Output Sampling for Output Diversity in Automatic Unit Test Generation

Héctor D. Menéndez, Michele Boreale, Daniele Gorla, and David Clark

Abstract—Diverse test sets are able to expose bugs that test sets generated with structural coverage techniques cannot discover. Input-diverse test set generators have been shown to be effective for this, but also have limitations: e.g., they need to be complemented with semantic information derived from the Software Under Test. We demonstrate how to drive the test set generation process with semantic information in the form of output diversity. We present the first totally automatic output sampling for output diversity unit test set generation tool, called OutGen. OutGen transforms a program into an SMT formula in bit-vector arithmetic. It then applies universal hashing in order to generate an output-based diverse set of inputs. The result offers significant diversity improvements when measured as a high output uniqueness count. It achieves this by ensuring that the test set's output probability distribution is uniform, i.e. highly diverse. The use of output sampling, as opposed to any of input sampling, CBMC, CAVM, behaviour diversity or random testing improves mutation score and bug detection by up to 4150% and 963% respectively on programs drawn from three different corpora: the R-project, SIR and CodeFlaws. OutGen test sets achieve an average mutation score of up to 92%, and 70% of the test sets detect the defect. Moreover, OutGen is the only automatic unit test generation tool that is able to detect bugs on the real number C functions from the R-project.

Index Terms—Unit Testing, Output Sampling, Output Diversity, SMT Solver, OutGen

1 Introduction

In software testing, structural coverage of code is often used to drive automated test set generation, particularly for unit testing. It has been encapsulated in state of the art search based tools such as EvoSuite [?] and CAVM [?] and is regarded as an industry standard. However, there is evidence that test sets constructed on the basis of branch or line coverage have limitations on their ability to discover faults in code [?]. For example, an empirical study discovered that "Although the automatically generated test sets detected 55.7% of the faults overall, only 19.9% of all the *individual test sets* detected a fault" (our emphasis) [?]. We could improve the fault finding ability of coverage if we could "add semantic information" and diversity to the test set, but what can this mean in practice?

A useful clue can be found in the work of Alshahwan and Harman on output uniqueness as a principle for selecting test sets for web pages [?], [?]. Diversity, particularly input diversity, for test sets has been widely studied in different forms [?], [?], [?] but here the attention is drawn by the use of outputs. Outputs capture "semantic information" by their very nature. Inputs do not contain semantic information, but any observation of a complete or partial execution of a program on an input can be deemed an output of the program.

Alshawan and Harman studied the correlation between numbers of unique outputs in test sets and bugs detected. They defined a notion of observation (similarity/uniqueness) on outputs; sampled inputs from a uniform distribution on the inputs; executed the program on these inputs; and evaluated how the number of unique outputs correlated with the bugs identified in programs. They showed that output uniqueness correlated highly ($> 90\%$) with coverage and had superior fault finding ability ($+47\%$) over coverage [?]. Why is this measure so efficacious?

One explanation can be found in the work of Clark and others [?], [?]. A notion of observation/oracle defines a "testing channel" in information theoretic terms. If the program is deterministic, the maximum capacity of the channel is given by $\log_2 |O|$, i.e. the logarithm of the number of possible unique outputs. Assuming we have a set of inputs that produces exactly all the unique outputs, adding additional inputs does not increase the capacity. So, identifying a "channel adequate" set of inputs involves selecting one input from the inverse image of each output. In general it may be difficult or impossible to find a channel adequate set of inputs which leads naturally to the highest output uniqueness score, as Alshahwan and Harman remarked [?]. However, our work here creates a good approximation by creating diversity on the outputs during the automatic unit test generation process.

We focus on two questions: how can we automatically generate unit test sets with a high output uniqueness count? And can we show that this is useful? To answer these questions we develop a constraint solving approach and redesign the XORSample' algorithm of Gomes and others for finding diverse witnesses (or models) for constraints [?]. Rather than sampling diversely from the input space for a program, as did Alshahwan and Harman, we adapt the Gomes' algorithm to sample diversely from the output space as a more efficient route to high output uniqueness scores. To this aim, we

- *H.D. Menendez is at the Department of Computer Science, Middlesex University London, London, UK. E-mail: h.menendez@mdx.ac.uk*
- *D. Clark is at the Department of Computer Science, University College London, London, UK. E-mail: david.clark@ucl.ac.uk*
- *M. Boreale is with University of Florence, Italy. D. Gorla is with Sapienza University of Rome, Italy.*

Manuscript received April 19, 2005; revised August 26, 2015.

develop the first automated and Output Diversity Driven (ODD) unit test set generating tool, called OutGen. This use of output diversity introduces a difference between our work and that of Alshahwan. We characterise their work as an Input Diversity Driven (IDD) approach to optimising a test set's output uniqueness score.

This difference turns out to be significant. Modulo size, ODD test sets are evidently statistically better than IDD test sets in two ways. ODD test sets have, on average, a 50% higher output uniqueness score than IDD test sets. ODD test sets also have a 63% higher correlation between output uniqueness scores and ability to detect bugs when compared to IDD test sets. This increased power in bug detection is clearly manifested in our experiments on functions from the R-project [?], where only ODD test sets were able to find real bugs. We conjecture that sampling diversely from outputs has more chance of identifying inputs corresponding to rare software behaviours than sampling diversely directly from inputs.

Our experimental effort was conducted on 8 mathematical C functions from the R-project [?], the `tcas` program from SIR [?] and 100 C program pairs drawn from the CodeFlaws repository [?]. Each pair consists of a program with a defect (a real error) and the fixed version. For each program pair, we generated a pool of 500 test inputs per approach to test set generation. We compared six approaches: repeated solutions to constraints via CBMC's solver, CAVM, behaviour diversity, random selection of inputs using sampling from a uniform distribution, as well as IDD and ODD. The first four are merely sanity checks. Our main interest was in comparing ODD and IDD (Section 6). Apart from the above mentioned differences in average output uniqueness scores and bug finding ability between ODD and IDD, we found broadly similar correlations between output uniqueness scores and coverage criteria scores for both methods; this replicates and confirms Alshawan and Harman's results [?] in the (new) context of C programs.

ODD test sets were up to 4150% better at killing mutants and 963% better at detecting faults than the worst approach (Section 7). To ensure that this effectiveness is not time-dependent, we produced an experiment fixing the same amount of time to every generation tool. Our findings show that OutGen outperforms every other tool. ODD test sets are up to 1150% better than the worst tool at killing mutants and up to 728% better at finding faults.

The chief contributions of this paper are

1) ODD test set *generation*, a novel, white-box approach to unit test set generation that adds semantic information into test sets and is demonstrably superior to the existing IDD test set selection approach;
2) Theory that describes how the generation method works and that provides a theoretical guarantee of the quality of the diversity;
3) An open source tool, OutGen, that automatically generates ODD test sets;
4) Non-trivial, rigorous, statistical experimental validation of the usefulness of ODD test sets in comparison to Alshahwan and Harman's work as well as more standard techniques such as heuristic constraint solving, search, execution trace clustering and random input generation.

2 Background

While Alshahwan and Harman showed that output uniqueness correlates better with fault detection by the test set than any structural coverage metric, their control of the output samples was highly indirect, via sampling diversely from the input space. As the authors remarked in their future work discussion [?], increasing the output uniqueness scores of test sets is likely to lead to higher fault detection.

We aim at increasing the output uniqueness as much as possible by creating a test set associated with a diverse set of outputs. This means that our test set generator, considered as a random variable, will aim at having maximum entropy. According to Cover and Thomas [?], this can only happen when its probability distribution is uniform, i.e., by definition, every output – including rare outputs – has the same probability of appearing. As a consequence, this should increase the number of unique outputs produced by any test set generated by our generator.

To reach our generation goal, we develop a white box method that gives us control on the generated outputs. This novel method aims at reaching the maximum possible value of output uniqueness for a test set. We can then evaluate test set output uniqueness scores as generated via output diversity and compare with test set output uniqueness scores as generated via input diversity. Then we can compare both types of test set with respect to fault finding ability. To create our diverse outputs generator, we redesign the XORSample' algorithm [?].

2.1 XORSample': Universal Hashing on SAT Constraints

OutGen repurposes the XORSample' algorithm from sampling input diversity to sampling output diversity (Section 3) and switches its application context from electrical circuits, i.e. simple SAT constraints, to real programs represented as SMT constraint formulas (Section 4). We start by presenting the relevant background for this algorithm before giving the modifications required in the following sections.

