PopArt: Ranked Testing Efficiency

Iason Papapanagiotakis Bousy, Earl T. Barr, David Clark

Abstract—
Too often, programmers are under pressure to maximize their confidence in the correctness of their code with a tight testing budget. Should they spend some of that budget on finding "interesting" inputs or spend their entire testing budget on test executions? Work on testing efficiency has explored two competing approaches to answer this question: systematic partition testing (ST), which defines a testing partition and tests its parts, and random testing (RT), which directly samples inputs with replacement. A consensus as to which is better when has yet to emerge. We present Probability Ordered Partition Testing (POPART), a new systematic partition-based testing strategy that visits the parts of a testing partition in decreasing probability order and in doing so leverages any non-uniformity over that partition. We show how to construct a homogeneous testing partition, a requirement for systematic testing, by using an executable oracle and the path partition. A program's path partition is a naturally occurring testing partition that is usually skewed for the simple reason that some paths execute more frequently than others. To confirm this conventional wisdom, we instrument programs from the Codeflaws repository and find that 80% of them have a skewed path probability distribution. POPART visits the parts of a testing partition in decreasing probability order. We then compare POPART with RT to characterise the configuration space in which each is more efficient. We show that, when simulating Codeflaws, POPART outperforms RT after $100{,}000$ executions. Our results reaffirm RT's power for very small testing budgets but also show that for any application requiring high (above 90%) probability-weighted coverage POPART should be preferred. In such cases, despite paying more for each test execution, we prove that POPART outperforms RT: it traverses parts whose cumulative probability bounds that of random testing, showing that sampling without replacement pays for itself, given a nonuniform probability over a testing partition.

1 INTRODUCTION

Programmers test programs to check correctness, find bugs and prevent their recurrence. Testing is inherently under-approximate and resource-bound. As a result, testing only exercises a small proportion of the behaviours of almost all programs, especially those found in industry, because most programs have a vast number of behaviours. Testing efficiency is therefore key to improving software quality. Testing has two key costs: finding inputs that maximize a tester's confidence and executing the tested program.

A testing campaign selects an input, executes the subject under test (SUT), until it exhausts its testing budget. Some tests executed during the testing campaign are redundant because they exercise the same program behaviour, and thus provide no new information. A testing partition evaluates the results of the campaign, grouping redundant executions. Testing must execute its subject; it can, however, minimise the cost of input selection by randomly sampling inputs, usually uniformly with replacement. This strategy is called random testing (RT). The alternative is to spend some of the test budget to sample inputs from the testing partition without replacement, avoiding redundant tests; this strategy is called systematic partition testing (ST). Intuitively, RT wins when it rarely executes redundant or low-value tests; otherwise, ST wins. Researchers have debated the question of whether and when RT is more efficient than ST, or vice versa, since the early 1980s [1], [2]. Empirical work was contradictory; theoretical work in 1990s favoured ST [3], [4], [5].

In 2014, Böhme and Paul introduced UST, a fault-revelatory partition strategy, which they claim to be ideal, that uniformly samples the testing partition and therefore its input coverage increases linearly, in expectation. They showed that, under any realistically feasible test budget, RT is more efficient than UST. Although Böhme and Paul do not invoke it, the principle of indifference [6] justifies their assumption: Given no additional knowledge about a population, the best option is indeed to sample it uniformly.

Knowing the ground truth probability distribution over a program's inputs is a hard epistemic problem and is, in general, unknowable. Worse, this distribution interacts with any testing partition defined over the inputs, making the probability distribution over parts even harder to know. That said, it is highly unlikely that the distribution arising from their interaction is uniform. The uniform distribution is a special case of a probability distribution; its measure is zero against the space of all probability distributions.

So, while we cannot know an arbitrary program's likely behaviours, we *do* know that some behaviours are more likely than others; *i.e.* they are unlikely to be uniformly distributed. Key to maximising a tester's confidence is executing inputs that matter within the test budget. Inputs that a program is most likely to see in deployment are those that exercise paths with the highest probability weight. Domain experts often have an intuition about which program inputs and behaviours are more frequently observed; testers can use elicitation techniques [7] to capture these intuitions to create a probability distribution that prioritises the test of frequent or critical inputs.

We present probability-ordered partition testing (POPART), a novel systematic partition testing method that drops the uniformity assumption over its testing partition in order to achieve higher testing efficiency, outperforming UST. POPART can work with any *homogeneous* testing partition, that is, a testing partition that does not assign to the same part both failing and passing input values with respect to an oracle. In Section 3, we show how to construct such a partition using an executable oracle and a

subject program's program paths, which naturally arise in imperative programs and align with, because they define, program behaviour. POPART builds the oracle-defined path partition of the program under test and ranks the parts with respect to their probability-weighted input coverage. It then explores the parts in decreasing probability order, under a resource bound. As a systematic partition testing technique, POPART samples the testing partition without replacement.

We then turn to the comparison of POPART with RT. First, in Section 3, we analytically compare the two methods, defining the *break-even* point, the time at which their testing efficiency becomes equal, then prove that, beyond this break-even point, POPART's efficiency is higher. Then, Section 4 presents the results of a simulation-based comparison that both confirms our theoretical results and explores the effect that different input parameters, such as the testing budget, the relative cost of POPART with respect to RT's execution and the testing partition probability distribution, have on the testing efficiency of both methods.

The results of the comparison show that each testing method has its place in a practitioner's toolkit and the comparison characterises the space in which each is best. One intuitive view of entropy in information theory is as a measure of how flat a distribution is: maximal entropy is, under this view, maximally flat. By indexing probability distributions by their entropy to measure their "flatness", we show that as the entropy of the testing partition decreases, the *break-even* point is reached exponentially faster. Indeed, when part probabilities are sufficiently different (*i.e.* testing partition with a skewed probability distribution) and high probability-weighted coverage is required, POPART has an exponentially smaller cost than RT. For non-uniform probability distributions, POPART outperforms RT when the coverage goal exceeds 90%, otherwise RT wins (Figure 7). The existence of such skewed probability distributions have been the subject of research on *e.g. hot paths*. In Section 2, we perform a case-study over programs from the Codeflaws repository [8] to confirm that indeed 80% of sampled programs induce a highly skewed path probability distribution, best approximated by a decreasing exponential function. When we simulate code to match Codeflaws, it takes fewer than $100{,}000$ RT executions to reach POPART and RT's *break-even* point (see fig. 5). Our theoretical and simulation-based comparisons of POPART and RT both reaffirm the superiority of random testing when testing budgets are small, high coverage is not needed, or when dealing with near-uniform testing partitions.

Even when the test budget is highly constrained and high coverage is not required, however, POPART can still be useful. Its ability to work directly over the testing partition allows ranking of parts using a different metric such as *importance*. For example, a physics simulation may be designed to work best on certain input ranges, some of a program's inputs might contain sensitive data, or the tester might just be more interested in specific values (as when bug fixing) regardless of the probability in deployment. Like with the input probability distribution, learning input importance in deployment is also a hard problem which requires domain experts to approximate.

Our key contributions follow:

- We present Probability Ordered Partition Testing (POPART), a new systematic partition testing strategy that leverages the non-uniformity of homogeneous testing partitions.
- We prove that combining an executable oracle with a program's path partition creates the smallest testable *homogeneous* testing partition.
- We present both analytical and simulation-based comparisons of POPART with RT, showing that when simulating Codeflaws, POPART outperforms RT after $100{,}000$ executions and, more generally, POPART outperforms RT on skewed distributions when the probability-weight input coverage target exceeds 90%.

POPART's implementation, and the scripts and data used to evaluate it, can be found at 10.6084/m9.figshare.18544298.

2 MOTIVATING EXAMPLE

Not all paths are created equal; in almost all programs, some paths execute more often than others. In a popular program, a path that implements an important feature will execute more often than an error-handling path. This has been subject of research trying to optimize code based on the frequency of its execution and focus their resources on *hot paths* [9], [10]; branch prediction is built into CPUs to exploit them [11]. In the field of reliability engineering, Musa [12] argued that software testing should leverage such usage profiles to prioritize the discovery of likely-to-occur bugs.

One reason some paths execute more often than others is the fact that not all of a program's inputs are created equal: in the field, a program encounters some of its inputs far more often than others, as dictated by the probability distribution over its inputs, or usage profile. Researchers rarely study this source of hot paths, since it requires gathering telemetry from programs in the field or domain expertise, and resort instead to uniformly sampling inputs.

Another reason for hot paths is program structure: some paths simply handle more inputs, which makes them frequent under many input distributions, notably the uniform. To quantify this second source of non-uniformity in path execution frequency, we sampled 10 programs from Codeflaws, a curated corpus of real programs with real bugs [8]. Using PIN [13], we instrumented these programs, not their runtime or libraries, to output their decisions at each conditional jump. Codeflaws programs take integers and strings as inputs. To generate a string, we uniformly select a length modulo a bound, then uniformly fill it with characters. We executed the instrumented programs $100{,}000$ times (*ca.* 20 hours) on uniformly sampled inputs and harvested their paths.

The resulting path probability distributions are far from uniform. Inline with the conventional wisdom, all tested programs followed the pattern of some paths being significantly more likely to execute than others. Figure 1 shows the distribution of the estimated probabilities for two programs. To approximate their distributions, we sorted the empirical path probabilities in decreasing order and used curve fitting [14]. For the curve fitting process we used the following functions: 1. *exponential* $f(x) = a \cdot e^{bx}$, 2. *power* $f(x) = a \cdot x^b$, 3. *polynomial* $f(x) = a \cdot x^2 + b \cdot x + c$ and 4. *logarithmic* $f(x) = a \cdot \ln(x) + b$. An exponentially decreasing (negative exponent) function best fits 9 of the 10. The last

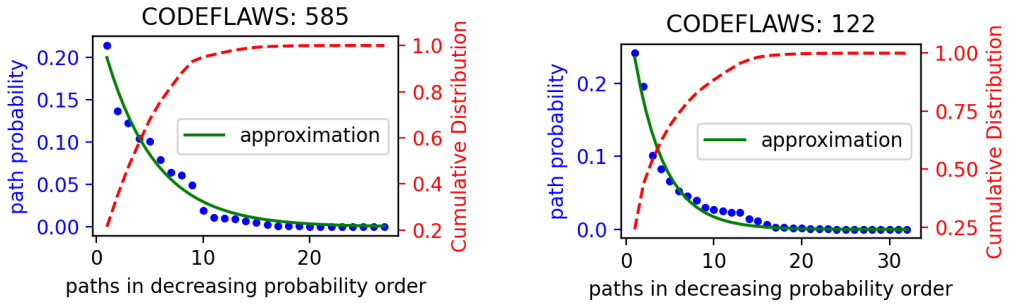


Fig. 1: The empirical probability distributions over the paths of two Codeflaws programs (585 left and 122 right), when executed on uniformly sampled inputs. Both are best approximated by an exponentially decreasing function.

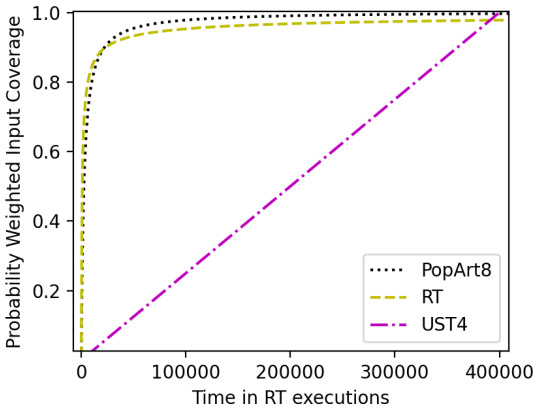


Fig. 2: Unlike UST, POPART is a systematic testing technique able to capitalize on skewed probability distributions over testing partitions. This allows POPART to match RT and overtake it after $\approx 26k$ test executions despite a $\times 8$ slowdown.

program had only 2 paths, so we excluded it. The mean normalised entropy[1] of the approximate path probability distributions is 0.69. None were uniform.

Figure 2 shows how RT, a systematic testing approach, which we call *uniform systematic testing* (UST), that uniformly selects parts without replacement, and POPART cover the probability-weighted input space over time. The three testing methods are simulated on a skewed testing partition with 100,000 parts that have an normalised entropy of 0.69. Here, we use time to abstractly represent a test budget and normalise execution times such that it takes unit time for RT to execute one test, 4 units of time for UST and 8 units of time for POPART[2]. By sampling parts uniformly, UST's expected coverage of the input space grows linearly. In contrast, both RT and POPART both prioritise high probability parts, with RT sampling them with replacement while POPART visits them in decreasing probability order (no replacement).

