputs_2 , NAME	return concat(inputs_2, NAME)

Figure 3. Annotated BDD base of a circuit



Algorithm 3 Random sampling over a BDD base

```
1: function SAMPLE($forest, heads$)
2: ▷ heads is a sequence of the position of the defined
3: ▷ variable for each gate definition
4:     $state \leftarrow$ Boolean sequence initialized to false
5:     for $i \leftarrow 0$ to $size(heads) - 1$ do
6:         $trav \leftarrow roots(forest)[i]$
7:         $pos \leftarrow heads[i]$
8:                                     ▷ Read previous gate values
9:         while $level(trav) < pos$ do
10:            if state[level(trav)] then
11:                $trav \leftarrow high(trav)$
12:            else
13:                $trav \leftarrow low(trav)$
14:                                     ▷ Compute gate value
15:        if $trav = 1$ then                    ▷ Input
16:            $state \leftarrow randomBool()$
17:        else                                  ▷ Not input
18:            if $Pr(trav) = 1$ then
19:                $state[pos] = true$
20:                $trav \leftarrow high(trav)$
21:            else
22:                $trav \leftarrow low(trav)$
23:    return $state$
```

The sampling is uniform, because:

- The input variables (i.e., the independent support) can have any value, so they are chosen uniformly.
- The rest of the variables are a function of the input variables, so the probability of a state is the same as the probability of choosing the input variables, namely $\frac{1}{2^s}$, where s is the input variable size.

Importantly, the reduction in the number of nodes is noticeable even in this small example. It is down to 29 in the BDD base from 145 in the original BDD with no heuristic variable ordering.

5. Experimental results

We have articulated this section around two research questions:

- **RQ1**: How feasible and scalable are the different approaches?
- **RQ2**: What degree of performance can be achieved?

The experiments were carried out on an HP Proliant Gen9 server with two Xeon E5-2660v4 processors with 28 cores each and 224 Gb of memory. To manage the BDDs, we used the CUDD[3] library, which has built-in support for BDD bases.

Table 2 shows the circuits we have translated from the ISCAS'89 conference, together with the number of variables and clauses. It is worth pointing out that although the BDD base for *Genrandom* was built straight from the translation, the circuits had to be further translated to CNF. There is quite a range of sizes, which will provide the variability needed to compare competing approaches, which are expected to show very different behaviour. The size of the support set is also reported because *Quicksampler* and *Unigen3* rely on it to achieve better performance.

KUS and our approach, *Genramdom*, rely on building a d-DNNF graph and a base of BDDs, respectively. Table 4 shows the number of nodes for each approach, Table 5 shows the time needed to build each of the graphs. *Genrandom* avoids the exponential growth of the nodes by simplifying the problem as only one BDD is generated for each gate definition, showing a linear growth. *KUS*, however, builds one giant graph with exponential growth. The d-DNNF graph for the biggest system circuit, *s6669*, has a footprint of 34 Gigabytes and almost 75 million nodes. Meanwhile, our BDD base only takes 336K of space with roughly 8 thousand nodes. Clearly, *KUS* is very close to the limits of its scalability potential, while *Genrandom* scales very nicely. The same thing can be said in terms of time: It takes almost 3 hours to build the d-DNNF for *s6669*, while *Genrandom* builds the BDD base in eleven seconds.

Table 4. Sizes of BDDs and d-DNNF

Model	#BDD Nodes	#d-DNNF nodes
s344	430	281
s499	492	408
s635	763	369
s938	1,234	635
s967	1,158	7,179
s991	1,338	804
s1196	1,539	6,828
s1269	1,617	1,220,102
s1512	2,045	23,032
s3271	4,270	22,379
s3330	4,606	927,096
s3384	4,441	8.978
s4863	6,435	2,227,121
s6669	8,424	74,983,801

To further compare the scalability and performance of the approaches, we designed and conducted an

[3]https://github.com/ivmai/cudd

Table 5. Time taken to build BDDs & d-DNNFs (seconds)