Let f be a satisfiable propositional logic formula in conjunctive normal form. Let $\Sigma = \{0, 1\}$ be an alphabet for f's variables. For $\mathbf{x}, \mathbf{y} \in \Sigma^n$, f defines a relation on them when $f(\mathbf{x}, \mathbf{y}) = true$ [?]. We call *witness* any model of f, i.e. any value for $\mathbf{x}, \mathbf{y}$ satisfying this relation; let W_f denote the set of all witnesses of f. We assume the variables $\mathbf{x}$ and $\mathbf{y}$ all take values on a fixed domain $D \subseteq \Sigma^n$.

When f is submitted to a solver, we obtain a witness that is the result of a heuristic process [?]. The heuristic is deterministic and affects the witness generation process, producing the same result for every query with the same or similar parameters. This does not satisfy near-uniformity of the generation. Indeed, since we aim at diversity, the deterministic behaviour, embedded into the solver, is adversarial for us. We want to get a different witness in every query and to have these witnesses spread throughout the output space.

Our approach repurposes the XORSample' algorithm [?]. This algorithm partitions the input space of f into cells, near-uniformly chooses a cell and uniformly at random takes an element of the cell. The first selection (cell) spreads the witnesses through the whole space, reducing the adversarial behaviour of the solver [?]. Once the cell is chosen, the second selection is a uniform selection on every possible witness, of which there are fewer depending on the cell size. To guarantee that we are choosing the cell near-uniformly at random, we employ so called r-wise independent hash functions. To define this notion, here and in what follows, we write $a \in_u A$ to denote the uniformly at random selection of the element a from the set A.

Definition 1. Let n, s and r be positive integers, with $n \geq s$. Then, the family $\mathcal{H}(n, s, r)$ contains all the r-*wise independent* hash functions from D^n to D^s, i.e. all the $h : D^n \to D^s$ such that, for each $\alpha_1, \ldots, \alpha_r \in D^s$ and for each distinct $\beta_1, \ldots, \beta_r \in D^n$, it holds that

$$\mathtt{Pr}\left[\bigwedge_{i=1}^{r} (h(\beta_i) = \alpha_i) \ : \ h \in_u \mathcal{H}(n, s, r)\right] = \frac{1}{2^{sr}}$$

For selecting a cell near-uniformly at random, we uniformly choose a function h and a point $\alpha \in D^s$. As $s \leq n$, several inputs of D^n will be projected onto α. The inverse image $h^{-1}(\alpha)$ describes all these inputs. From this subset, the cell will be obtained as the restriction to the inputs that are witnesses for f_P. Figure 1 shows the projection schema. On the left hand side, we have the original space D^n, where the witnesses for f_P are denoted by the '$\star$' symbol. On the right hand side, we have the projective space D^s, onto which h projects the original space. The cell is the set of witnesses within the dashed square.

Within D^n, the r-wise independent hash function guarantees that: (1) for every $\alpha \in D^s$, there is a pre-image $h^{-1}(\alpha)$; (2) the set of all pre-images defines a partition of D^n; and (3) if we choose h and α uniformly at random, the cell selection is near-uniform [?]. The issue is then to find a family that is both efficient and able to generate and select cells near-uniformly. We consider $\mathcal{H}_{xor}(n, s, 2)$ [?], that contains all the 2-wise independent hash functions $h : \{0, 1\}^n \to \{0, 1\}^s$ made up of a s-tuple of functions $(h_1, \ldots, h_s)$ from $\{0, 1\}^n$ to $\{0, 1\}$ that can be uniquely described by an $a \in \{0, 1\}^{s(n+1)}$ as $h_i(\mathbf{b}) \triangleq a_{i,0} \oplus a_{i,1} b_1 \oplus \ldots \oplus a_{i,n} b_n$, where $i \in \{1, \ldots, s\}$, $\mathbf{b} \in \{0, 1\}^n$ and $\oplus$ is the XOR operator. Thus, the uniformly random selection of h from $\mathcal{H}_{xor}(n, s, 2)$ can be simply done by uniformly choosing $s(n + 1)$ random bits.

3 Adapting XORSample' for output diversity

Our method considers a propositional formula f_P, that represents a deterministic program P, and aims at creating witnesses whose outputs follow a uniform distribution. We assume that $f_P(\mathbf{x}, \mathbf{y}) = \mathit{true}$ if and only if $P(\mathbf{x}) = \mathbf{y}$, where $\mathbf{x} = (x_1, ..., x_n)$ are P's input variables and $\mathbf{y} = (y_1, ..., y_m)$ are its output variables. We let the input space be $\mathcal{X} \triangleq D^n$, the output space be $\mathcal{Y} \triangleq D^m$ and the space of *actual outputs* be $\mathcal{O} \triangleq \{v \in \mathcal{Y} \ : \ \exists u \in \mathcal{X}.\ f_P(u, v) = \mathit{true}\}$,

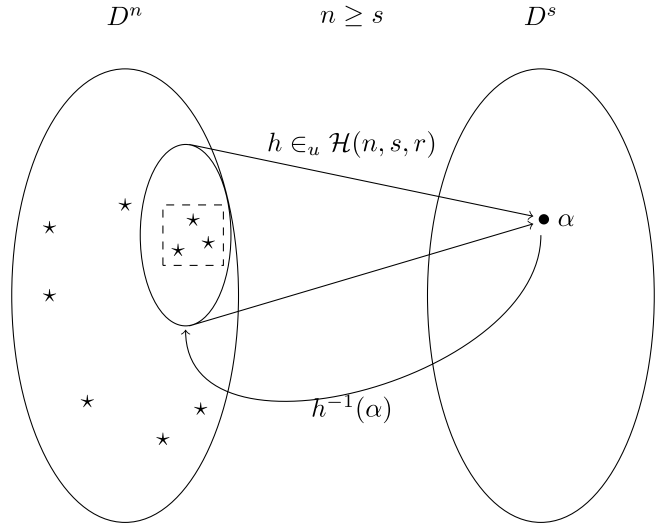

Figure 1. Cell description and selection process. The r-wise independent hash function h projects a set of inputs from D^n to a chosen value $\alpha \in D^s$. The inverse function $h^{-1}(\alpha)$ defines the cell (viz., the dashed square) when intersected with the set of all inputs that form a witness for f_P (denoted with $\star$).

where $\triangleq$ denotes "is defined as".

W.r.t. Section 2.1, f_P plays the role of f, and the program P is the relation connecting its arguments.

Let $\mathcal{F}$ denote the set of all propositional formulae f_P associated to a deterministic program P. A *generator* $G : \mathcal{F} \to \mathcal{X} \times \mathcal{Y}$ is a non-deterministic function that, given a formula f_P, returns a witness (u, v) for it. Our aim is to create a near-uniform generator.

Definition 2. A generator G is *near-uniform* if there exists $c \in (0, 1]$ such that $\mathtt{Pr}\left[G(f_P) = (u, v)\right] \geq \frac{c}{|W_{f_P}|}$, for every $f_P \in \mathcal{F}$ and $(u, v) \in W_{f_P}$.

To create an output diverse generator, we adapt XORSample' [?] to the output domain in three steps: (1) we randomly divide the set of actual outputs $\mathcal{O}$ into s cells of approximately equal size and randomly select one cell C; (2) we randomly select a pair $(u, v) \in \mathcal{X} \times \mathcal{O}$ such that $f_P(u, v) = \mathit{true}$ and $v \in C$; and (3) the input part u becomes the test case.