As a program's input space and the size of a testing partition grows (*i.e.* the number of parts), even a large tech company's testing budget cannot cover all of them. Paul and Böhme argued [15] that the point where RT's line intersects UST's is well beyond a realistic testing budget. We agree with them; assuming the execution of a test takes 1 second, it would take ≈ 4.6 days of CPU time for UST to overtake RT in this example. In contrast, it takes ≈ 7 hours for POPART to become more efficient than RT, a manageable testing effort (*e.g.* can run overnight). Furthermore, Figure 2 shows that even when POPART has lower probability weighted input coverage than RT, it is not far off. Combined with the fact that POPART exactly measures its coverage and does not rely on an expected value as does RT, could result in POPART being preferred in certain scenarios (*e.g.* when exact lower bounds of coverage are required for certain reliability assessment). The result of comparing testing methods varies by changing the parameters such as the size of the testing partition, the probability distribution over it the relative cost of each testing method. Section 4 explores how these variables affect the output of such comparison.

3 SYSTEMATIC TESTING WITH POPART

Probability Ordered Partition Testing (POPART, PA for brevity) is a systematic partition testing method aimed at maximizing probability-weighted input coverage under a resource budget. It visits a testing partition's parts in descending order of their probability weight.

For POPART to work, it needs a testing partition that is aligned with an oracle. We discuss how such a partition, called *homogeneous*, can be created using an executable oracle and the path partition. Then, we show how a probability distribution over a program's input values induces a probability distribution over any testing partition and how the probability-weighted testing partition can be used to assess the testing efficiency of a testing strategy. The core difference between POPART and RT is POPART's ability to sample *without* replacement the testing partition. We connect RT with the coupon collectors problem and use it to prove the point in time at which POPART's efficiency surpasses that of random testing. Finally, we compare POPART with a *uniform systematic testing* (UST) technique.

3.1 Creating a Homogeneous Testing Partition

A *testing partition* is an arbitrary partition over a program's inputs used to select tests. A useful testing partition is one where the result of testing one element of a part generalises

1. Normalised entropy captures how far from uniform a probability distribution is and allows us to compare probability distributions with different event space sizes.

2. The relative cost of UST follows the related work of Bohme and Paul [15]. We doubled POPART's cost over UST to account for its probability computations, Section 4 explores the effect of changing this value.

to all the elements of that part. A testing partition that has this property is *homogeneous*. Given an oracle $\mathcal{O}(F, i)$ that tests a program F on input i and returns *true* (**tt**) or *false* (**ff**), a testing partition $\mathcal{P}$ is called *homogeneous* with respect to $\mathcal{O}$ if there is no part $p \in \mathcal{P}$ such that $i, j \in p \wedge \mathcal{O}(F, i) \neq \mathcal{O}(F, j)$. That is, every part may contain only failing or only passing inputs but not both. The testing efficiency literature has studied some testing partitions that are not homogeneous and whose part therefore must be sampled multiple times (Section 6.3).

For efficiency, homogeneity alone is not enough. Indeed, the identity partition is homogeneous but offers no advantage over directly sampling a program's input space. So another important property of testing partitions is their size or cardinality. Systematic testing requires homogeneity.

To make an arbitrary testing partition homogeneous, we can intersect its parts with the oracle $\mathcal{O}$'s partition to form a new partition. Given a testing partition $\mathcal{P}$ and an oracle $\mathcal{O}$ for the program F, let the oracle-infused partition, OiP, split each part $a_j \in \mathcal{P}$ into up to two parts $b_j = \{i \mid i \in a_j \wedge \mathcal{O}(F, i) = \textbf{tt}\}$ and $\bar{b}_j = \{i \mid i \in a_j \wedge \mathcal{O}(F, i) = \textbf{ff}\}$. OiP is both *homogeneous* with respect to $\mathcal{O}$ and all the elements in any part of OiP are equivalent under $\mathcal{P}$; that is, every part of OiP is a subset of a part from $\mathcal{P}$. We capture this relationship in the following partial order: $\mathcal{P} \preceq \mathcal{P}'$ iff $\forall p \in \mathcal{P}, \exists p' \in \mathcal{P}' \cdot p \subseteq p'$.

We now show that OiP is the homogeneous least upper bound of this partial order.

Theorem 3.1 (Minimality of OiP)**.**

$$OiP = lub\,\{\mathcal{P}' \preceq \mathcal{P} \mid \mathcal{P}' \textit{ is homogeneous}\}$$

Proof. $OiP \preceq P$ and is homogeneous by construction. $\forall p \in \mathcal{P}$, OiP partitions p into at most two homogeneous, but different parts, again by construction. Suppose $\exists \mathcal{P}' \cdot OiP \prec \mathcal{P}' \preceq \mathcal{P}$. Then $\exists p \in \mathcal{P}, p' \in \mathcal{P}', p_o \in OiP \cdot p_o \subset p' \subseteq p$. Since OiP homogeneously partitions p into different parts and $p_o \subset p', p'$ cannot be homogeneous. □

The difficulty of constructing OiP lies in the fact that the oracle's bi-partition of the input space is unknown *a priori* so it is, in general, impossible to assess whether an arbitrary testing partition is *homogeneous* under that oracle. Previous research has assumed *homogeneous* testing partitions without proving it (Section 6). Testing itself provides a means for assessing homogeneity. Deterministic programs execute a path to produce their output. They cannot vary their behaviour along a path. Therefore, for deterministic programs, a computable homogeneous testing partition is one whose homogeneity can be incrementally under approximated via execution, in the course of a testing campaign.

We now show how, using an executable oracle, we can construct such partition. Paths, and more specifically, path conditions, which are the conjunction of all conditions encountered along a path, partition a program's input space and form the program's *path partition*. Under this partition, all inputs that exercise a path (*i.e.* satisfy the same path condition) belong to the same part. Of course, the path partition of a program F may not be homogeneous under $\mathcal{O}$. If, however, the testing oracle $\mathcal{O}$ is executable, the program transformation ϕ in Listing 1 builds a program whose path partition *is* homogeneous with respect to $\mathcal{O}$.

Listing 1: Given an executable oracle $\mathcal{O}$, the program transformation ϕ yields a program whose path partition is *homogeneous* with respect to the oracle.

```
1  φ(F, O) =
2  Run o = F(i) for k steps
3  if F(i) has not terminated
4      return unknown
5  else
6      return O(F,i,o)
```

Any path terminating at line 4 in a program transformed by ϕ corresponds to a set of inputs for which the program does not terminate in k steps. If it does terminate, the program F, the input i and the output o are given to the oracle. The oracle returns **tt** on passing inputs, **ff** on failing inputs or `unknown`, when it cannot decide because it is only a partial specification or because it does not terminate. In essence, ϕ applies Theorem 3.1 to the path partition, creating the smallest homogeneous testing partition for a program's path partition. Importantly, it constructs this partition under realistic assumptions: namely, it handles partial oracles and non-terminating programs. In the rest of the paper, we assume such a testing partition: that is, the path partition of an oracle-infused program.

3.2 A Tale of Two Distributions

We rarely have the luxury of testing a program that is "finitely testable", *i.e.* one whose entire input space, I, can be exhaustively tested with a feasible testing budget. Instead, we must sample its behaviour, either over its inputs or the parts of a testing partition[3]. Two probability distributions govern either sampling: the input probability distribution and the part probability distribution. In general, knowing either distribution is a difficult epistemological problem. Domain experts often have an intuition about which program inputs and behaviours are frequent, important, or both; testers can use elicitation techniques [7] to capture these intuitions and use them to prioritise the testing of frequent or critical inputs. When we cannot elicit a useful distribution, we resort to the principle of indifference [16] and resort to a uniform distribution: denoted U^I for the input distribution and $U^{\mathcal{P}}$ for the part distribution.

Let $\mathcal{X}$ be a discrete probability distribution over inputs in I. $\mathcal{X}$ can be inferred from observing the program's usage, also called a usage or an operational profile, or it can be a synthetic probability distribution, created for testing purposes. Previous work formalizing testing has equated input frequency and importance. While testing more likely inputs can increase the tester's confidence about the correctness of the program more quickly [12], this often does not hold. Consider a cyberphysical system with safety-critical values: the value of testing the system with these safety-critical values might greatly outweigh their in-deployment probability. To reflect that, a synthetic probability distribution should be used, one that combines a usage profile with an importance weight.

3. Section 3.1 discusses partition homogeneity and how we can build such a testing partition.

The idea behind systematic testing is to use sampling without replacement over parts to avoid wasting the test budget on redundant tests. As noted above, it too obeys a probability distribution. The input probability distribution is a naturally occurring distribution, the existence of which has been extensively explored in the field of reliability engineering. In contrast, $\mathcal{S}$ ('S' for section or slice), the part probability distribution, has been overlooked. Often, it has been assumed to be uniform, like the input distribution $\mathcal{X}$, and for the same reasons[4]. This assumption does not hold in practice, as Section 2 showed. The reason follows:

Observation 3.1. *The input probability distribution, $\mathcal{X}$, induces a probability distribution over the parts of a testing partition, $\mathcal{S}$ where the probability of each part $p_j \in \mathcal{P}$ is*

$$\mathbb{P}(p_j) = \sum_{x \in p_j} \mathbb{P}(x), x \in I.$$

To see why consider a set A, let $X \sim \mathcal{X}^A$, for some distribution $\mathcal{X}^A$, be a random variable over A and let $S = \{x_1, x_2, \ldots, x_n\}$ be samples drawn from X. For any partition $\mathcal{P}$ over A, S samples the random variable $Y = \{p \mid \exists y \in S \cap p \in \mathcal{P}\}$. Sampling X without replacement only samples Y without replacement when $\mathcal{P}$ is the identity partition.

3.3 Measuring Testing Efficiency

By convention, testing efficiency is the ratio of coverage over resources spent. Usually, coverage is measured via syntactic structures but this has been shown to be suboptimal: Gay *et al.* report

> "test inputs generated specifically to satisfy structural coverage criteria via counterexample-based test generation were typically less effective than randomly generated test inputs" [17], [18], [19].

Kuhn *et al.* [20] argue that testing should supplement structural coverage with some form of input coverage. Our work, like previous studies [21], [22], focuses on probability-weighted input coverage; this allows us to rigorously compare testing methods in terms of the probability of observing tested behaviours.

Let $\mathcal{A}$ be a testing method, given **b**, a temporal testing budget[5]. It selects and tests a program F on a subset of F's inputs. Because it may sample with replacement, $\mathcal{A}(\mathcal{P}, \mathcal{S}, t)$ returns a multiset M whose support set is I. By sampling I, $\mathcal{A}$ necessarily also samples $\mathcal{P}$ under $\mathcal{S}$: implicitly and with an induced $\mathcal{S}$ in the case of RT or explicitly with a chosen $\mathcal{S}$ under ST (Section 3.2): specifically, $Q = \{p_j \mid i \in M \cap p_j \in \mathcal{P}\} \subseteq \mathcal{P}$. We lift $\mathbb{P}$ from inputs to partitions by defining $\mathbb{P}(p_j) = \sum_{i \in p_j} \mathbb{P}(i)$.

We call *probability-weighted input coverage* (*pic*) the cumulative probability weight of each part in Q.

$$pic(Q) = \sum_{p_j \in Q} \mathbb{P}(p_j) \tag{1}$$

4. "In the absence of any relevant evidence, a rational agent will distribute their credence (or 'degrees of belief') equally amongst all the possible outcomes under consideration"–Benjamin [16].

5. We focus on time as the resource for testing. To account for space, one could construct a wrapper around the testing method which, upon reaching a memory limit, returns the set of explored testing parts.

$|M| - |Q|$ counts the redundant tests $\mathcal{A}$ executes. The higher the *pic* a testing method can achieve for a given testing budget the more *efficiently* covers the testing partition $\mathcal{P}$ and, by extension, the probability-weighted input space of the program F.