Model	#BDD base	d-DNNF
s344	0.207	0.023
s499	0.208	0.022
s635	0.397	0.051
s938	0.683	0.077
s967	0.588	0.304
s991	0.834	0.197
s1196	0.825	0.828
s1269	0.930	118.375
s1512	1.486	2.068
s3271	3.894	8.920
s3330	4.835	80.418
s3384	4.728	1.255
s4863	7.169	1584.111
s6669	11.474	10,605.473

experiment to produce a thousand samples of each of the circuits. The results are shown in Table 6. Starting with *SMARCH*, we can see that the performance is very poor as it is the slowest sampler by several orders of magnitude. Scalability is also rather poor as it fails to deliver for four of the systems. The problem is that sharpSAT is unable to count the number of solutions, which means that DPLL model counting cannot scale. *Unigen3* and *Quicksampler* show very good performance except for the two biggest circuits. Both samplers rely heavily on independent support sets, and when they reach a certain size, performance starts to degrade. Even so, scalability is good insofar as all circuits were correctly sampled. SPUR is faster than *Unigen3* and *Quicksampler* for the smaller circuits, but slower for the biggest. It also fails to complete two of the samples, which shows that scalability is not so good. *KUS* shows a clear dependence on the number of nodes of the d-DNNF: When it is small, the system is rather fast, when it is big, the system is very slow. The number of nodes depends on the complexity of the circuits, more than the number of gates, which produces unpredictable results. For this reason, *KUS* fails to sample circuit s6669: The d-DNNF was built, but it is so big that generating samples takes too long. Last but not least, there is *Genrandom*. Not only does it beat all the other samplers in terms of performance, it does so while showing minimal delays when circuit size grows, making it also the most scalable approach. It takes less than one second to sample each of the systems. Although Unigen3 and Quicksampler lag Genrandom one order of magnitude for the bigger circuits, it is worth mentioning that these samplers by default only sample the independent support variables, which means that computing the value of the rest of the variables may take a rather longer time.

To answer **RQ1**, we may say that all approaches are feasible for small to medium circuits. For large circuits, only Quicksampler and Genrandom qualify. In terms of scalability, the most scalable system is *Genrandom*, followed by *Unigen3*, *Quicksampler*, *KUS*, *SPUR* and *SMARCH*, respectively.

The answer to **RQ2** is that the strongest performance was shown by *Genrandom*, followed by *Unigen3*, which puts the independent support set to good use. The rest of the systems degrade quickly with the size of the input.

6. Conclusions and Future work

We have proposed a novel way of using a base of BDDs to perform random sampling over translations of ICs to Boolean logic in order to apply fuzzy testing techniques. Our tool, *Genrandom*, is uniform by design, while showing no penalty in performance. A collection of 14 circuits from the ISCAS'89 conference have been translated to Boolean logic to be used as benchmark. We have compared it against five uniform random sampling tools and found *Genrandom* to be the fastest, beating the second-best by an order of magnitude. Moreover, our algorithm can perform sampling of even the biggest circuit, *s6669*, with ease. We conclude that our approach is the best among the evaluated options in terms of performance and scalability.

As future work, we will try to adapt our sampling approach to circuits specified in CNF, instead of an intermediate-level language, to access a wider selection of circuits to benchmark.

Acknowledgments

This work has been supported by the Universidad Nacional de Educacion a Distancia (projects 2022V/PUNED/007 and 2021V/-TAJOV/001).

References

Achlioptas, D., Hammoudeh, Z. S., & Theodoropoulos, P. (2018). Fast sampling of perfectly uniform satisfying assignments. *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 135–147.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *C-35*(8), 677–691.