To deal with program variables, we also extend XORSample' from binary variables to bit-vector representations, similarly to [?]. For floating point arithmetic, we use a mixed abstraction to transform the floating point representation into bit-vectors [?]. This allows us to use satisfiability modulo theory (SMT) solvers with bit-vector theory. To this aim, let $l = \lceil \log_2 |D^m| \rceil$ and $t(\mathbf{y}, \mathbf{y}')$ be the formula saying that $\mathbf{y}' = (y'_1, ..., y'_l)$ is the binary encoding of $\mathbf{y}$, i.e.

$$t(\mathbf{y}, \mathbf{y}') \triangleq \left(\sum_{i=1}^{m} y_i |D|^{i-1} = \sum_{i=1}^{l} y'_i 2^{i-1}\right). \tag{1}$$

To divide the space $\{0, 1\}^l$ into 2^s cells, for a single random cell we consider the conjunction of s XOR constraints

$$C(\mathbf{y}') \triangleq \bigwedge_{i=1}^{s} c_i(\mathbf{y}'), \tag{2}$$

Algorithm 1 Output Diversity via XORSample'

Input: A formula $f_P(\mathbf{x}, \mathbf{y})$, representing a deterministic program $P : D^n \to D^m$; a natural number $s \leq n$

function *sample*(f_P, s)

1. $l \leftarrow \lceil \log_2 |D^m| \rceil$
2. Let $\mathbf{y}'$ be the tuple of fresh binary variables $y'_1, \ldots, y'_l$
3. $\forall i = 1, \ldots, s \; \forall j = 0, \ldots, l \; . a_{i,j} \in_u \{0, 1\}$
4. Build $F(\mathbf{x}, \mathbf{y}, \mathbf{y}')$ as specified in Equation (4) (and in (1) and (2))
5. $mc \leftarrow SMT_ModelCount(F(\mathbf{x}, \mathbf{y}, \mathbf{y}'))$
6. **if** $mc = 0$ **return** $\perp$
7. $\Lambda \leftarrow \emptyset$
8. **for all** (u, v, v') model of $F(\mathbf{x}, \mathbf{y}, \mathbf{y}')$ **do**
9. **if** $v \notin \{\bar{v} \; : \; (\bar{u}, \bar{v}, \bar{v}') \in \Lambda\}$ **then**
10. Add (u, v, v') to Λ
11. $(u, v, v') \in_u \Lambda$
12. **return** (u, v, v')

where, for $1 \leq i \leq s$:

$$c_i(\mathbf{y}') \quad \stackrel{\triangle}{=} \quad (a_{i,1} y'_1 \oplus \cdots \oplus a_{i,l} y'_l = a_{i,0}) \qquad (3)$$

with $a_{i,j} \in_u \{0, 1\}$. Note that each cell will contain, on average, the same number of actual outputs. The formula we submit to the SMT solver is:

$$F(\mathbf{x}, \mathbf{y}, \mathbf{y}') \stackrel{\triangle}{=} f_P(\mathbf{x}, \mathbf{y}) \wedge t(\mathbf{y}, \mathbf{y}') \wedge C(\mathbf{y}'). \qquad (4)$$

Even if F is a formula with variables $\mathbf{x}, \mathbf{y}, \mathbf{y}'$, the XOR constraints c_i are relatively short, because they only involve the binary variables $\mathbf{y}'$.

In this way, we near-uniformly partition the space of actual outputs. However, choosing at random one of the witnesses of F does not ensure that the obtained input generates an actual output with probability around $\frac{1}{|\mathcal{O}|}$ (that is the aim of our sampling procedure). To illustrate this, imagine that one output value v has several pre-images through P, i.e. many elements of the domain are mapped by P into v. It is then clear that uniformly sampling the space of witnesses of F will return an input value u that, with high probability, will have v as image. Thus, the induced probability distribution on outputs will be far from uniform.

For this reason, after having calculated all witnesses for F, we keep just one model for every output value. Then, uniformly choosing among these witnesses will yield a near-uniform distribution over all outputs. The detailed pseudo-code for the sampling algorithm adapts the algorithm XOR-Sample' developed in [?] and it is given in Algorithm 1. The main differences are:

- We work with an SMT solver instead of a SAT solver, because our formula f_P involves program values and not simply boolean variables, and requires to deal with more complex constrains (Section 4); for this reason, we have to include in F also the formula t, defined in (1).
- We do not uniformly choose one model of F, but first select (in lines 8–11) a representative model for every output value. Then, we uniformly choose among these.

Original Program

```
1 byte func(byte x)
2 {
3   if(x>=0) return pow(2,x);
4   return 0;
5 }
```

Program Constraints for P

$$F_P(x, y, y') = \wedge \begin{cases} f_P(x, y) = \wedge \begin{cases} (x \geq 0) \implies (y == 2^x) \\ (x < 0) \implies (y == 0) \end{cases} \\ C(y') = \wedge \begin{cases} \bigoplus_j (a_{1,j} y'_j) == a_{1,0} \\ \bigoplus_j (a_{2,j} y'_j) == a_{2,0} \end{cases} \\ t(y, y') \end{cases}$$

Potential Coefficients

$a_{1,j} = 00101111$	$a_{1,j} = 10000000$
$a_{2,j} = 11110000$	$a_{2,j} = 00110000$
$a_{i,0} = 11$	$a_{i,0} = 00$

Related Cells

$y' = 00100000 \quad x = 5$	$y' = 01000000 \quad x = 6$
	$y' = 00001000 \quad x = 3$
	$y' = 00000100 \quad x = 2$
	$y' = 00000010 \quad x = 1$
	$y' = 00000001 \quad x = 0$

Figure 2. Example of a program, the formula f_P related to the program constraints, and the manipulated formula including the hash functions F_P. The example also shows two potential set ups for the hashes coefficients and the cells that they generate. The algorithm selects an element of the cells uniformly at random to create an input.

Figure 2 shows an example of the whole process, illustrating the steps of Algorithm 1. It starts with the program P consisting of the function `func`. Then, it creates the program constraints defined as the formula $f_P(x, y)$. This formula and the value of s that we assume as 2 in the examples are inputs for Algorithm 1. In this example, there are only two scenarios: $x \geq 0$ and $x < 0$. Then, it sets the XOR constraints, following equation 3 (lines 1 to 3 of Algorithm 1). As $s = 2$, we only set two equations. Following algorithm 1, to create each equation we only need to choose the coefficients $a_{i,j}$ (line 3 of Algorithm 1). These coefficients are chosen uniformly at random. This allows us to select the cell near-uniformly. Once we have the hash equations that construct $C(y')$, we can create $F_P(x, y, y')$ combining $C(y')$, $t(y, y')$ and $f_P(x, y)$. Once we submit F_P to the solver, it will provide all the valid witnesses inside the cell defined by the hash functions, i.e., all the valid outputs of that cell. In the example, as the output is a power of 2, the first cell (left side) can only catch one solution (00100000), corresponding to 2^5. This solution is the only one that satisfies the two XOR constraints that define the cell. The second cell has six solutions. In this case, the algorithm selects one of these solutions uniformly at random (lines 8 to 10 of Algorithm 1), and this creates the input.

Soundness of the algorithm is proved by the following result.

Theorem 1. Let $|\mathcal{O}| = 2^{s^*}$ and $0 < s < s^*$. Let $v \in \mathcal{O}$ and $p(v)$ denote the probability that $sample(f_P, s)$ returns a model with v as output value (i.e., a model of the form (u, v, v')). Then,

$$p(v) \;>\; \frac{c'(s^* - s)}{2^{s^*}}$$

where $c'(\cdot)$ is the function

$$c'(\alpha) = \frac{2^\alpha}{2^\alpha + 1} \cdot \frac{2^{\frac{\alpha}{3}} - 1}{2^{\frac{\alpha}{3}} + 1}$$

Furthermore, $sample(f_P, s)$ returns $\bot$ with probability smaller than $1 - c'(s^* - s)$.

Proof Let $C(\mathbf{y}')$ be the XOR constraint built by $sample(f_P, s)$ in line 4. For every model $\sigma = (u, v, v')$ of the resulting formula F, let Y_σ be the binary random variable whose value is 1 if and only if $C(v') = 1$. Then, $\mathtt{E}[Y_\sigma] = 2^{-s}$ and $\mathtt{Var}[Y_\sigma] \le 2^{-s}$. Furthermore, if we call ℓ the random variable $\sum_{\sigma \in \Lambda} Y_\sigma$, we have that

$$\mathtt{E}[\ell \mid Y_\sigma = 1] = 1 + \frac{2^{s^*} - 1}{2^s} \qquad \mathtt{Var}[\ell \mid Y_\sigma = 1] \le \frac{2^{s^*} - 1}{2^s}$$

(the proof of these results is the same as for Theorem 2 in the appendix of [?]). Let us call μ the value of $\mathtt{E}[\ell \mid Y_\sigma = 1]$ and $\alpha = s^* - s$; then,

$$2^\alpha < 1 + 2^\alpha - 2^{-s} = 1 + 2^{s^* - s} - 2^{-s} = \mu < 2^\alpha + 1 \quad (5)$$

Let $\beta \ge 0$; by Chebychev's inequality and $\mathtt{Var}[\ell \mid Y_\sigma = 1] < \mu$, we have that

$$\mathtt{Pr}\left[|\ell - \mu| \ge \frac{\mu}{2^\beta} \;\middle|\; Y_\sigma = 1\right] \le \left(\frac{2^\beta}{\mu}\right)^2 \mathtt{Var}[\ell \mid Y_\sigma = 1] < \frac{2^{2\beta}}{\mu}$$

i.e.

$$\mathtt{Pr}\left[(1 - 2^{-\beta})\mu < \ell < (1 + 2^{-\beta})\mu \;\middle|\; Y_\sigma = 1\right] > 1 - \frac{2^{2\beta}}{\mu} \quad (6)$$

Finally, notice that, if $s > 1$, then

$$(1 + 2^{-\beta})\mu \le 2\mu < 2(2^\alpha + 1) = 2^{s^* - s + 1} + 2 < 2^{s^*} \quad (7)$$

otherwise, it can be possible that $(1 + 2^{-\beta})\mu > 2^{s^*}$, but ℓ will never assume values in $\left(2^{s^*} + 1, (1 + 2^{-\beta})\mu\right]$ because by hypothesis $|\mathcal{O}| = 2^{s^*}$.