Definition 3.1 (Testing Efficiency)**.** *Given a testing strategy $\mathcal{A}$, a testing partition $\mathcal{P}$ over the inputs of the subject under test, and a probability distribution $\mathcal{S}$ over over $\mathcal{P}$, the expected testing efficiency of $\mathcal{A}$ on $\mathcal{P}$ under $\mathcal{S}$ in time t is*

$$e(\mathcal{A}, \mathcal{P}, \mathcal{S}, t) = \frac{1}{t}\text{pic}(\mathbb{E}(\mathcal{A}(\mathcal{P}, \mathcal{S}, t))). \tag{2}$$

It is possible to compute the testing efficiency of a particular test suite on a specific computer, *i.e.* dropping expectation on the right-hand side of Equation (2). We are, however, in keeping with most research in testing efficiency, more interested in the general testing efficiency of a testing method $\mathcal{A}$. This makes the testing efficiency an expected value of applying $\mathcal{A}$ to a program.

The goal of testing is to increase the testers' confidence that a program is correct. In measuring probability-weighted input coverage, we quantify the probability of observing one of the tested behaviours under the assumed input probability distribution $\mathcal{S}$. If then $\mathcal{S}$ is an approximation of the in-deployment usage profile, maximizing the probability-weighted input coverage corresponds to minimizing the probability of a user encountering an untested behaviour. Alternatively, a test engineer might have a different strategy in determining his confidence in a programs' readiness for deployment. She can, for example, deem some particular input values (*e.g.* safety-critical, limit values) more important based on experience. Re-weighting the usage profile or constructing a new, synthetic probability distribution, allows focusing a testing campaign on values of interest.

3.4 Sampling With and Without Replacement

The core difference between RT and ST is that RT samples with replacement and ST without replacement. We define RT and connect it to the *Coupon Collectors* problem to derive the expected number of input samples (tests) it needs to take to discover a part of $\mathcal{P}$.

Definition 3.2 (Random Testing)**.** *Given a program F, random testing (*RT*) tests F by sampling F's input I under the probability distribution $\mathcal{X}$ with replacement.*

By convention, we assume the expected cost of testing, *i.e.* executing, each sample is one unit of time. As is traditional, we have defined RT to sample the tested program's input domain. Nonetheless, it simultaneously also samples the testing partition $\mathcal{P}$, using a probability distribution over $\mathcal{P}$'s parts induced by $\mathcal{X}$, as Observation 3.1 explains.

Böhme and Paul derived the following expression for RT's *pic* as a function of time, under the, just stated, standard assumption that each test takes unit time [22, Lemma 2].

$$pic(\text{RT}(\mathcal{P}, \mathcal{S}, t)) = \left[1 - \sum_{i=1}^{|\mathcal{P}|} \mathbb{P}(p_j)\left(1 - \mathbb{P}(p_j)\right)^t\right] \tag{3}$$

Intuitively, by sampling the input space, RT gradually discovers parts of the testing partition $\mathcal{P}$, initially covering

many new parts and gradually slowing down as repeated samples fall in the already explored parts. This process is an instance of the classical coupon collectors problem which seeks to answer the following question: "Given N coupons, how many coupons do you expect you need to draw with replacement before having drawn at least k different coupons?" [23]. For random testing, coupons are the parts of $\mathcal{P}$ which are drawn by sampling $\mathcal{S}$.

Ferrante and Saltalamacchia derived an analytical expression of the expectation of collecting k coupons by sampling with replacement from any given probability distribution [24]. We adapt their proposition 1 [24, §3.2] to our setting.

Let C_k be the random number of executions to discover the k^{th} part. NB: C_k is only the additional executions to discover the k^{th} part; it does not include the executions needed to find the previously discovered parts. Let $\overline{\mathbb{P}}(p_1, \ldots, p_k) = 1 - \mathbb{P}(p_1) - \ldots - \mathbb{P}(p_k)$, for $k \leq N$. For any $k \in \{2, \ldots, N\}$, the expected value of C_k is

$$\mathbb{E}[C_k] = \sum_{p_1 \neq p_2 \neq \cdots \neq p_{k-1} \in \mathcal{P}} \frac{\mathbb{P}(p_1) \cdots \mathbb{P}(p_{k-1})}{\overline{\mathbb{P}}(p_1)\overline{\mathbb{P}}(p_1, p_2) \cdots \overline{\mathbb{P}}(p_1, p_2, \ldots, p_{k-1})} \quad (4)$$

and the expected executions to discover k different parts is

$$\mathbb{E}[C_N(k)] = \sum_{s=1}^{k} \mathbb{E}[C_s]. \quad (5)$$

Equation (5) is monotonic since $\mathbb{E}[C_N(k+1)] = \mathbb{E}[C_N(k)] + \mathbb{E}[C_{k+1}]$ and $\mathbb{E}[C_{k+1}] > 0$; we now show that Equation (4) is also. This is necessary in order to later on, in theorem 3.4, prove that POPART will be more efficient than RT after some testing budget **b**.

Lemma 3.2. $\mathbb{E}[C_{k-1}] < \mathbb{E}[C_k]$

Proof idea. Any point in the coupon collector's problem can be modelled as a single step in geometric distribution. Accumulate the probability of the coupons collected so far: the complement is the chance of a new coupon. The coupon collector's problem is dynamic in the sense that the probability of a concrete subcollection k is not fixed, so neither is the cost of one of size $k+1$. We handle this by modelling the probability of a subcollection as a random variable and taking its expectation. This contrasts with Equation (4) which directly defines this expectation via exhaustive enumeration. Below, we show that the expected cost to reach a subcollection of size k is necessarily less than that to reach a subcollection of size $k+1$ by modelling each as a separate instance of the geometric distribution, albeit this pair of geometric distributions has the property that the success probability of the smaller subcollection is, by construction, greater than that of the larger subcollection.

Proof. Let D be the set of coupons.

Let $D_k \subseteq D$ with cardinality k be the subset of D discovered after $\mathbb{E}[C_N(k)]$ expected coupon purchases.

Let $B_k = \sum_{b \in D_k} \mathbb{P}(b)$ be the probability of D_k.

To move from D_{k-1} to D_k, one must pick a new coupon, ignoring any previously picked coupons. This is the geometric distribution. The expected probability of picking an old coupon at the arbitrary subcollection D_{k-1} is $\mathbb{E}[B_{k-1}]$, so the expected probability of a new coupon is $1 - \mathbb{E}[B_{k-1}]$. Under the geometric distribution, the expected trials to a success is the inverse of the probability of a success, so the expected trials to move from D_{k-1} to D_k is $\frac{1}{1-\mathbb{E}[B_{k-1}]}$. The set D_k, contains all elements of D_{k-1} plus the k-th coupon c_k, *i.e.* $D_k = D_{k-1} \cup \{c_k\} \Rightarrow B_k = B_{k-1} + \mathbb{P}(c_k)$ and, since probabilities are positive, $\mathbb{E}[B_{k-1}] < \mathbb{E}[B_k]$.

$$\begin{aligned} \mathbb{E}[B_{k-1}] < \mathbb{E}[B_k] &\implies 1 - \mathbb{E}[B_{k-1}] > 1 - \mathbb{E}[B_k] \\ &\implies \frac{1}{1 - \mathbb{E}[B_{k-1}]} < \frac{1}{1 - \mathbb{E}[B_k]} \\ &\implies \mathbb{E}[C_{k-1}] < \mathbb{E}[C_k] \end{aligned}$$

□

Definition 3.3 (Systematic Partition Testing)**.** *Given a program F and a testing partition $\mathcal{P}$ equipped with a distribution over its parts $\mathcal{S}$, systematic testing (*ST*) tests F by sampling $\mathcal{P}$'s parts under $\mathcal{S}$ without replacement.*

Unlike RT, a ST technique $\mathcal{A}_S$ *directly* samples the testing partition $\mathcal{P}$ *without* replacement. This allows it to discover exactly one new part with each test. Let $Q \subseteq \mathcal{P}$ be the parts tests; $\mathcal{A}_S$'s *pic* is simply $pic(Q)$. However, $\mathcal{A}_S$'s sampling has a cost. Let the expected cost for sampling be Δ units of time. As such the expected number of parts discovered by $\mathcal{A}_S$ in time t is:

$$\mathbb{E}\Big(\big|\mathcal{A}_S(\mathcal{P}, \mathcal{S}, t)\big|\Big) = \frac{t}{\Delta}. \quad (6)$$

3.5 POPART *vs.* RT

We define POPART and then compare it against random testing to show that it is more efficient on skewed testing partitions. Intuitively, POPART maximises the probability-weighted input cover of a subject program's inputs.

Definition 3.4 (Probability Ordered Partition Testing)**.** *Given a program F and a testing partition $\mathcal{P}$ equipped with a distribution over its parts $\mathcal{S}$,* POPART *tests F by visiting $\mathcal{P}$'s parts in decreasing* pic *under $\mathcal{S}$ without replacement.*

POPART is an instance of $\mathcal{A}_S$, so its expected cost to discover a part is Δ. Let $O = \langle p_1, p_2, \cdots \rangle$ be POPART's greedy ordering of $\mathcal{P}$'s parts by representative probability weight and let O_k be O's prefix of length k. Given time t, POPART visits the first $d = \lfloor \frac{t}{\Delta} \rfloor$ parts in O (Equation (6)), in expectation (since execution times are expected values), so its testing efficiency is:

$$e(\text{PA}, \mathcal{P}, \mathcal{S}, t) = \frac{1}{t} pic(\mathbb{E}(\text{PA}(\mathcal{P}, \mathcal{S}, t))) = \frac{1}{t} pic(O_d) \quad (7)$$

POPART's sensitivity to the probability weight of a partition's parts allows it to possibly achieve higher probability-weighted input coverage than random testing. One immediate, if contrived, example is the partition $\mathcal{P}$ equipped with : $\mathcal{S} = \langle 0.4, 0.3, 0.1, 0.1, 0.1 \rangle$. Given time $t = 6$ and relative cost $\Delta = 2$, the probability-weighted coverage of random testing (Equation (3)) is $pic(RT(\mathcal{P}, \mathcal{S}, 6)) \approxeq 0.78$, while that of POPART is $pic(\text{POPART}(\mathcal{P}, \mathcal{S}, 6)) = 0.8$.

More importantly, Section 4 shows that it is easy to construct *plausible* counterexamples in which POPART outperforms RT, despite its higher execution cost per test. *Sampling without replacement* is the reason. To show this, we use the

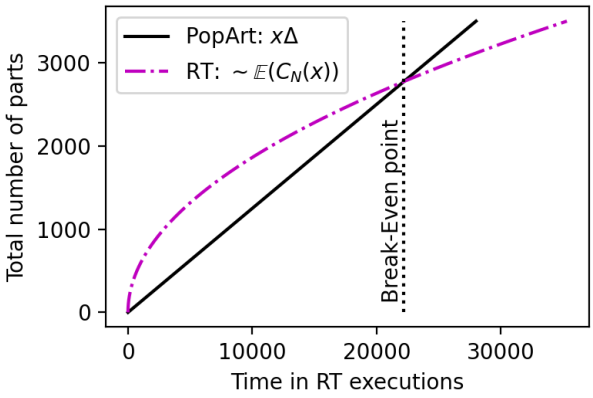


Fig. 3: Assuming the testing partition obeys Zipf-Mandelbrot and POPART execution cost is 8 times more expensive than RT's, POPART and RT *break-even* after $\approx 22,000$ concrete executions. From that point onward, POPART is more efficient due to RT's repeated sampling.

expected number of tests needed to discover a fixed number of parts; but first, consider the following lemma.

Lemma 3.3.

$$|\text{PA}(\mathcal{P},\mathcal{S},t)| = |RT(\mathcal{P},\mathcal{S},t)| \implies \mathbb{E}[\text{pic}(\text{PA}(\mathcal{P},\mathcal{S},t))] \geq \mathbb{E}[\text{pic}(RT(\mathcal{P},\mathcal{S},t))]$$

Proof. POPART is an ideal systematic partition testing strategy that samples $\mathcal{S}$ in order of probability weight; each subset of parts that POPART returns has maximal probability weight for its cardinality, which RT can only, at best, match. □

The question of which method does better then turns into the problem of computing the number of parts visited by each for in a given amount of time. For POPART, this is easy: as a systematic method, it discovers one part in Δ units of time. To do the same for RT, we need to solve Equation (5), which as Ferrante and Saltalamacchia point out, is computationally intractable for domains larger than 10. Luckily, they also show how to approximate $\mathbb{E}[C_N(k)]$ with the following function [24, §3.3], when the underlying distribution follows the Zipf-Mandelbrot law [25], a generalisation of Zipf's law:

$$\mathbb{E}[C_N(k)]_{\text{ZM}} \sim \left(\frac{k}{\left(\sum_{i=1}^{N}(c+i)^{-\theta}\right)^{-1}\Gamma(1-\theta^{-1})} \right)^{\theta} \quad (8)$$

Figure 3 plots the number of distinct parts RT testing and POPART discover when the testing partition obeys a Mandelbrot distribution with parameters $\theta = 2, c = 10$ over a domain of size $N = 100{,}000$. We use Equation (8) to approximate $\mathbb{E}[C_N(k)]$ and fix $\Delta = 8$ *i.e.* each of POPART's tests takes 8 times more time than one of RT's. In Section 4, we use simulation to approximate $\mathbb{E}[C_N(k)]$ on a wider set of probability distributions. Initially, RT's cheaper cost allows it to discover more parts, but, as it starts to repeatedly sample inputs that belong to already visited parts, POPART catches up. After time $x = 22.196$, both methods have discovered the same number of parts, 2775. We name this the *break-even* point: before it, RT has discovered more parts; after it, POPART has the upper hand.