Table 6. Sampling time in seconds for one thousand states

Model	Smarch	Unigen3	Quick sampler	SPUR	KUS	Genrandom
s344	3826.574	0.160	0.353	0.990	0.311	0.065
s499	8571.137	0.010	0.536	0.813	0.362	0.055
s635	22623.871	0.010	2.335	2.499	0.532	0.129
s938	37917.224	1.600	0.797	3.271	0.903	0.135
s967	34159.838	0.630	0.991	3.149	3.027	0.120
s991	41241.301	8.130	0.896	1.960	1.277	0.144
s1196	44089.649	0.420	1.857	3.818	3.897	0.180
s1269	Error	0.580	1.511	873.440	760.700	0.182
s1512	104928.073	1.750	1.393	11.466	16.571	0.278
s3271	181816.511	1.110	2.975	111.652	2831.127	0.465
s3330	Error	1.740	7.783	Error	8630.214	0.584
s3384	361704.470	2.150	3.301	111.119	6.343	0.574
s4863	Error	2.480	453.015	376.705	2477.425	0.741
s6669	Error	7.150	329.365	Error	Error	0.770

Chakraborty, S., & Meel, K. S. (2019). On testing of uniform samplers. *33rd Conference on Artificial Intelligence (AAAI)*, 7777–7784.

Chakraborty, S., Fremont, D. J., Meel, K. S., Seshia, S. A., & Vardi, M. Y. (2015). On parallel scalable uniform SAT witness generation. *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 304–319.

Chakraborty, S., Meel, K. S., & Vardi, M. Y. (2013). A Scalable and Nearly Uniform Generator of SAT Witnesses. *25th International Conference on Computer Aided Verification (CAV)*, 608–623.

Chakraborty, S., Meel, K. S., & Vardi, M. Y. (2014). Balancing scalability and uniformity in sat witness generator. *51st Annual Design Automation Conference (DAC)*, 1–6.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, *5*(7), 394–397. https://doi.org/10.1145/368273.368557

Dutra, R., Laeufer, K., Bachrach, J., & Sen, K. (2018). Efficient Sampling of SAT Solutions for Testing. *40th International Conference on Software Engineering (ICSE)*, 549–559.

Fu, Z., & Malik, S. (2007). Extracting logic circuit structure from conjunctive normal form descriptions. *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, 37–42. https://doi.org/10.1109/VLSID.2007.81

Heradio, R., Fernandez-Amoros, D., Galindo, J., Benavides, D., & Batory, D. (2022). Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering*, *27*(2), 44.

Knuth, D. E. (2009). *The art of computer programming, volume 4, fascicle 1: Bitwise tricks & techniques; binary decision diagrams*. Addison-Wesley Professional.

Li, C. M. (2000). Integrating equivalency reasoning into davis-putnam procedure. *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 291–296.

Oh, J., Batory, D., Myers, M., & Siegmund, N. (2017). Finding Near-optimal Configurations in Product Lines by Random Sampling. *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 61–71.

Oh, J., Batory, D. S., Heule, M. J. H., Myers, M., & Gazzillo, P. (2019). *Uniform sampling from kconfig feature models* (tech. rep. TR-19-02). Department of Computer Science. The University of Texas at Austin.

Ostrowski, R., Gregoire, E., Mazure, B., & Sais, L. (2002). Recovering and exploiting structural knowledge from cnf formulas. *2470*, 185–199. https://doi.org/10.1007/3-540-46135-3_13

Roy, J., Markov, I., & Bertacco, V. (2004). Restoring circuit structure from sat instances. *ACM/IEEE Intl. Workshop on Logic and Synthesis, Temecula, CA*, 361–368.

Seltner, H. (2014). *Extracting hardware circuits from CNF formulas* (Master's thesis). Johannes Kepler Universität Linz.

Sharma, S., Gupta, R., Roy, S., & Meel, K. S. (2018). Knowledge Compilation meets Uniform Sampling. *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 620–636.

Soos, M., Gocht, S., & Meel, K. S. (2020). Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. *Computer aided verification* (pp. 463–484). Springer International Publishing. https://doi.org/10.1007/978-3-030-53288-8_22

Takanen, A. (2009). Fuzzing: The past, the present and the future. *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 202–212.

Thurley, M. (2006). sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 424–429.