To conclude:

$$\begin{aligned}
p(v) &\triangleq \mathtt{Pr}\left[sample(f_P, s) \text{ returns some } (u, v, v') \triangleq \sigma'\right] \\
&= \mathtt{Pr}\left[Y_{\sigma'} = 1 \;\wedge\; \sigma' \text{ is the } p\text{-th model of } \Lambda\right] \\
&= \mathtt{Pr}\left[Y_{\sigma'} = 1\right] \cdot \mathtt{Pr}\left[\sigma' \; p\text{-th model of } \Lambda \mid Y_{\sigma'} = 1\right] \\
&= \tfrac{1}{2^{s^*}} \textstyle\sum_{i=1}^{2^{s^*}} \frac{\mathtt{Pr}[\ell = i \mid Y_{\sigma'} = 1]}{i} \\
&\ge \tfrac{1}{2^{s^*}} \textstyle\sum_{i=1}^{(1+2^{-\beta})\mu} \frac{\mathtt{Pr}[\ell = i \mid Y_{\sigma'} = 1]}{i} && \text{by (7)} \\
&\ge \tfrac{1}{2^{s^*}} \textstyle\sum_{i=1}^{(1+2^{-\beta})\mu} \frac{\mathtt{Pr}[\ell = i \mid Y_{\sigma'} = 1]}{(1+2^{-\beta})\mu} \\
&= \tfrac{1}{2^{s}(1+2^{-\beta})\mu} \mathtt{Pr}\left[\ell \le (1 + 2^{-\beta})\mu \;\middle|\; Y_{\sigma'} = 1\right] \\
&> \tfrac{1}{2^{s^*}(1+2^{-\beta})\mu} \left(1 - \tfrac{2^{2\beta}}{\mu}\right) && \text{by (6)} \\
&> \tfrac{1 - 2^{2\beta - \alpha}}{2^{s^*}(1+2^{-\beta})(1+2^{\alpha})} && \text{by (5)} \\
&= \tfrac{1}{2^{s^*}} \cdot \tfrac{1 - 2^{2\beta - \alpha}}{(1+2^{-\beta})(1+2^{-\alpha})}
\end{aligned}$$

It can be proved that the second fraction gets its maximum for $\beta = \frac{\alpha - 1}{3}$. To write this second fraction with a simpler mathematical expression, viz. $c'(\alpha)$, we use (like in [?]) $\beta = \frac{\alpha}{3}$ and have the final bound.

What we have just proved entails the final claim: if we call ξ the output of $sample(f_P, s)$, it suffices to note that

$$\mathtt{Pr}\left[\xi \ne \bot\right] = \sum_{v \in \mathcal{O}} p(v) > c'(s^* - s)$$

□

According to the theorem, Algorithm 1 implements a near-uniform generator, due to $c'(s^* - s) \in (0, 1)$ whenever $s < s^*$ and $|\mathcal{O}| \le |W_{f_P}|$. This is the method by which we generate inputs, i.e. a test set, that produces a near-uniform set of outputs for a given program. The following section explains the technical adaptation of the algorithm to real programs.

4 OutGen: Output Diverse Generator for Unit Testing

The output diversity method creates witnesses from constraints, and, therefore, can be instantiated for several different problems, insofar as these problems can be modelled in an input/output fashion. This work instantiates it as an automatic tool for the generation of output-diverse test sets on C programs. We name this tool OutGen[1].

OutGen uses CBMC [?] to generate a formula that contains every possible program path to an output. This formula represents the behaviour of the program in single static assignment form [?]. Loops are handled in the usual way for CBMC: unwinding them to a preset limit. Once the formula is defined, we set the output. Initially, the output is determined by the observations $\mathcal{O}$ associated with the *return* values of the software under test (SUT). OutGen parses the CBMC formulae and translates them to expressions suitable for the Z3 solver. These expressions are SMT expressions in bitvector arithmetic, and mixed abstractions for floating points [?].

Some programs employ `scanf` to get interactive inputs and `printf` to set interactive outputs, during the program execution. To identify inputs that are assigned during the program execution, OutGen automatically instruments any `scanf` calls and global variables, setting them as inputs for the software testing framework. The parser controlling the instrumentation is `pycparser`, a C parser for Python. Once this step finishes, the f_P formula is passed to the XORSample' algorithm.

The next step applies the adaptation of XORSample' to the output domain (Section 3). OutGen automatically identifies the `return` expressions and propagates their values to the final state of a new local variable, defined as the ϕ-function of all possible return values. The ϕ-function selects the value, from all possible values, depending on the chosen branch. In this way, we produce a single formula that contains every output. Then, the hash functions are

1. You can find the open-source implementation and data of OutGen in https://github.com/hdg7/OutGen . This will be published on github after the reviewing process.

included as additional constraints to this formula. We set the observation from the outputs on the final states, and we automatically include the final state of the expression controlling the observable output, when this expression belongs to, or controls, a branch containing `printf` statements.

Once the constraints are included in f_P, OutGen asks the Z3 solver for witnesses, generating the test set. The solver generates inputs for the test set in such a way that they attain a uniform distribution on the outputs. To evaluate the resulting test set on the SUT, OutGen automatically creates a harness for the SUT, and runs the test set. The instrumentation limits the time allowed to run a test to 4 seconds, in order to avoid non-termination. The Linux tool `gcov` measures the coverage through an automatic instrumentation process. The `gcc` compiler performs the instrumentation and `gcov` measures the coverage during runtime. Once the harness finishes, OutGen provides statistics about both uniformity of outputs and coverage metrics. In the following sections, we will use this information to evaluate succes in achieving our aims (Section 7).

5 Research Goals

OutGen generates a set of program inputs with the property of near optimal output diversity (Section 3). To evaluate the usefulness of OutGen, we consider three questions. As Theorem 1 shows, OutGen is near-uniform, but not completely uniform. That means that our target for diversity might be compromised, especially considering that universal hashing only guarantees success of the algorithm on around the 80% of the trials [?], therefore we want to know empirically *"How uniform is the set of outputs for a test set in practice?"*. As we mentioned in Section 2.1, we created an output diverse generator to increase the output uniqueness measure, making it more likely that rare outputs be produced. While Alshahwan and Harman were able to find strong correlations between output uniqueness, coverage and faults detected, by applying input diversity, we want to know: *"How does output diversity affect the ouput uniqueness scores in comparison with input diversity?"*. Last but not least, we need to know how output diversity driven test set generation compares with other state of the art tools for automatic unit test generation. In those terms we ask: *"How effective is our test set generation method?"*. To answer the first question, our evaluation uses a statistic that measures the proximity of the outputs generated to a sample from a uniform distribution, our benchmark for diversity. To answer the second question, we evaluate the improvements on output uniqueness score between input diversity driven (the approach of Alshahwan and Harman [?]) and output diversity driven (OutGen), measuring it in terms of correlations with coverage, mutation scores and number of faults detected. For the effectiveness question, we measure the abilities both to kill mutants and to detect faults, and we also compare these with similar measures for four other generation approaches: pseudo-random generation, CAVM, behaviour diversity and CBMC. In what follows we elaborate these three questions into a set of research questions.

RQ1: For each generated test set, do all the outputs have the same probability of being generated?