Definition 3.5 (Break-even point)**.** *Given a testing partition $\mathcal{P}$ and a probability distribution over it $\mathcal{S}$, the break-even point of* POPART *and* RT *is the smallest number of parts $k > 1$ such that* POPART *covers them with fewer resources than* RT.

This *break-even* point occurs because POPART incurs Δ, a constant expected cost to discover a new part, whereas RT's cost of discovering new parts is ever increasing, as shown in Lemma 3.2. When this *break-even* point occurs depends on both $\mathcal{S}$, the probability distribution over parts and Δ, POPART's slowdown. Let $\mathbb{P}(p_1), \mathbb{P}(p_2), \ldots, \mathbb{P}(p_n)$ be probability weights of $\mathcal{P}$'s parts under $\mathcal{S}$ indexed in decreasing probability order *i.e.* $\mathbb{P}(p_j) \geq \mathbb{P}(p_{j+1})$. Then, the *break-even* point is the smallest positive value k such that:

$$k\Delta \leq \mathbb{E}[C_N(k)] \quad (9)$$

Evaluating the right-hand side of Equation (9) using Equation (4) and Equation (5) is, as previously mentioned, computationally intractable for larger domains. If the parts of $\mathcal{P}$ follows a Zipf-Mandelbrot probability distribution with known θ and c parameters, we can use Equation (8). Alternatively, we can use Equation (10) as an upper bound on $\mathbb{E}[C_N(k)]$.

$$\mathbb{E}[C_N(k)] \leq 1 + \frac{1}{1-\mathbb{P}(p_1)} + \cdots + \frac{1}{1-\left(\sum_1^k \mathbb{P}(p_k)\right)} \quad (10)$$

Equation (10) replaces the number of expected samples to get new parts, C_i, with their value when using RT under the assumption that random testing samples parts in decreasing probability order. This assumption is not, in fact, strong, since, *in expectation*, RT will sample parts in that order since the probability of RT sampling each part is equal to its probability under $\mathcal{S}$. It takes just a single sample to discover the first part since no part has already been seen. Once RT has already discovered $i-1$ parts, the cost (*i.e.* number of samples) of finding a new one follows, as shown in the proof of Lemma 3.2, a geometric distribution and is equal to $\mathbb{E}[C_i] = \frac{1}{1-\sum_1^i \mathbb{P}(p_i)}$.

When the *break-even* point has been reached, POPART will have a higher *pic* than RT and will continue to do so for any subsequent point in time. We show this in the following theorem.

Theorem 3.4 (When Without Replacement Pays for Itself)**.** *Given resources $\boldsymbol{b}$, a testing partition $\mathcal{P}$ and a probability distribution over the parts, $\mathcal{S}$,* POPART *outperforms* (i.e. *makes better use of $\boldsymbol{b}$)* RT*, in expectation, whenever $\boldsymbol{b}$ exceeds the point where* POPART*'s and* RT*'s path discovery cost breaks even:*

$$\exists k \in \mathbb{N}\ s.t.\ k\Delta \leq \mathbb{E}[C_N(k)] \wedge k\Delta \leq \boldsymbol{b} \implies \forall t \geq k\Delta, \text{pic}(\mathbb{E}[\text{POPART}(\mathcal{P},\mathcal{S},t)]) \geq \text{pic}(\mathbb{E}[RT(\mathcal{P},\mathcal{S},t)]). \quad (11)$$

Proof idea, the antecedent of the implication of the theorem statement assumes that POPART and RT reach a *break-even* point before exhausting **b**. First, we use Lemma 3.3 to show that at the break-even point, POPART has a higher *pic*

than RT. Then, we show, by induction, that, if the antecedent is true for k parts, it must be true for all values $\geq k$.

Proof. When $k\Delta \leq \mathbb{E}[C_N(k)] \wedge k\Delta \leq \mathbf{b}$ holds, k is the number of parts such that $k\Delta$, the time POPART takes to cover those parts, is less than $\mathbb{E}[C_N(k)]$, the time RT needs, and that $k\Delta$ occurs before the time budget $\mathbf{b}$. Since both methods have discovered the same number of parts, it follows, from Lemma 3.3, that

$$|\text{RT}(\mathcal{P}, \mathcal{S}, k\Delta)| \leq |\text{PA}(\mathcal{P}, \mathcal{S}, k\Delta)| = k \implies$$
$$pic(\mathbb{E}[\text{PA}(\mathcal{P}, \mathcal{S}, k\Delta)]) \geq pic(\mathbb{E}[\text{RT}(\mathcal{P}, \mathcal{S}, k\Delta)])$$

Now, we show that the theorem holds for any value $t \geq k\Delta \iff \forall t\colon |\text{PA}(\mathcal{P}, \mathcal{S}, t)| \geq k$. We do so by induction on the number of parts *after* the first k parts: that is, we prove that $\forall i > k, k\Delta \leq \mathbb{E}[C_N(k)] \implies i\Delta \leq \mathbb{E}[C_N(i)]$.

Base case: The existence of a break-even point is *true* under the assumption of the implication of the theorem statement when $i = k$.

Induction hypothesis: Assume $\exists i > k\colon i\Delta \leq \mathbb{E}[C_N(i)]$.

$$i\Delta \leq \mathbb{E}[C_N(i)] \implies^{Equation\ (5)} \tag{12}$$
$$i\Delta \leq \sum_{s=1}^{i} \mathbb{E}[C_s] \implies \tag{13}$$
$$i\Delta \leq \mathbb{E}[C_1] + \cdots + \mathbb{E}[C_i] \implies^{Lemma\ 3.2} \tag{14}$$
$$i\Delta \leq i\mathbb{E}[C_i] \implies \tag{15}$$
$$\Delta \leq \mathbb{E}[C_i] \tag{16}$$

Induction step:

$$(i+1)\Delta \leq \mathbb{E}[C_N(i+1)] \iff \tag{17}$$
$$i\Delta + \Delta \leq \mathbb{E}[C_N(i)] + \mathbb{E}[C_{i+1}] \tag{18}$$

$$Eq.\ (16) \implies^{Lem.\ 3.2} \Delta \leq \mathbb{E}[C_{i+1}] \tag{19}$$
$$Eq.\ (19), Eq.\ (12) \implies Eq.\ (18) \tag{20}$$

□

The practical relevance of Theorem 3.4 turns on the magnitude of Δ and the feasibility of test budgets $\mathbf{b}$ that allow POPART to catch up to RT. As Figure 5 shows, the time it takes to reach the break-even point decreases exponentially with decreasing entropy of the testing partition. Both our experiment in Section 2 and related work [12] support the idea that in practice, program paths induce a highly skewed probability distribution. This leads us to the following conjecture.

Conjecture 1. *Given a skewed path probability distribution, the* break-even *point is reachable within a realistic testing budget.*

We empirically argue that this conjecture holds in Section 4.1 where we look at both the *pic* and the time at which the *break-even* point occurs.

3.6 Uniform Systematic Testing

Böhme and Paul analysed uniform systematic testing (UST), a form of ST restricted to uniformly sampling undiscovered parts [22]. Their choice of the uniform distribution rests on the principle of indifference [16] as the only reasonable solution to the hard epistemic problem of inferring the true distribution over a program's testing partition. They characterise this method as *ideal* ST, show that, despite wasting resources on redundant test executions, RT outperforms it over realistic test budgets, then conclude that, since UST is idealised ST, that RT outperforms ST. We summarise their finding and compare it to POPART, a ST strategy for which their finding does not hold.

Definition 3.6 (Uniform Systematic Testing[6])**.** *Given a program F and a testing partition $\mathcal{P}$, uniform systematic testing (*UST*) tests F by sampling $\mathcal{P}$'s parts uniformly without replacement.*

Uniform sampling without replacement renormalises to uniform over the undiscovered parts at each step. UST is an instance of $\mathcal{A}_S$, so its cost to take each sample is Δ units of time. Because it specifies a uniform probability distribution over undiscovered parts, UST's expected $pic(\text{UST})$ increases linearly [22, Lemma 1]:

$$pic(\mathbb{E}[\text{UST}(\mathcal{P}, U, t)]) = \frac{t}{\Delta|\mathcal{P}|} \quad , \quad t \in [0, \Delta|\mathcal{P}|] \tag{21}$$

Böhme and Paul results imply that RT is more efficient than UST in practice [22, §5]. In our notation, this claim is

$$\exists t < \mathbf{b} \text{ s.t. } e(\text{RT}, \mathcal{P}, \mathcal{S}, t) > e(\text{UST}, \mathcal{P}, U, t) \tag{22}$$

The testing efficiency of RT on the left of Equation (22) takes $\mathcal{S}$, *not* U^I because, execution of uniformly sampled inputs generates a distribution over the parts of the testing partition $\mathcal{P}$ that are unlikely to be uniform, Observation 3.1 in Section 3.2. Indeed, Section 2 presents strong evidence that they are nonuniform in practice. Thus, this result not only compares sampling with and without replacement but across two different probability distributions. This difference in distribution extends to the efficiency computation: that Equation (22) values the same part differently on the two sides of the inequality. Let the most probable part have probability $\frac{1}{2}$ under $\mathcal{S}$. RT earns $\frac{1}{2}$ for it, while all UST ever earns is $\frac{1}{\Delta|\mathcal{P}|}$. This fact also intuitively explains the result. RT does, statistically and in practice, come close to visiting the testing partition's parts in greedy order and maximising its *pic* as Section 4.2 shows, while UST is unlikely to and would not be rewarded for doing so, even if it did.

4 TESTING STRATEGIES UNDER SIMULATION

In this section, we use simulation to compare POPART with Random Testing (RT) and uniform systematic testing (UST). Of particular interest is $\mathcal{S}$, the induced probability distribution over the parts and the size of its support. In Section 4.1, we explore the *break-even* point of POPART and RT, varying the slowdown Δ of POPART and measuring how long it takes to reach the *break-even* point and at what *pic* it occurs. This result gives us our first takeaway: the importance of focusing engineering efforts into lowering Δ for any implementation of POPART which in turn, brings the *break-even* point much earlier.

Throughout our comparisons, we use entropy to measure the flatness of a probability distribution. When all parts of

6. This definition is equivalent to Böhme's and Paul's Systematic Testing Technique.

a testing partition are equiprobable, the probability distribution over parts is horizontal and entropy is maximized[7]. Conversely, when all but one parts have probability zero, the entropy of the testing partition is zero. We use entropy to indicate how far from uniform a probability distribution is. Section 4.2 shows how the entropy of $\mathcal{S}$ affects the performance of POPART, RT and UST. Our results show that each testing method wins depending on the probability distribution over the testing partition, the slowdown of systematic techniques and the target *pic*. However, under realistic assumptions, UST loses to both RT and POPART. Therefore, most of this section focuses on comparing the latter two methods. We find POPART is more efficient when targeting *pic* above ≈ 0.9; otherwise, it loses to RT, confirming RT's superiority over systematic methods for lower coverage targets. The takeaway is to choose POPART only for *pic* targets above ≈ 0.9. Practitioners can use these takeaways to inform their choice of a testing strategy.

Testing methods discover the parts of a testing partition $\mathcal{P}$. This section reports the *pic* that each method achieves in time t on probability distributions $\mathcal{S}$ with different supports. We normalise time such that RT's input sampling and test execution takes expected unit time and different Δ values for POPART and UST.

Our simulations are subject to the usual internal validity threats. Our model, presented in Section 3, may not correctly capture all the parameters of the compared testing strategies. To combat this threat, we adopted models from prior work and augmented them by ranging over probability distributions indexed by their entropy. Second, the scripting that performs and reports the results of the simulations: the interested reader can inspect them at 10.6084/m9.figshare.18544298. Our choice of the probability distributions in Section 4.1 (Normal, Uniform, Zipf-Mandelbrot) represents an external threat to validity. We selected the Normal and Uniform probability distributions because related work has used them; we selected the Zipf-Mandelbrot distribution based on the results of Section 2, which empirically showed path probabilities following a power-law.

4.1 POPART Break Even Analysis

Section 3.5 introduced the concept of a *break-even* point, the point where POPART and RT have discovered the same number of $\mathcal{P}$'s parts and, after which, following from Theorem 3.4, POPART has higher efficiency than RT. Using simulation, we set out to find this break-even points for different distributions. We do so by selecting three distributions, a power-law represented by the Zipf-Mandelbrot distribution, the normal and the uniform distribution, and vary the size of their support (the number of parts in $\mathcal{P}$) from $1,000$ to $100{,}000$ as well as the relative cost of POPART compared to RT, Δ. In practice, we expect useful and interesting distributions to follow a power-law distribution and show the results on the other two distributions as baselines. This is supported by both our case study of codeflaws programs presented in Section 2 and previous work on estimating part execution frequencies [12].

7. The value of the maximum entropy depends on the size of the support of the probability distribution

Figure 4 shows the results of our simulations for all three probability distributions over parts. The first thing to notice is that, the more skewed a probability distribution is, the earlier the break-even point comes. Under a uniform distribution (Figure 4c), POPART cannot leverage path ordering to quickly increase its *pic* while at the same time incurs a slowdown Δ. It has to "wait" until RT starts repeatedly sampling the same parts to catch up (if possible). On the other hand, when $\mathcal{S}$ follows a power-law (Figure 4a), the break-even point is reached earlier.

Beyond the distribution over parts, two more factors affect the probability-weighted input coverage at which break-even points occur. The first one is the relative cost of POPART compared to RT, Δ. As expected, if POPART's executions are only $\Delta = 2$ times slower than RT's, break-even points are reached earlier than when it is $\Delta = 8$ times slower. Finally, our simulations showed that when $\mathcal{S}$ is skewed, increasing the size of $\mathcal{P}$ brings the break-even point to a lower *pic*. This last observation is a promising result for POPART since, in practice, testing partitions, like the oracle-infused path partition, are both skewed and have a very large number of parts.

Figure 4 shows the *pic* at which POPART and RT break even for different path probability distributions $\mathcal{S}$ but how much time is needed to reach that point? To answer this question we generated multiple probability distributions of varying entropy values and simulated both testing methods to compute how much time is needed to reach the *break-even* point. The results in Figure 5 show that the entropy of $\mathcal{S}$ largely determines how quickly the *break-even* point is reached. The time to reach the break-even point decreases exponentially with decreasing entropy of $\mathcal{S}$.

To generate probability distributions with supports of size N with different entropies, we used the Zipf-Mandelbrot distribution with parameters c, θ where the probability of the k-*th* element is given by the following PMF:

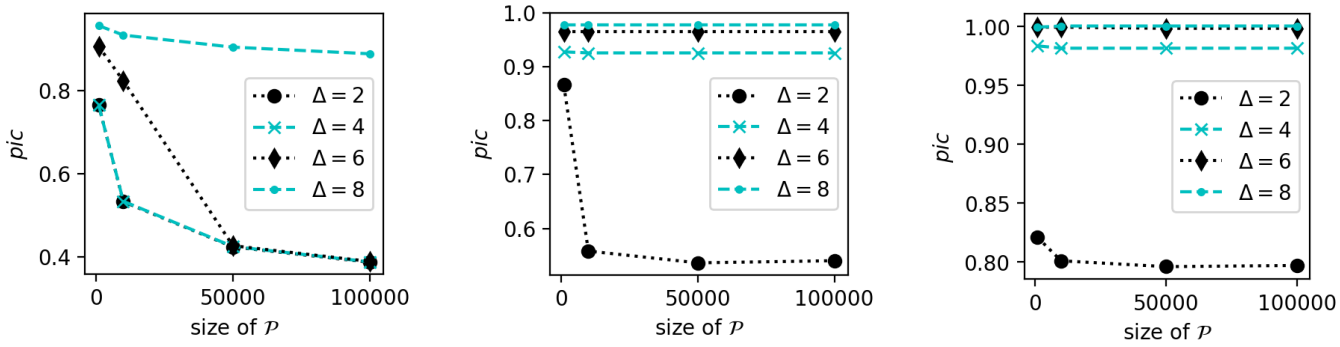
$$\mathbb{P}(k) = \frac{1/(k+c)^{\theta}}{H_{N,c,\theta}}, \text{where } H_{N,c,\theta} = \sum_{i=1}^{N} \frac{1}{(i+c)^{\theta}} \qquad (23)$$

Increasing θ, increasingly skews $\mathcal{S}$ (lower entropy); in contrast, increasing c pushes $\mathcal{S}$ towards the uniform distribution (higher entropy). The maximum possible entropy of a distribution increases as its support size increases. For presentation purposes, we normalise $\mathcal{S}$'s entropy by dividing it by the entropy of the uniform distribution with the same support.

For Figure 5, we fixed the size to $N = 100{,}000$ and varied the parameters c and θ. For each probability distribution, we simulate RT 10 times and compute the mean. Since POPART is deterministic for a given distribution, this experiment requires only a single simulation per slowdown (Δ) value.

4.2 Entropy Determines POPART's Efficiency

The entropy of the path probability distribution, $H(\mathcal{S})$, significantly affects the efficiency of POPART. Here, we explore this observation by generating probability distributions of varying entropy and simulating POPART on them. We similarly simulate RT for the same probability distributions to show its difference from POPART. We generate these probability distributions using the Zipf-Mandelbrot distribution



(a) $\mathcal{S} = \text{Mandelbrot}(\theta = 2, c = 10)$ (b) $\mathcal{S} = \text{Normal}(\sigma^2 = \frac{|\mathcal{P}|}{10})$ (c) $\mathcal{S} = \text{Uniform}$

Fig. 4: For three different probability distributions over the testing partition $\mathcal{P}$, we compute using simulations the *break-even* point for PopArt and RT for different values of Δ (relative cost of PopArt). The x-axis represents the size of $\mathcal{P}$ and the y-axis the probability-weighted input coverage at which PopArt and RT have explored the same number of parts. Skewed probability distributions, such as the *Zipf-Mandelbrot*, bring the *break-even* point earlier, especially for larger testing partitions.

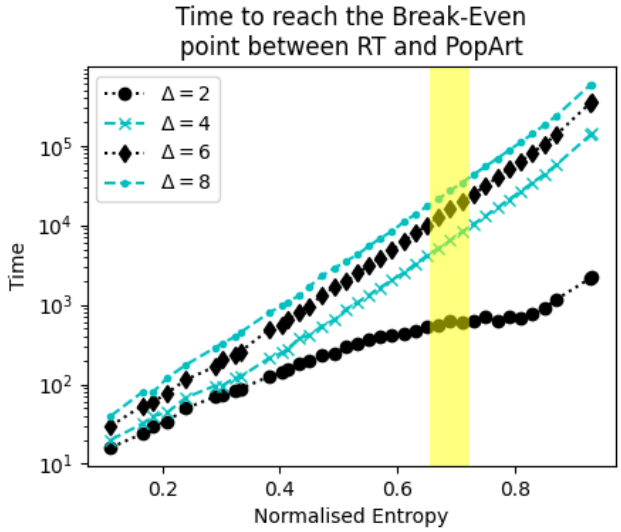
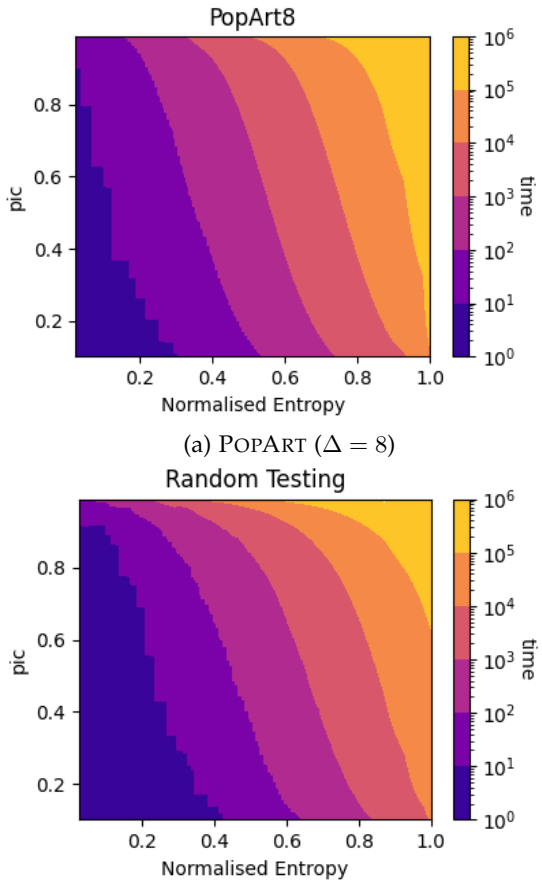


Fig. 5: Slightly lowering the entropy of $\mathcal{S}$ exponentially reduces the time to reach the *break-even* point. The yellow line indicates the normalised entropy of path probability distributions harvested in Section 2. $|\mathcal{P}| = 100{,}000$.



(a) PopArt ($\Delta = 8$)

(b) Random Testing

Fig. 6: For different probability distributions generated using Equation (23) with $N = 100{,}000$, we simulate PopArt and RT to show the time it takes for each method to achieve a certain *pic* for distributions with different entropies.

introduced in Equation (23). Our simulations show that, when $\mathcal{S}$ is non-uniform and when high probability-weighted input coverage is necessary, PopArt outperforms RT.

Figure 6 shows how much time it takes for PopArt and RT to achieve a certain amount of *pic* for probability distributions indexed by normalised entropy. The probability distributions are over a support of size $N = 100{,}000$. To account for the randomness of RT, we repeat each RT simulation 10 times and keep their average. As in Section 3, our time unit is the expected cost of a single RT test execution; PopArt has an $\times 8$ slowdown, *i.e.* $\Delta = 8$. Figure 6a shows that PopArt's ability to leverage skew in $\mathcal{S}$ allows it to quickly reach high *pic* when $\mathcal{S}$ is far from uniform.

Figure 6b shows how random testing performs for the same part probability distributions. RT has similar behaviour to PopArt; under a non-uniform distribution $\mathcal{S}$, it is very likely to sample the high-probability parts first and does so with a much lower cost. However, upon reaching a probability-weighted input coverage higher than 0.9, it starts to significantly slowdown, due to repeated samplings of already tested parts. Going towards more uniform distribu-

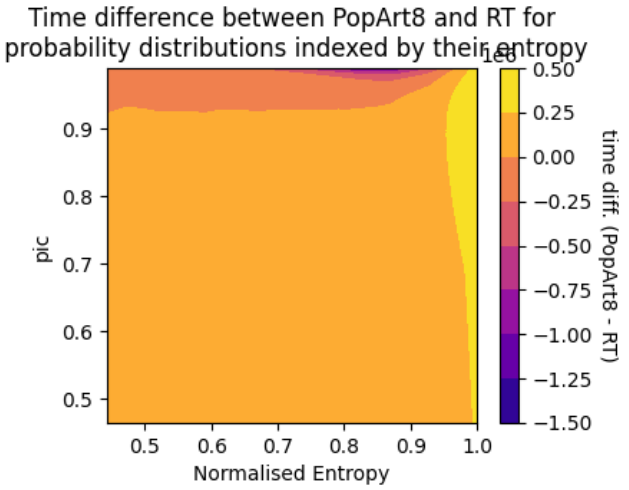


Fig. 7: For non-uniform probability distributions over the testing partition, POPART ($\Delta = 8$) is faster than RT when the target probability-weighted input coverage is above ≈ 0.92.