According to Theorem 1, when OutGen generates an output v, $p(v) > c'(\alpha)/2^{s^*}$. Knowing that 2^{s^*} is the total number of outputs, p would only be a uniform distribution when $p(v) = 1/2^{s^*}$, by definition. So uniformity depends on the value of $c'(\alpha)$, that in turn depends on the chosen *cell* α, and the choice of pairwise independent hash functions. Here, we measure how much this α affects our goal, that is, how close we are to a uniform distribution. We use the L2-test to perform this evaluation [?]. Given an ϵ, the L2-test passes when the distribution is ϵ-far from the uniform distribution, using a collision-based approach. This test is chosen because it is one of the few statistical tests that can deal with non-continuous and heterogeneous domains, as the output domain often is (Section 8).

RQ2: How do OutGen generated test sets affect the output uniqueness scores in terms of correlations with: structural coverage metrics (for line, branch and path coverage), mutation score and fault detection, and how does this compare with those for input diversity?

We measure the Spearman rank correlations between both test set size and output uniqueness score with line, branch and path coverage, mutation score and fault detection. This information measures the influence of a test set's output uniqueness score and size on these test set metrics.

We also compare this correlation between the test set generated through output diversity (OutGen) and an equivalent test set generated from input diversity (XORSample'), via an adaptation of Alshawan and Harman's method to C programs.

For mutation testing, we create mutants, i.e. variations on the original programs containing a single fault. Using the corpus of mutants, our aim is to measure the mutation score, i.e., the percentage of mutants that each test set identifies (kills). The test sets run on both the program and the mutant. The mutant is detected when the outputs of the original program and the mutant are different. In that case, we consider that the test set *strongly kills* the mutant. Similarly, we detect a fault when the buggy version and the fixed version of a program produce a different output. Here, we investigate how often an OutGen test set covers a real or artificial bug and produces different outputs in the buggy –or mutant– and fixed versions of a program, using various repositories.

RQ3.1: How effective is an OutGen generated test set at killing mutants and detecting faults in comparison with input diversity driven, CBMC, CAVM, behaviour diversity, and a pseudo-random generation? RQ3.2 How does a restricted time budget affect the performance and effectiveness of OutGen in comparison to the other methods?

We compare correlation information to measure the relative effectiveness of OutGen. This evaluation measures the mutation score of each approach and the overall ability to detect bugs in different software repositories. For comparison reasons, we include a pseudo-random generator, which is our random baseline, CAVM [?] –the state of the art search-based unit test generation for C–, a behaviour diversity whitebox technique based on the Normalized Information Distance of system traces [?], and the solver generation of

inputs based on the constraints produced with CBMC. We also measure time and memory consumption of the tools.

For the second part, we perform a similar evaluation providing the same amount of time to every tool and measure mutation scores and fault detection. The amount of time provided to each tool/method was 20 minutes, as this should give random testing enough time to be competitive with the other tools.

6 Experimental setup

This section discusses the dataset and evaluation details for the evaluation of OutGen.

We created our first experimental dataset by collecting a set of 8 functions from the R-project[2] [?] written in C.

These functions are statistical functions that compute floating point values on probability distributions.

They are affected by 5 real bugs. These bugs were reported to the R-bugs repository[3] during 2015 and 2018. They had not been identified for at least 6 years after these functions were created. They affect specific behaviours of the floating point arithmetic of the functions. These were chosen from a total of 8 bugs identified in the R-project subversion logs. The other 3 bugs were discarded as they affect memory consumption and not input/output behaviour. We generated the constraints for the fixed and the buggy version of the functions, respecting their dependencies in the specific repository version. The average number of dependencies for each function are 25 files (14 files without headers), the average function size is 1545 LoC. For reproducibility purposes, we created a public repository from this data[4].

We also used the `tcas` program extracted from the Software-artifact Infrastructure Repository (SIR) [?]. This program is an aircraft collision avoidance system composed of 135 LoC and works with integer inputs and outputs. The repository includes 41 seeded errors.

Finally, we included programs from the Codeflaws[5] [?] (CF) repository. This is composed of 3902 defects from 7436 C programs. These programs originate in the Codeforces[6] competition, a web page setting C programming contests for professional programmers. For our experiments we chose 100 defects from the numerical programs (integers and floating point inputs and outputs), uniformly at random. We did not consider any programs using pointers. This repository provides real programs, each including a program with a single defect and its fixed version. The average size is 50 LoC (total LoC 35654). These programs are of a size comparable to unit testing.

The evaluation itself has four parts. The first part is the uniformity evaluation, performed with the L2-test [?]. For the first part, we created as many tests as the L2-test requires to measure distance to a uniform distribution. The second part determines how output diversity improves the strength of the effect of output uniqueness. In this part, our experiments compare output diversity driven generation with input diversity driven generation. Based on the work of Alshahwan and Harman [?], we want to measure how our output diversity driven test set generator (OutGen) affects the Spearman rank correlation coefficients compared to an input diversity driven generator, such as the one presented by these authors. As Alshahwan and Harman's work is a methodology and not a tool by itself, we need to use a diverse input generator to reproduce their experiments, for that reason we use XORSample'. The next part measures the effectiveness in terms of mutation score and fault exposure, and the last part the performance of the test sets when limiting the generation time.

For the second and third parts, the test pool size is 500 per program. We create pools for XORSample' and OutGen in the second part and also for CBMC (with loop unwinding of 20, s value of 3 and default parameters for the rest), CAVM (using default parameters), the behaviour diversity method based on NCD [?], and a random baseline, for the efficiency comparison. For the NCD method, we generate a set of 1,000 test cases using the random baseline. Then, for each test case, we extract the program trace using gcov. This trace contains information about the control flow graph, method calls and number of times each part of the program is visited. This serves as the VAT model (VAriability of Tests) [?]. Then, NCD acts as the Universal Test Distance, creating a pairwise comparison among the test cases. After, following the methodology of [?], we create a hierarchical clustering of the tests' traces, and select the number of clusters as the test set size. Using the dendrogram, we discriminate the clusters. One element of each cluster is selected uniformly at random to create the final test set.

From the test pools, we select 20 test sets and investigate the rank correlations of line coverage, branch coverage, path coverage, mutation score and percentage of faults detected with test set size and output uniqueness (in the same way as Alshahwan and Harman [?]). Each test set has a size ranging between 10 and 500, and they are selected uniformly at random without repetitions from the pools. This variation on size is to avoid the effect of size on the evaluation, as remarked by Papadakis et al. [?]. Every experiment was repeated 30 times per program and the distribution for the former measures are reported as medians. The last experiment compares the generation performance and effectiveness when restricting the amount of time for each generation tool to 20 minutes. Similarly to the previous experiment, we produce 30 repetitions per tool and evaluate the mutation score and the faults detected of the generated test sets.

7 Experiments

We show that OutGen generates test sets whose output distributions are close to uniform, improving the output uniqueness measure of a test set. We evaluate the quality of the generated test sets in terms of uniformity, output uniqueness score, coverage, mutation score and fault detection, in the context of ODD and compare them with IDD, CBMC, CAVM, the behavioural diversity method based on

2. https://www.r-project.org
3. https://bugs.r-project.org
4. https://www.dropbox.com/s/2iqchfic2ougd5x/dataTSE.zip?dl=0
5. https://codeflaws.github.io
6. http://codeforces.com

Table 1
Percentage of test sets generated with OutGen that pass the L2-tests. We compare the three repositories (R, SIR and CF) for different values of ϵ^2, to evaluate how close we are to the uniform distribution.

Functions	0.1	0.05	0.01
R-functions	100.0%	87.5%	75.0%
SIR	100.0%	100.0%	100.0%
CF	100.0%	98.0%	97.0%

NCD, and a pseudo-random generator. Finally, we discuss the limitations of OutGen.

7.1 Measuring OutGen Uniformity

The L2-test requires the generation of enough witnesses to guarantee that we achieve a given confidence level (usually 95%) for the statistic. Following the study of Diakonikolas et al. [?], we use their Lemma 5 to bound the number of samples (m) as $m \leq 6n^{1/2}/\epsilon^2$, where n is the domain size and ϵ^2 the proximity to uniformity. For this reason, we run our evaluation on three levels of ϵ^2: 0.1, 0.05 and 0.01, generating an average of 6,000, 12,000 and 60,000 tests respectively for each program, allowing repetitions in the output sampling process.