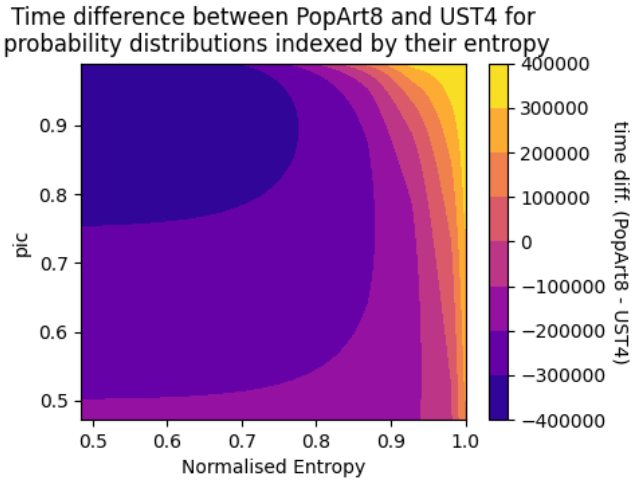


Fig. 8: POPART ($\Delta = 8$) is faster than the uniform systematic testing strategy, UST ($\Delta = 4$), on all but the most uniform part probability distributions despite being 100% slower.

tions, RT does become slower to gain *pic* but at the same time is unlikely to draw new samples from the already visited parts making it faster than POPART.

Figure 7 directly compares POPART with a slowdown $\Delta = 8$ and RT, on probability distributions with normalised entropy higher than 0.5 and *pic* > 0.5. We focus on this region since it is where the two methods most differ and because we believe it to be the most practically relevant, as testers who care about coverage aim to maximise coverage. Probability distributions with lower normalised entropy tend to have a few high probability parts and a very long tail of near-zero probabilities. For example, the Zipf-Mandelbrot distribution used in Figure 4a has a normalized entropy of 0.69.

Figure 7 shows that, for non uniform $\mathcal{S}$, when the target *pic* is above ≈ 0.92, POPART ($\Delta = 8$) is faster than RT. On the other hand, lower *pic* targets and uniform part probability distributions favor RT. Interestingly, the boundary between the two methods appears almost constant for distributions with normalised entropy between 0.5 and 0.85. A graph, like this one, for their software, would help practitioners, who care about probability-weighted input coverage, make an informed selection of a testing method. Given an estimation of the entropy of the distribution $\mathcal{S}$ if a *pic* higher than ≈ 0.92 is desired, then POPART ($\Delta = 8$) should be preferred; otherwise, we should use RT.

While the boundary between which testing method is more efficient between RT and POPART ($\Delta = 8$) remains around a fixed *pic* value (≈ 0.92) when $H(\mathcal{S}) \leq 0.85$, the time it takes to reach 0.92 probability weighted coverage changes significantly depending on $\mathcal{S}$'s entropy. The time to reach the aforementioned boundary is in fact the time it takes to reach the *break-even* point as shown in Figure 5 for $\Delta = 8$.

For comparison with the UST testing strategy, Figure 8 shows the time difference to achieve different *pic* targets across different part distributions. Here UST has a slowdown $\Delta = 4$ while POPART keeps the same $\Delta = 8$ slowdown used in figures 6a and 7. The result is that UST ($\Delta = 4$) is faster for part probability distributions very close to uniform since POPART ($\Delta = 8$) gets no leverage from its ordering but is 2 times slower than UST ($\Delta = 4$). Part distribution with lower entropy heavily favours POPART ($\Delta = 8$). It should be noted that the y-axis goes up to 0.999 *pic*. If a tester requires 100% probability-weighted input coverage, she should use UST over both POPART and RT.

Based on the results shown in Figure 7 and Figure 8, it is clear that each testing technique has a part of the space on which it outperforms its competitors. Our contribution to the testing efficiency research is in introducing POPART and highlighting the space in which it wins over previously studied testing strategies. Previous work has already shown that a uniform $\mathcal{S}$ is unlikely, which makes POPART a competitive option when testing needs require a high degree of probability-weighted input coverage such as critical systems.

5 REALISING POPART

Implementing POPART rests on constructing the testing partition $\mathcal{P}$ and using an input probability distribution to induce a part probability distribution $\mathcal{S}$. For POPART to achieve wide-spread use, its overhead (Δ) must be much smaller than the eight-fold slowdown we have assumed, outside of Section 4.1 where we vary Δ, in Section 4. Nonetheless, POPART can be built and used today using the tools of *probabilistic program analysis* which combines symbolic execution with model counting [26], [27]. Symbolic execution uses the path partition, a partition that is naturally aligned with program behaviors. Using it, every path π_i defines a part of the path partition such that $x \in p_i \Leftrightarrow \pi_i \wedge x$. Model counting is used to *count* the number of inputs satisfying a path constraint, allowing for such analysis to compute path probabilities.

Filieri *et al.* developed a path selection strategy [28], *FPS*, for probabilistic symbolic execution which upon reaching a conditional, computes the probabilities of both branches and selects one at random based on these probabilities. An interesting feature of *FPS* is that it effectively samples *without* replacement the path partition and traverses paths with a probability that is proportional to the probability of

sampling an input from the corresponding part [28, thm 1]. While *FPS* does not guarantee that paths will be traversed in exact decreasing order of probability, as Section 5.2 shows, the order in which paths are tested is very close to POPART's.

Our POPART prototype extends Java *Symbolic Pathfinder* (SPF) [29]. It implements Filieri *et al.*'s probabilistic path traversal, using the LattE model counter [30]. Using these tools permits our prototype to compute exact *pic*, unlike RT and UST, which must estimate *pic* using Equation (3) and Equation (21). Running our implementation on a Macbook Pro 2018 (2.2GHz cpu, 16GB ram) inside a Linux Mint virtual machine, we observe an average slowdown of $\Delta \approx 6.5$ in steady state, when subtracting VM and symbolic execution start-up time to mitigate against measurement instability [31]. To be conservative, we used $\Delta = 8$ in Section 4.

Both symbolic execution and exact integer model counting significantly limit the types of programs that this implementation can test. Symbolic execution has significantly improved over the last decade and now handles industrial scale code such as cyberphysical control systems in the aerospace industry [32], [33] and addresses modern software engineering challenges such as testing deep neural networks and smart contracts [34], [35]. However, path explosion remains a problem for any exact path-based method, such as symbolic execution. Using *FPS* to prioritise the most likely paths allows POPART cover most of the probability-weighted input space with a small number of paths, given that the path probability distribution has low entropy.

Propositional model counting [36], [37] is a natural fit with symbolic execution as both techniques work with symbolic representations of the path constraints. It comes however at a significant cost (it belongs to the #P complexity class) as it has to count all possible solutions to a query. Barvinok's algorithm [38], [39] speeds up the solution, making it's complexity linear in the number of variables, but restricts the types of constraints to linear integer constraints. Since LattE, uses Barvinok's algorithm, our implementation inherits these limitations. To relax these restrictions, approximate model counting based on sampling could be used instead [37] which does not limit the type of constraints it can be applied to but is only effective for small input domains.

Below, we conduct two case studies to showcase the different aspects of *FPS* and its relationship to POPART. Each case study symbolically executes a harness that calls a subject program. Our implementation of *FPS* takes as input the number of symbolic variables and a minimum and maximum value for those symbolic integer variables; those are then passed to the LattE model counter that computes the probabilities of branches and paths. The first case study is a sanity check: it shows that our implementation of *FPS* does indeed prioritise high probability paths. Because *FPS* is probabilistic, its *pic* may differ from POPART's for a given testing budget. Our second case study investigates how important this difference is.

5.1 *FPS*'s *pic* Over Time

To experimentally observe whether using symbolic execution with Filieri's *et al.* probabilistic path selection strategy *FPS*, does indeed prioritise high probability paths, we compare pairs of sorting functions such as *merge sort* and *quick sort*.

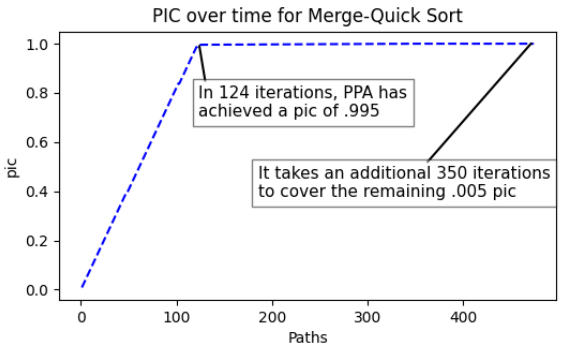


Fig. 9: Probability-weighted input coverage as a function of paths traversed by *FPS* when comparing Merge and Quick sort.

A testing harness is creating an array, passing it to both functions, and finally comparing their output. At the end of each path traversal, the implementation computes the path probability and prints it. We are interested at the rate with which *FPS* gains probability-weighted input coverage. For this experiment, we need to concretize the size of the arrays that are being sorted since Symbolic PathFinder does not handle arrays with symbolic length. The results shown in Figure 9 are for arrays of 5 elements with values in the range $[-999, 1000]$ and a uniform probability over them.

Figure 9 shows that *FPS* does, indeed, traverse the high probability paths first, quickly covering most of the probability-weighted input space. We observe that the paths form two sets with regard to their probability weight. The first 124 paths explored have a significantly higher probability, accounting for 99.5% of the total probability weight. Interestingly, this is also the case for other pairs of sorting functions, such as $\langle$merge sort, selection sort$\rangle$, for which *FPS* achieved a *pic* of 0.995 in the first 123 iterations (out of 208 paths) and $\langle$bubble sort, selection sort$\rangle$ where *FPS* calculated a *pic* of 0.995 in 122 iterations (out of 523 paths). On average, our implementation took 80 milliseconds to execute a path, reaching the 0.995 *pic* mark in ≈ 9.8 seconds. These examples are good cases for POPART, since their path conditions induce a skewed probability distribution on the paths which POPART leverages to quickly cover most of the probability-weighted input space.

5.2 Comparing *FPS* and POPART

In Section 3 we showed how selecting the paths in decreasing probability order leads to better *pic* over time than random testing and partition testing with uniform path selection probability. Our implementation of *FPS*, is *likely* to exercise the paths in decreasing probability order. Its path traversal order will be, at best, POPART's greedy order and, at worst, the complete opposite (meaning that after i iterations, POPART will have exercised the i *less* likely paths) due to the probabilistic nature of the path selection strategy used.

To measure how far from POPART's order is *FPS*'s implementation *on average*, we generated 10 random programs and used ran *FPS* on each 100 times. It took our implementation 20 milliseconds, on average, to execute a path through both programs. We then compare how the average *pic* achieved over time compares with the greedy order, with the results

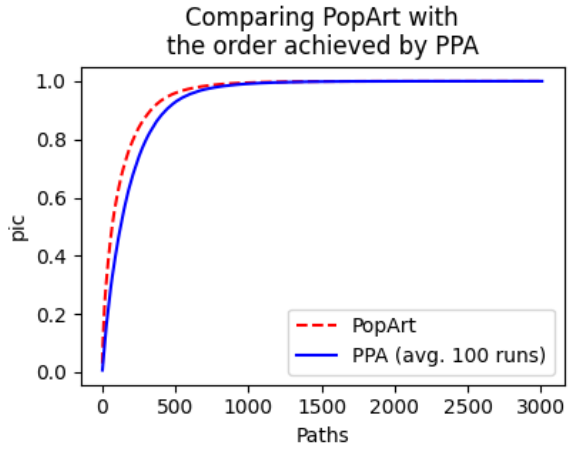


Fig. 10: Comparing the *pic* per number of parts explored achieved by *FPS* vs PopArt. *FPS* manages to stay very close to the theoretical best (PopArt), having a difference of at most 0.15 *pic* accross all 100 runs.

shown in Figure 10. These indicate that using the probabilistic path selection technique achieves near-optimal *pic* over time while only storing the symbolic states of a single path unlike what a greedy approach would do as discussed below.

Two interesting observations for this experiment are that first the gap between the greedy and probabilistic path selection closes rapidly as more paths are explored. Second, the performance of *FPS* is very consistent despite being a randomized process. Intuitively, *at best* it can explore the paths in the exact greedy order therefore having the average so close to the greedy path order suggests that there should be few "bad" outliers. To confirm this we did the Kolmogorov-Smirnov test between the average curve and *all* executions of *FPS* on the generated programs; the result, k-s $= 0.151$, means that the furthest the individual curves have been from the average at any point is 0.151.

It is possible to have a path selection strategy such that paths are explored in decreasing monotonic order with regards to their probability. This can be achieved if symbolic execution always selects the branch with the highest probability out of all the explored edges of the CFG. Doing so will achieve the greedy path order but loses the property of being a DFS-like (entry to exit) traversal of the CFG. This is a significant drawback for practical symbolic execution since in order to find the most likely path it might have to compute and store most program states. Therefore it is likely to run out of resources for any non-trivial program.

6 Related Work

PopArt as presented in this paper is both very different *and* also similar to random testing. Both methods are guided by the input probability distribution and the probability distribution it induces over the testing partition. These distributions allow each to quantify software dependability as Thévenod-Fosse and Waeselynck argue [40]. Rather than rely on deterministic or arbitrary test case generation, using a *usage profile*, a probability distribution over inputs derived from their in-deployment frequencies, allows both quantifying and prioritising likely faults [12], a core concern of reliability engineering. While other types of testing might reveal more errors, they lack the necessary framework to quantify the degree of input-output behavioral coverage they achieve and therefore are not suitable for reliability engineering. Both PopArt and RT can do so because they are guided by the *usage profile* rather than heuristics.

The closest research on quantifying the progress of a testing campaign is Bohme and Falk's [41] and Bohme *et al.*'s [42] work on fuzzing. However, unlike the expected probability input coverage computed for RT and PopArt, their work can only bound the probability of a fuzzer finding a new program behaviour; that is, their framework can only provide statistical guarantees about *the fuzzer's* input probability distribution which can be arbitrary far from the user's usage profile, making the result incompatible with reliability testing.

Our work on PopArt can be seen as a novel instance of the test case prioritisation problem [43]. Test case prioritisation (TCP) is the problem of selecting and ordering a subset of the test cases available in a test suite with the goal of maximizing some measure given that only a limited number of tests can be executed; this is the core concept of PopArt too, i.e. maximise coverage under a given resource bound. The novelty of PopArt comes from the measure it uses to rank and evaluate the parts of its testing partition. Probability-weighted input coverage is unlike traditional TCP approaches that use structural coverage measures such as branch or statement coverage [44]. Another difference of PopArt from TCP methods is that it does not require a test suite, instead, it uses a testing partition which is either provided or can be constructed as shown in Section 3.

In the rest of this section, we first discuss the challenges of software reliability testing, then, we review the attempts of software engineering researchers at characterizing and comparing random testing with partition and systematic testing. Finally, we discuss probabilistic program analysis upon which rests our implementation of PopArt.

6.1 Probabilistic Program Analysis

PopArt is a systematic testing strategy that visits the parts of a homogeneous testing partition in decreasing order of their probability weight, without replacement. Its implementation rests on the advances made in the field of probabilistic program analysis which combines symbolic execution [45] and model counting [36] in order to precisely reason about a program's paths and their probabilities. Symbolic execution is used to traverse the program paths and using its symbolic representation of path constrains allows for computing path probabilities using model counting.

Geldenhuys *et al.* were first to introduce probabilistic symbolic execution (PSE) [27], which used the LattE model counter [30] to count the number of inputs traversing a path and, under the assumption of a uniform input probability distribution, compute its probability. Since then researchers have worked on improving the engineering of PSE in Java-Pathfinder [26], applying the techniques to solve problems, such as software change quantification in change impact analysis [46]. They have adapted model counting to the specific needs of program analysis by decomposing queries, rewriting them to a normal form and caching the results [47],

[48], and modifying them to support non-uniform input distributions [49].

Core to PopArt's implementation is Filieri's *et al.* path selection strategy based on path probabilities [28]. Filieri's *et al.* work, aimed at increasing the efficiency of probabilistic program analysis, proved that using the probability of each branch to choose which branch to explore will visit a program's paths in the same order as if one directly uniformly sampled the program's input domain. This property makes symbolic execution visit paths in an order very close, in expectation, to their probability weight, which PopArt requires, as shown in Section 5.2.

6.2 Reliability Testing

Testing remains the most common approach for validating software. Developers often employ a mix of different testing methods to increase their confidence in the correctness of their product [50], [51]. Broadly, these testing methods either generate their inputs manually or automatically. In *manual testing*, a developer selects a set of inputs. *Automatic testing* relies on a combination of heuristics and randomness to generate test input.

A special category of automatic testing, closely related to PopArt, is *statistical testing* [40], [52]. Any form of testing that selects inputs based on an input probability distribution that captures the in-deployment usage, also called *usage profile*, allows it to establish a statistical measure of reliability of the underlying software under that usage profile. This is in contrast with other stochastic testing methods, such as *fuzzing* [53], [54], which despite their randomness, do not allow for a direct statistical assessment of the reliability of the tested software since, as Musa *et al.* [55, §5] observe "making a good reliability estimate depends on testing the product as if it were in the field", a constraint not met by the arbitrary distributions induced by fuzzing.

Fuzzing has recently made great progress and seen increasing adoption by practitioners thanks to increasingly available computational power and to a series of improvements, in both its theory and implementation [56], [57], [58]. Google's fuzzing platform *OSS-Fuzz*, for example, found $30,000$ bugs in 500 open source projects [59]. Yet, despite being able to find faults, fuzzing does not directly assess the reliability of a program under a usage profile. This is due to its reliance on heuristics that, while maximizing a target measure, *e.g.* line coverage, introduce sample bias with an unknown divergence from a program's behaviour under its expected usage. The closest to reliability testing via fuzzing is Bohme's *et al.* work [41], [42], [60] on modeling fuzzers and estimating the probability of the discovery of a new program behaviour during a fuzzing campaign.

Leveraging usage profiles to guide testing effort has been a longstanding goal of reliability testing in software engineering [61]. Musa [12], [62] was one of the pioneers who investigated usage profiles and argued for their importance, highlighting ways in which practitioners could reduce the very large space of possible input probability distributions. Similarly, Whittaker and Poore showed how to model usage profiles as Markov chains and derive reliability measures of the distribution of failures within the software [63]. How to effectively use a usage profile for reliability in software testing has mostly concerned the aerospace and telecommunication industries [64], which have been historically tightly regulated. As software increasingly pervades the global economy and becomes integral to critical infrastructure, we argue that the scope for usage profiles should be expanded.

There are two core challenges in effectively applying the principles of reliability engineering to software testing. First, we need to capture the usage distribution in some form and then have an *efficient* mechanism to sample it. Neither is easy [65, §13.3]. The first is usually approximated by a combination of domain knowledge of some expert and usage data [66], [67]. We acknowledge the difficulty of defining a good usage profile; PopArt itself does not attempt to solve this problem but instead relies on good input, as usual, specifically a good usage profile, to produce meaningful results. We note, however, that PopArt can tolerate noise in the usage profile and, further, its user can restrict the usage profile to a subset of the program's inputs for which the usage profile is better known, zeroing or falling back on the uniform for the rest. The difficulty of the second challenge, generating input data by sampling some arbitrary distribution, is perhaps best captured in the *structured input generation problem* (SIG) [68]. SIG is the problem of sampling an arbitrary structure under some probability distribution, such as uniformly sampling C programs. It poses two challenges: 1) efficiently building valid structures and 2) ensuring that the sampling procedure over valid structures obeys a given distribution.

To date, research has focused on the first subproblem. Researchers have tried to address SIG by using symbolic methods [68], [69], [70], generative grammars and fuzzing [71], [72], [73]. Visser *et al.* [68] proposed a method for symbolic execution which using *lazy initialisation* allows it to generate complex data structure based on the control flow of a program. Boyapati *et al.* [70] presented Korat, a tool that uses function specifications written in Java to automatically generate possible input data structures up to a predefined bound. To test programs using database systems, Emmi *et al.* [69] implemented a multilingual concolic execution that generates well-formed SQL queries. Compilers and interpreters are another type of program that only accepts highly structured inputs. To better test them, Godefroid *et al.* [71] proposed a whitebox fuzzing technique that builds a symbolic grammar that is then used to generate valid inputs. To test a PDF parser, Godefroid *et al.* [72] showed how a neural network can be used to generate input grammars which are then given to a fuzzer. An interesting feature of their work is that they encode a usage distribution in a recurrent neural network and then, use it to guide the fuzzing process. Finally, Olsthoorn *et al.* [73] combined search-based test case generation with grammar-based fuzzing, using the former to create the general structure of an object and the later to fill the fields such as strings.

To assess reliability, generating valid structured input is not enough. To see why, imagine building an arbitrary number of valid inputs that are arbitrarily unlikely to ever be encountered in the field. The subproblem of generating structured data that follows a usage profile has not been addressed and remains unsolved. Like all work to date, PopArt itself does not address this problem. Instead, PopArt assumes that it is given a usage profile that the user defined to impose a

useful distribution over a subject program's structured inputs. Future work would be combining POPART's implementation with Visser's *et al.* [68] symbolic input generator. Their combination would apply Filieri's *et al.* [28] path selection strategy to guide the construction of structured inputs and could impose a distribution over these structured inputs, induced by a program's paths, that would require only a usage profile over primitive data types.

6.3 Random Testing *vs.* Partition Testing

Testing consumes resources, which are necessarily finite, if not scarce. Testing efficiency studies how best to spend testing resources. Over the last decades, research in test efficiency has compared random testing (RT) to partition testing (PT) and systematic testing (ST) in search of the most efficient method in different settings. The interest in these testing methods stems from their ability to provide quantitative reliability estimations of software [2], [74].

RT does not waste resources on selecting test inputs, but rather conserves all resources for sampling the input domain and testing the target program. The principle idea is that spending resources to select inputs does not pay off in general, because of the cost of selection and the fact that selecting redundant tests is rare. Indeed, Hamlet empirically showed RT's utility due to its speed, implementation simplicity, and absence of bias [75].

In contrast, PT expends resources to divide the input domain and subsequently sample from its subdomains [3], [76]. The goal of partition testing is, is to leverage an equivalence relationship (which defines the partition) over subsets of inputs to generalise the result of a single test to an entire subdomain.

The ideal partition to use for systematic partition testing would be the test oracle partition. A test oracle imposes a bi-partition on a program's input domain, separating the inputs on which the program computes the correct output from those on which it does not. This partition is ideal because it is total and all inputs in a part behave in the same way with respect to the oracle, *i.e.* parts are *homogeneous* [2], [3]. In practice, unless a formal specification is given and covers all inputs, test oracles are partial and impose a tri-partition that divides a program's input domain into correct, incorrect and *unknown* for values on which the oracle is not defined or does not halt. The oracle partition, or more concretely, an input's assignment to one of its parts, is, however, unknowable *a-priori* (otherwise we would not need testing) and therefore a different testing partition is needed.

In practice, we cannot use the test oracle partition, so partition testing resorts to dividing the input domain into test subdomains. Subdomains are sampled for test input values and finally, evaluated against the test oracle. These test subdomains have two problems in practice — overlap and non-homogeneity. In the literature, this division into test subdomains has traditionally been called a partition, despite not necessarily being a mathematical partition, because the test subdomains may neither cover the domain nor be disjoint. The first issue is not a problem in practice, because testing is finite and can rarely exhaustively check all test subdomains. The intersection of test subdomains has, however, proven to be a problem in practice. Researchers call such test subdomains *overlapping*, because they permit an input value to be a member of more than one subdomain [77]. When dividing the input domain using criteria, such as *branches* or *statements*, it is difficult (and unlikely) to be able to guarantee no overlap between subdomains. Test subdomains can be *non-homogeneous*: they can include inputs belonging to both a correct and an incorrect oracle part. This arises when the input domain is divided into test subdomains using criteria other than the test oracle's.

Constructing a set of testing subdomains that are non-overlapping and homogeneous is hard in practice because of the undecidability of the test oracle's equivalence relation. This is perhaps why some of the previous work on partition testing has used the term "partition" aspirationally when in fact they considered non-homogeneous and possibly intersecting subsets of the input domain. Under these assumptions, a single test from each subdomain is not enough to allow for generalisation which lead researchers to define different sampling strategies with the common element that they are sampling *with replacement*. To distinguish the different types of partition testing, in the rest of this section, we will refer to partition testing with replacement as PT_r and to partition testing *without* replacement, also called systematic, as ST. Unlike PT_r, ST assumes a standard, mathematical partition, that is, in terms of PT, its partition is *homogeneous* with non-overlapping parts, by definition. In Section 3.1 we showed how an executable oracle can be used to create a partition that is both homogeneous with respect to the given oracle and by construction has, assuming deterministic executions, non-overlapping subdomains since it is induced by program paths. Whether to spend testing resources to construct a testing partition to sample or simply use RT has been a long-running debate.