Table 1 shows the proximity of the sampled outputs to the uniform distribution, according to the L2-test. For CF, the diversity of outputs is very close to the uniform distribution (97% of the programs passed the test when $\epsilon^2 = 0.01$). This also happens with the SIR repository. For the R-functions, a 75% of them are 0.01 close to a uniform distribution, which is still a strong percentage considering that all are 0.1 close. We observed that the results are good for large output domains and worse for small ones; however, in small domains we are able to reach the maximum output uniqueness score.

RQ1: *OutGen is uniform for large domains and near-uniform for domains with low channel capacity.*

7.2 Improving Output Uniqueness Properties via OutGen

We start by measuring the output uniqueness score of the output and input diversity driven test pools generated, as described in Section 6. These two test pools are formed by 500 tests per program. OutGen aims at improving the score. This will show clearer correlations between output uniqueness and other metrics, such as structural coverage, mutation score and faults detected. The average output uniqueness of the ODD pool for all programs was 28.92 and for IDD pool was 19.23. On average, the ODD test sets have a score value that is 50.39% higher than IDD.

Having a clearer vision of output uniqueness, we evaluate its correlation with structural coverage. Figure 3 shows the coverage results for input and output diversity driven generated test sets. The tool `gcov` measures the coverage in terms of lines covered (L), branches covered (B) and paths taken (P). We firstly compare the correlations between the size and output uniqueness with the structural coverage metrics. For ODD, the figure shows that size has a similar correlation with line (0.596), branch (0.521) and path coverage (0.622). IDD has similar results, but the values are slightly higher (line is 0.671, branch is 0.575 and path is 0.695). This is really important because the coverage metrics in ODD are less dependent on size than in IDD. Output uniqueness correlates better than size with every single coverage metrics, specially with line coverage. These results are similar to those discovered by Alshahwan and Harman [?].

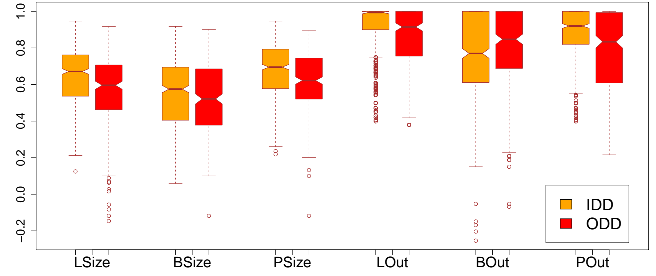

Figure 3. Spearman's rank correlation for all the test sets generated with IDD (left) and ODD (right). Each plot compares line, branch and path coverage with size and output uniqueness. The distribution corresponds to all the experiments.

Table 2
Spearman's rank correlation for all the test sets generated with input (IDD) and output diversity driven (ODD). Each row compares the correlations of size and output uniqueness with mutation score and faults detected. The distribution corresponds to all the experiments. The ▲symbol shows that there is a statistical significant improvement according to the Wilcoxon's test (p-value < 0.05) between output and input diversity driven test cases.

Correlations	IDD	ODD
Size - Mut	0.541 ± 0.233	0.562 ± 0.211
Out - Mut	0.512 ± 0.189	0.551 ± 0.286
Size - Fault	0.478 ± 0.169	0.435 ± 0.151
Out - Fault	0.611 ± 0.234	▲0.997 ± 0.302

The output uniqueness correlation differs depending on the coverage metrics and generation method. We can see that, when the test sets are output diversity driven, the line correlation (0.915) is better than branch (0.847) and path (0.834). When the test sets are input diversity driven, the correlation is also better for line coverage (0.993) than for path (0.820) or branch (0.770) coverage. There are interesting improvements in the coverage correlations when the test sets are input diversity driven; nevertheless, these improvements only affect line and path coverage.

For the correlations with mutation score, we have created up to 100 mutants for each program with the aim of evaluating how many mutants the test sets are able to kill. All mutants are generated using Milu [?] with the default parameters. We consider the original program as the oracle and the mutant as a buggy version of the program. When the test set distinguishes them by generating a different output, we consider that the mutant is killed. For the defects, we consider that an error is found when the output differs between the fixed and the buggy program.

Table 2 (top) shows the Spearman's rank correlation of the mutation score with the test set size and the output uniqueness. The left side shows the results for IDD. The results show that neither the test set size (0.55) nor the output uniqueness (0.51) have strong correlations with the mutation score for the IDD test sets. Comparing these results with the ODD test sets, we can see that they are similar for test set size (0.56) and output uniqueness (0.54) correlations.

Table 2 (bottom) shows the correlation results for the faults detected by the test sets. On the left side, the table reports the results for IDD comparing the correlations between detecting faults with size and outputs. The best metric is output uniqueness (0.61). Size (0.48) obtains less correlation as was originally reported by Alshahwan and Harman [?]. The right side of the table shows significantly different results. Output uniqueness has a high correlation (0.997) with fault detection, while size keeps a low correlation (0.44). These results show that an ODD generated test set improves the correlations for output uniqueness by 63%, while the size correlation, connected with the inputs, is less significant. This improvement shows that output diversity driven test sets can exploit the advantages of output uniqueness better than input diversity driven test sets and, as a consequence, they will detect more bugs.

In order to evaluate whether there is statistical significance on these correlation results, we applied the Wilcoxon test [?] to the result distributions, following the guidelines of Arcuri and Briand [?]. We consider the p-value for the test to be smaller than 0.05 in order to consider that the test is passed. The results show that there is only statistical improvement on the fault detection abilities.

RQ2: *Output diversity driven test sets improve the output uniqueness score of the test set by 50% on average compared to input diversity driven test sets. Although both techniques correlate in a similar way with structural coverage and mutation score, ODD excels producing a strong correlation between faults detected and output uniqueness, reaching an up to 63% improvement.*

7.3 Evaluating OutGen's Efficacy and Performance

We measure efficacy in killing mutants or detecting bugs on the total number of mutants generated and bugs considered. We have also generated tests with CBMC constraints, CAVM, behaviour diversity (NCD) and a pseudo-random generator to compare efficacy with OutGen. During the experiments, CAVM produced out-of-memory for 52% programs of Codeflaws (the results highlighted with * in Table 3 are restricted to the programs where the tool worked). Constraining ODD to the set of programs where CAVM works, the results are almost the same as for the general ones. CAVM has type compatibility limitations that do not allow it to generate inputs for the R-function. It produces errors when it needs to work with abstract identifiers such as 'Infinite' or 'not-a-number (NaN)', as is the case for floating points functions.

Table 3 (top) shows the median mutation score for every approach during the experiments.The pseudo-random generator, behaviour diversity and IDD have similar results on CF and the R-functions (84% and 46%, respectively). CBMC achieves better results on SIR (37%), similar for R-functions (46%), and worse for CF (69%). It is significant that for SIR the random generator and the behaviours-based one are only able to kill 1% of the mutants. ODD gets the best results in all the cases, close to CAVM for CF (90% for CAVM and 92% for ODD), and showing a significant improvement for SIR (59%). With respect to the second best technique, we get an improvement of 2% in CF, 57% in SIR and 29% in R. With respect to the worst, we reach an improvement of 33% in CF, 4150% for SIR and 30% for R. All the results have been tested according to the Wilcoxon test [?], and ODD is significantly better in all the cases but on the comparison between CAVM and ODD for codeflaws. The test is passed considering a p-value smaller than 0.05.

Table 3 (bottom) shows the bug detection results. The pseudo-random generator, behaviour diversity and IDD normally achieve the second position (for CF the random baseline reaches a 67%, and, for SIR, IDD reaches a 43%), while CBMC gets worse results in CF (49%) and SIR (17%). Again, the pseudo-random generator and behaviours diversity have problems in identifying the bugs from SIR (5%). In every experiment, OutGen gets the best results (70% for CF and 52% for SIR). The improvements with respect to the second are 3% for CF and 21% for SIR, while they are 42% for CF and 963% for SIR w.r.t. the worst. It is important to remark that none of the tools are able to find the real bugs in the R functions except OutGen (11%). These errors are related to specifics of floating point types. Our system is designed to work with abstractions of infinite, -infinite and not-a-number (NaN) as possible input/output values and the solver theory can reason about them, because we use the bitvector arithmetic theory with mixed abstractions for floating point (Section 7.1). IDD and CBMC use the same theory, however, Rand, CAVM and NCD can not use it because they do not use solvers. Considering that these bugs were not identified for the first 6 years of their existence, this makes ODD a human competitive approach, opening new possibilities for detecting difficult floating point bugs.

Similarly to the former case on mutation score, we applied the Wilcoxon test to evaluate whether the improvement for ODD is significant and we discover that it is in all cases except for the comparison with CAVM for codeflaws.

As regards the generation performance of the techniques, Table 4 shows that IDD and ODD perform worse than random and CBMC. This is a consequence of their dependence on CBMC to generate constraints. Nevertheless, they perform significantly better than CAVM, the state of the art search-based generator, especially for memory consumption and compatibility with different types of complex programs.

RQ3.1: *Output diversity driven test sets perform better at detecting mutants than CBMC, CAVM, the pseudo-random generator, behaviours diversity and input diversity driven. They improve the mutation score up by 4150%. For finding bugs, OutGen performs up to 963% better. Moreover, for the R-functions, it is the only technique that can detect bugs, detecting bugs on the 11% of the test sets. However, CBMC and compression based techniques are slower to generate test sets than random or pure CBMC.*

Table 3
Mutation score (Mut) and bugs detected (Defs) for the pseudo-random generator (Random), CBMC, CAVM, input (IDD), behaviour (NCD) and output diversity driven (ODD). Each row compares the median mutation score and bugs detected from the whole set of test sets generated for each tool. It breaks down by repository: the R-functions (R), SIR and CodeFlaws (CF). The (*) represents a reduced set of programs. The ▼symbol shows that the results of each technique are worse than ODD with statistical significance according to Wilcoxon's Test (p-value < 0.05).

Data	Rand	CBMC	CAVM	IDD	NCD	ODD
Mut R	▼46.0%	▼46.2%	-	▼46.5%	▼46.2%	**60.0%**
Mut SIR	▼1.4%	▼37.8%	▼4.1%	▼22.0%	▼1.5%	**59.5%**
Mut CF	▼84.0%	▼69.2%	*90.4%	▼84.1%	▼83.9%	**92.1%**
Defs R	▼0.0%	▼0.0%	-	▼0.0%	▼0.0%	**11.0%**
Defs SIR	▼4.9%	▼17.1%	▼12.2%	▼43.2%	▼5.1%	**52.1%**
Defs CF	▼67.1%	▼49.5%	*68.3%	▼52.5%	▼66.9%	**70.2%**

Table 4
Time and memory consumption for the pseudo-random generator (Random), CBMC, CAVM, input (IDD), behaviour (NCD) and output diversity driven (ODD). Each row compares the median time and memory consumption from the whole set of test sets generated for each tool. It breaks down by repository: the R-functions (R), SIR and CodeFlaws (CF). The (*) represents a reduced set of programs.

Data	Rand	CBMC	CAVM	NCD	IDD	ODD
Time R	29 s	16 m	-	21 m	52 m	19 m
Time SIR	31 s	2 m	6 m	4 m	11 m	4 m
Time CF	35 s	31 m	*28 m	39 m	63 m	35 m
Mem R	42 M	97 M	-	155 M	3.1 G	171 M
Mem SIR	38 M	61 M	130 M	113 M	170 M	111 M
Mem CF	32 M	56 M	*128 G	110 M	2.5 G	122 M

7.4 Comparing OutGen's Scalability

Comparing performance in a fixed time, in this case 20 minutes, we compare the results for all the different generation methods. Table 5 shows these results. Similar to the previous results in Table 3, ODD dominates the detection of faults in every repository and it is still the only methodology that can identify bugs in the R functions. It is important to remark that the random sampling increases the number of inputs generated by three orders of magnitude with respect to the previous experiment (from 500 to 500,000) and it still struggles to detect particular bugs such as the ones in SIR or the R-functions. Also, the execution time of these large randomly generated test sets is significantly higher (from 5 milliseconds, on average, to 10 seconds). For the rest of the tools the performance is similar, but for IDD performance reduces drastically on CF and the R functions for the mutation score and the fault detection rate for CF.

Clearly, in certain cases random testing does better than OutGen, given the same time budget: this likely happens when the inverse images of outputs for the program or function are well balanced, i.e. roughly the same size. In cases where they are highly unbalanced, this cannot be the case (these issues and related examples are also discusse in Sections 6-7 of Boreale and Paolini [**?**]).

> **RQ3.2**: *Output diversity driven test sets, generated with the same amount of time, perform better at detecting mutants and faults than the other techniques. They improve the mutation score by up to 1158%. For finding bugs, OutGen performs up to 728% better. Moreover, for the R-functions, it is the only technique that can detect bugs, detecting bugs with 11% of the test sets.*

7.5 Threats to Validity

During these experiments, we have identified two potential threats to validity: the solver and the definition of diversity. The solver limits the program size (in terms of scalability) and forces us to apply the approach in a unit testing way for bigger functions, such as the R-functions. This limitation is a consequence of the complexity of the constraints. The solver has difficulty solving some constraints such as MD5 hashes. This current limitation does not allow us to apply our tool to a project of the size of the R-project, for instance. We want to extend it to bigger programs in the future. For this reason, our plan is to focus on memory models [**?**], so we can consider the whole output buffer for OutGen. On the other hand, our notion of diversity is based on the uniform distribution. Without the spread generated during the cell selection process (Section 4), this might generate redundancy between outputs, with respect to any notion of similarity.

Other external threats are: the limitations of CBMC, mostly related to the loop unwinding process, and the SMT solvers limitations to deal with complex programs. The main problem related to the loop unwinding is bug masking, i.e., the error might not manifest itself in the approximation of the program produced by the constraints. Another relevant limitation of CBMC that affects our generality is the conversion between floating point and integer which is currently not implemented in the tool. Nevertheless, our methodology can be extended to other languages if the analyst can extract the program constraints from them. It is important to remark that our methodology does not need to discard tests for improving diversity on outputs, as output-uniqueness [**?**];

Table 5
Mutation score (Mut) and bugs detected (Defs) for the pseudo-random generator (Random), CBMC, CAVM, input (IDD) and output diversity driven (ODD), using the same amount of time (20 minutes) for every generator. Each row compares the median mutation score and bugs detected from the whole set of test sets generated for each tool. It breaks down by repository: the R-functions (R), SIR and CodeFlaws (CF). The (*) represents a reduced set of programs. The ▼symbol shows that the results of each technique are worse than ODD with statistical significance according to Wilcoxon's Test (p-value < 0.05).

Data	Rand	CBMC	CAVM	IDD	ODD
Mut R	▼55.5%	▼46.8%	-	▼28.6%	**61.0%**
Mut SIR	▼5.3%	▼42.6%	▼6.8%	▼34.5%	**66.7%**
Mut CF	▼86.6%	▼62.4%	▼*85.1%	▼55.2%	**88.7%**
Defs R	▼0.0%	▼0.0%	-	▼0.0%	**11.0%**
Defs SIR	▼6.8%	▼23.5%	▼16.9%	▼45.1%	**56.3%**
Defs CF	▼68.2%	▼41.3%	▼*67.2%	▼33.1%	**69.9%**

furthermore, the faults in our program corpus are real faults rather than seeded.

8 Related Work

The method we present is an automatic test set generation method based on output diversity. This section discusses our innovations with respect to the state of the art, focusing on automatic test set generation, diversity of tests sets, and output diversity or uniqueness.

8.1 Automatic Test Generation (ATG)

Generating proper test sets with no human intervention is an open problem. As per the Anand et al. study [?], the main areas contributing to automatic test generation are: symbolic execution, model-based, combinatorial testing, adaptive random testing and search-based testing. These techniques aim at achieving some coverage criteria. Shamshiri et al. showed that, overall, coverage-based techniques only find 55.7% of their corpus faults, while, individually, only 19.9% of coverage based test sets expose a fault [?]. One problem is a faults' failure to propagate to observable outputs. Generating inputs with a higher output diversity propagates the behaviour to observable outputs better than coverage techniques, as we show in Section 7.4.

Indeed, this problem affects several ATG strategies. Some examples are search-based tools (like CAVM [?] or OCELOT [?]) and model checkers with testing abilities (like CBMC [?]). These techniques lack diversity during the test generation process, which is the main requirement of our automatic generation tool. Our inspiration parts from the idea of testing a specific program point. Alipour et al. [?] and Gotlieb and Petit [?] combined this idea with search algorithms and solvers, respectively. Their aim was to increment the number of generated tests passing through a specific program point, or their probability. In analogy with our case, that point has every output. We consider all potential outputs for a function and unify them in terms of constraints (Section 4). Nevertheless, this output extension alone does not alleviate the problems of coverage. Output *diversity* is needed in the test set generation process.

8.2 Diversity

Diversification parts from the assumption that every test is equally likely to activate or detect a bug [?]. In the literature there is much work aimed at diversifying test sets [?], [?], [?], [?], [?], [?], [?], [?], or using diversification as a filter [?], [?], [?]. In this area, Chakraborty et al. generated a methodology to create uniform inputs for testing electrical circuits. They modelled the circuit as a conjunctive normal form (CNF) formula and applied a SAT solver to generate inputs [?], [?]. This approach needed to face up to the adversarial behaviour of the solver and that in turn forced the authors to add universal hashing functions to improve the spread of witnesses on the input space. They extended the XORSample' algorithm [?] with the BGP algorithm [?] to achieve this goal. The authors also extended their methodology to SMT solvers [?] but with the aim of creating a model counter. Dutra et al. [?] presented a competitive adaptation of Chakraborty et al.'s sampling method based on fuzzy testing. In our case, we extend it to deal with real programs using bit-vector arithmetic. Similarly to Chakraborty's, our approach generalizes through the $\mathcal{H}_{xor}$ hash family, but their approach focuses on input diversification. However, highly distributed inputs might not activate all the behaviours of the programs: it is important to include the semantics of the program in the diversification. For this reason, we also include the program behaviour, through adding extra constraints in the generation process.

A natural methodology of diversification is adaptive random testing [?], which diversifies the inputs using a random search, but it does not necessarily include semantics during the generation process. Other improvements combine coverage and diversity, for example, Fraser and Arcuri [?] extended the search process of EvoSuite to a multi-objective approach, where each objective is a coverage criterion.

There is no concrete measure of diversity. Some authors proposed to apply metrics from information theory. Those with better results used the normalized compression distance (NCD) [?], and entropy [?]. These metrics aim at improving the test set diameter [?], that describes, in terms of the metric, the average differences among the tests inside the test set. Other authors use search based approach to select diverse test sets based on their behaviours, using the information from the execution trace, specially function calls [?],

[?].

In our case, we consider diversity in terms of uniformity and entropy, because every search for diversity leads to the uniform distribution that is considered in Information Theory as the most informative distribution for sampling purposes [?], [?]. Entropy and uniformity are connected, because entropy is maximal when the data distribution is uniform [?]. However, measuring uniformity is itself challenging and it is the main evaluation weakness that we identified in the related work [?], [?], [?], [?], [?]. This evaluation requires statistical tests. Our approach includes it (Section 7.1). We studied different methodologies for evaluating the output distribution in order to identify whether our technique is generating uniform witnesses or how close we are to a uniform distribution. We studied statistical tests for continue uniform distributions [?], but they all assume that there are no sequence gaps between output values, which is not the case. We also researched discrete statistical tests [?], [?], [?], [?], [?], but they suffer from the same limitations. Only collision-based test statistics guarantee an appropriate evaluation for our scenario, and also give us a notion of distance [?]. The main problem with collision-based statistical tests is the high number of samples that are needed to guarantee the test confidence [?]. Nevertheless, our corpus allowed us to generate sufficient witnesses from the solvers and use them in the tests for uniformity.

Diversification can assist in dealing with some problems such as Failed Error Propagation [?], which is related to the entropy loss problem. Poulding and Feldt aimed at creating diversification by construction, applying a technique named Gödel testing [?]. Nevertheless, their approach relied on input driven diversity and they needed to create a manual test generator per program. Our technique uses the solver as a general purpose generator. Another application of diversification is test prioritization, as Henard et al. studied [?]. The authors discovered that the test set diameter approach performs better in a black box scenario, and they found little performance differences between black and white-box testing. In our work, we aim at improving the diameter of the output set, also known as the channel capacity of the output channel.

8.3 Output Diversity

The problem of using output diversity in ATG is open. One of the main problems identified by Alshahwan and Harman was the development of an automatic output diverse generator, such as the one that we present in the current paper [?], [?]. These authors developed an input diversity filter based on output-uniqueness, as a post-preprocessing step. They showed that their measure, output uniqueness, correlates better with fault detection than structural coverage. Nevertheless, the effectiveness depends on the definition of output. In their work, they define the output as textual or structural aspects of HTML code, and the difference as *not equality*. This makes the system too sensitive to small changes. Rojas el at. [?] introduced an output-uniqueness criterion in EvoSuite called output coverage. It aims at incrementing the number of different outputs during the search, but an output's selection probability depends on input sampling. In our system all possible outputs have the same probability for being chosen. This guarantees that rare outputs are as likely as frequent outputs to appear.

Matinnejad et al. studied output diversity in Simulink models [?]. Their main goal was to diversify the output signals produced by the automatic generated test sets. In their case, the outputs are distinguished by the shape of the signal output, introducing a notion of similarity. They showed that diversity on outputs improves the results of random testing using signal diversity in finding faults. Without this notion of similarity, it is not possible to reduce the cost function that the clustering algorithm needs. In order to be more general, we prefer to have a diversity notion that does not depend on a similarity definition, as this requires semantic knowledge about the output itself. Moreover, our new experiments, which introduce a generic notion of similarity, the Normalized Compression Distance, show that use of such with input diversity does not necessarily improve the output sampling process (Section 7.4).

All the previous methodologies, including ours, assume determinism on outputs. However, as Gao et al. studied [?], the output might be affected by three different factors: code, behaviour and external interactions (system configuration). For this reason, we fixed the system, compiler and tools interacting with the test sets, to reduce this effect.

9 Conclusions

Output diversity is a large and unexplored front in testing, but one with a huge potential. This work has opened new horizons by introducing an output diversity driven generation method that improves the output uniqueness score of the test sets. This optimization, aiming at maximizing output uniqueness for a given test set, outperforms input diversity on fault detection and mutation score. This is in part a consequence of adding semantic information to the generation process.

Although output diversity is extremely valuable for its fault finding abilities, it is not a replacement for coverage. Combining coverage and output diversity could further improve the detection of faults. Combining ODD with IDD is also an open problem, as is developing a search-based method to replace the adaptation of XORSample', perhaps using uniformity tests as fitness functions. FGödel Testing [?], a search-based generator, will be less deterministic than an SMT solver-based approach, but will be able to deal with bigger programs. There is a need for new methods for maximising output uniqueness, especially those using different sampling techniques. We want to evaluate the effects of clustered, stratified and systematic sampling, for instance, on the output uniqueness.

Acknowledgments

This work has been supported by the InfoTestSS EP/P005888/1 research project from EPSRC. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan V GPU used for this research. Many thanks to Robert Feldt for useful discussion on how to find a language to explain and present the novelty in this work.

Héctor Menéndez is a Lecturer at Middlesex University London, working on applications of information theory to software testing. Originally, he worked designing machine learning algorithms based on graph structures and search based optimization. He has applied these ideas to several different fields, where the most relevant are malware analysis, unmanned air vehicles and, currently, software testing.

Michele Boreale is associate professor of Computer Science (Informatica) at Università di Firenze, Dipartimento di Statistica, Informatica, Applicazioni, Scuola di Scienze. His previous tenure positions have been: assistant professor in the same School and assistant professor at the Università "La Sapienza", Rome (first appointment February 1996). He has received a Ph.D. degree in Computer Science from the University "La Sapienza" (October 1995), and a Laurea degree cum laude in Scienze dell'Informazione (Computer Science) from the University of Pisa (February 1991).

Daniele Gorla received a degree in Computer Science from the University of Rome La Sapienza in December 2000. He then obtained a Ph.D. in "Computer Science and Applications" at the University of Florence in February 2005, under the supervision of prof. Rocco De Nicola and Rosario Pugliese. Since March 2006 he is a researcher, and now an associate professor, at the Department of Computer Science of the University of Rome "La Sapienza". His research focuses on formal methods of computing, in particular applied to programming languages and concurrency systems.

David Clark is a Reader in Software Engineering at University College London. His research interests include Software testing, Application of Information Theory to software analysis, Program flow security, Slicing programs and software models, Malware detection and classification. David has published articles on a wide range of topics, including disrupting android malware triage by forcing misclassification, quantifying the diversity of sets of test cases, and test oracle assessment and improvement.