Duran and Ntafos [1] were the first to experimentally investigate the difference of testing efficiency between RT and PT_r. They used synthetic examples and small programs to compare PT_r and RT in terms of error finding and coverage. They found that "random testing can be cost-effective for many programs", igniting the search for a better PT method. Hamlet and Taylor [2], followed up on Duran's and Ntafos' work with more comprehensive experiments that varied the division into test subdomains, the failure rates (*i.e.* percentage of failing inputs in each subdomain) and subdomain probabilities. They found PT_r to be slightly more efficient than RT when low probability parts had a high failure rate.

These partially contradicting empirical results were met with scepticism by Hamlet who argued that rather than rely on small scale experiments, researchers should instead analytically compare testing methods, something that allows generalizations and rigorous comparisons [78]. Following that direction, Weyuker and Jeng [3] adopted the assumption of previous empirical results [1], [2] and analytically showed the necessary conditions to minimize and maximize the probability that PT_r triggers at least one fault. Extending this result, Chen and Yu [4] analyzed the worst case for PT_r and showed that, when sampling parts proportionally to their size, PT_r cannot be worse than RT. In 1999, Gutjahr [5] further generalised the work of Weyuker and Jeng. Gutjahr introduced a probabilistic model for failure rate and showed that PT_r is up to k times better at finding errors than RT,

when it divides inputs into k subdomains.

POPART differs from and extends Gutjahr's result in two ways: the measure of testing efficiency and the assumptions it makes. POPART measures *pic*, which it seeks to maximise; Gutjahr's work measures time to error discovery. Additionally, Gutjahr's equates the execution cost of PT_r and RT, in contrast, our work models the difference of RT and POPART with the Δ parameter, allowing for a more flexible and realistic comparison since any non-trivial partition scheme will impose a computational overhead. Gutjahr's theoretical result rests on the strong assumption of uniform input probability. POPART, in contrast makes only the standard assumption of most programs — that users do not give it garbage input. Specifically, POPART requires a good usage profile to produce meaningful results. As Section 6.2 shows, inputs are unlikely to have a uniform distribution when the program is used. Further, even a uniform input distribution induces a highly skewed part probability distribution (Section 2). These facts call for testing efficiency research to better model both the probability distributions over inputs and the probability distribution over parts. Finally, POPART is a systematic testing technique which differs from Gutjahr's work that models partition testing with replacement (PT_r).

Arcuri *et al.* [21] and Chen *et al.* [79] both extensively summarise the evolution of the RT *vs.* PT_r debate. Arcuri *et al.* improved on Ciupa's *et al.* [80] empirical work on the types of faults discovered by random testing, proving a lower bound for the probability of RT covering a set of targets. For PT_r, Chen *et al.* proved theorems on the expected behaviour of PT_r using *proportional sampling* and note that previous work (including theirs) assumes an actual partition, *i.e.* disjoint testing subdomains. Weyuker *et al.* studied the problem of overlapping subdomains and recognized the need "to find systematic or formal methods of constructing the problem partition" [81]. One such method is symbolic execution, which our implementation of POPART uses. Symbolic execution, in theory, guarantees that the generated path conditions partition the input space. This requires symbolic execution to perfectly model all possible explicit and implicit (*e.g.* exceptions, interrupts) state transitions. Further, it must be equipped with an SMT solver capable of solving the resulting queries. While, in general, these queries can be undecidable, in practice, they often are not as witnessed by the success SMT solvers in the last two decades.

Systematic testing, as mentioned earlier, is any testing strategy whose testing partition is an actual partition, so its parts are disjoint, and, further, that testing a single input from each part is sufficient to test it, effectively assuming the *homogeneity* of its parts. Thus, systematic testing can safely sample its parts without replacement. Sharma *et al.* [82] empirically compared RT with ST by testing different Java containers, finding RT to be on-par with ST. Subsequently Böhme and Paul analytically compared the two methods and crowned RT as the most efficient testing method [22]. They show that an ideal systematic technique, UST (Definition 3.6), is less efficient than RT for feasible testing budgets.

Böhme's and Paul's result might look like it contradicts Gutjahr's [5] result on the difference of RT and PT_r; it does not. Two key differences allow them to escape Gutjahr's result. First, UST uniformly samples parts without replacement, while Gutjahr's result used sampling with replacement. Second, their analytical comparison measures the input coverage not the time to the first failure. Given these differences, they show that RT is initially more efficient than UST up to a point in time X. They then argue X is beyond reasonable testing budget limits, making RT more efficient, within the limits of what is feasible. Their model accounts for the slowdown ST incurs for test selection; in this, our work follows their lead using Δ to model the multiplicative slowdown of POPART over RT's execution time. By dropping the uniform sampling assumption over the testing partition and visiting parts in decreasing probability order (Section 3), we defined POPART, a novel, ST technique that does not uniformly sample parts, thereby sidestepping Böhme's and Paul's result, to achieves higher testing efficiency than RT within feasible resource bounds (*i.e.* $\ll$ Böhme's and Paul's X) for realistic path probability distributions when high coverage is needed (Section 4). Cyberphysical control systems must, because of regulation, achieve such high coverage which has pushed NASA towards using symbolic execution [32]. We show that the entropy of distribution over a program's testing partition governs how quickly POPART reaches the break even point, after which it is more efficient than RT; POPART reaches this break even point exponentially faster the lower this entropy is (excepting certainty).

7 CONCLUSION

"Testing is the most commonly used approach for software assurance, yet it remains as much judgment and art as science. We suggest that structural coverage measures must be supplemented with measures of input space coverage, providing a means of verifying that an adequate input model has been defined." —Kuhn *et al.*, Input Space Coverage Matters [20].

We have striven to rise to Kuhn *et al.*'s challenge. Our work views programs as a set of input-output behaviours, weighted by the probability of an input probability distribution. We presented POPART, a systematic probabilistic testing method that visits the parts of a testing partition in decreasing probability order, where part probabilities are induced from a user-provided input distribution. This allows POPART to focus the testing resources on the most likely program behaviours. Comparing it with random testing, we found POPART to be more efficient when coverage demands are above 92%. Finally, our comparison of PopArt with RT takes into account the entropy of the testing partition and shows how it affects the performance of both methods. We showed that both testing methods can leverage nonuniformity to increase their efficiency compared to a uniform systematic testing approach and that, as the entropy of the testing partition decreases, PopArt outperforms RT exponentially earlier.

ACKNOWLEDGEMENTS

This work has been partially supported by the EPSRC funded research project EP/P005888/1 Information Theory and Test Suite Selection.

REFERENCES

[1] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[2] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence." *Transactions on Software Engineering (TSE)*, vol. 16, no. 12, pp. 1402–1411, 1990.

[3] E. J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *Transactions on Software Engineering (TSE)*, vol. 17, no. 7, pp. 703–711, 1991.

[4] T. Y. Chen and Y. T. Yu, "On the Relationship Between Partition and Random Testing," *Transactions on Software Engineering (TSE)*, vol. 20, no. 9144263, pp. 636–638, 1994.

[5] W. J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, 1999.

[6] J. Weisberg, "Formal Epistemology," in *The Stanford Encyclopedia of Philosophy*, winter 2017 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2017.

[7] A. O'Hagan, C. E. Buck, A. Daneshkhah, J. R. Eiser, P. H. Garthwaite, D. J. Jenkinson, J. E. Oakley, and T. Rakow, "Uncertain judgements: eliciting experts' probabilities," 2006.

[8] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: A programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[9] J. R. Larus, "Whole program paths," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 259–269, 1999.

[10] D. Ung and C. Cifuentes, "Optimising hot paths in a dynamic binary translator," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 1, pp. 55–65, 2001.

[11] D. A. Jiménez, "Code placement for improving dynamic branch prediction accuracy," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 107–116, 2005.

[12] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE software*, vol. 10, no. 2, pp. 14–32, 1993.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[14] Wikipedia contributors, "Curve fitting — Wikipedia, the free encyclopedia," 2021, [Online; accessed 21-October-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Curve_fitting

[15] M. Böhme and S. Paul, "On the efficiency of automated testing," *International Symposium on Foundations of Software Engineering (FSE)*, pp. 632–642, 2014. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635923{%}5Cnhttp://dl.acm.org/citation.cfm?doid=2635868.2635923

[16] B. Eva, "Principles of indifference," April 2019. [Online]. Available: http://philsci-archive.pitt.edu/16041/

[17] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 409–424.

[18] G. Gay, M. Staats, M. W. Whalen, and M. P. Heimdahl, "Moving the goalposts: coverage satisfaction is not enough," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, 2014, pp. 19–22.

[19] A. Rajan, M. W. Whalen, and M. P. Heimdahl, "The effect of program and model structure on mc/dc test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 161–170. [Online]. Available: https://doi.org/10.1145/1368088.1368111

[20] R. Kuhn, R. N. Kacker, Y. Lei, and D. Simos, "Input space coverage matters," *Computer*, vol. 53, no. 1, pp. 37–44, 2020.

[21] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2011.

[22] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2015.

[23] Wikipedia contributors, "Coupon collector's problem — Wikipedia, the free encyclopedia," 2021, [Online; accessed 29-August-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Coupon_collector%27s_problem#Extensions_and_generalizations

[24] M. Ferrante and M. Saltalamacchia, "The coupon collector's problem," *Materials matemàtics*, pp. 1–35, 2014, https://ddd.uab.cat/record/132177.

[25] Wikipedia contributors, "Zipf–mandelbrot law — Wikipedia, the free encyclopedia," 2021, [Online; accessed 31-August-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Zipf%E2%80%93Mandelbrot_law

[26] W. Visser and C. S. Păsăreanu, "Probabilistic Programming for Java using Symbolic Execution and Model Counting," *South Africa*, vol. 10, 2017. [Online]. Available: https://doi.org/10.1145/3129416.3129433

[27] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2012, p. 166. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2338965.2336773

[28] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys, "Statistical symbolic execution with informed sampling," in *International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 437–448. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2635868.2635899

[29] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, 2013.

[30] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, "Effective lattice point counting in rational convex polytopes," *Journal of symbolic computation*, vol. 38, no. 4, pp. 1273–1302, 2004.

[31] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 international symposium on memory management*, 2013, pp. 63–74.

[32] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 15–26.

[33] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "Coral: Solving complex constraints for symbolic pathfinder," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 359–374.

[34] D. Gopinath, C. S. Pasareanu, K. Wang, M. Zhang, and S. Khurshid, "Symbolic execution for attribution and attack synthesis in neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 282–283.

[35] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[36] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting," *Frontiers in Artificial Intelligence and Applications*, vol. 185, no. 1, pp. 633–654, 2009.

[37] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman, "From sampling to model counting," *IJCAI International Joint Conference on Artificial Intelligence*, pp. 2293–2299, 2007.

[38] A. I. Barvinok, "A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed," *Mathematics of Operations Research*, vol. 19, no. 4, pp. 769–779, 1994.

[39] A. Barvinok, "Lattice points, polyhedra, and complexity," *Geometric combinatorics*, p. 21, 2007. [Online]. Available: http://books.google.fr/books?hl=en{&}lr={&}id=W{_}SPdwfPTw8C{&}oi=fnd{&}pg=PA21{&}dq={%}22Lattice+Points,+Polyhedra,+and+Complexity{%}22{&}ots=RkcEsr5xvZ{&}sig=beEc4WJSmOozRznQeMvi6iUlhWw

[40] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 5–25, 1991.

[41] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 713–724.

[42] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 678–689.

[43] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[44] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.

[45] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013. [Online]. Available: http://dl.acm.org/ft{_}gateway.cfm?id=2408795{&}type=html

[46] A. Filieri, C. S. PĂsĂreanu, and G. Yang, "Quantification of software changes through probabilistic symbolic execution," *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pp. 703–708, 2016.

[47] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, Reusing and Recycling Constraints in Program Analysis," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 58:1–58:11, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2393596.2393665{%}5Cnhttp://doi.acm.org/10.1145/2393596.2393665

[48] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan, "Constraint Normalization and Parameterized Caching for Quantitative Program Analysis ∗," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pp. 535–546, 2017.

[49] A. Filieri, C. S. Pasareanu, and W. Visser, "Reliability analysis in Symbolic PathFinder," in *International Conference on Software Engineering (ICSE)*, 2013, pp. 622–631.

[50] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 2012, pp. 36–42.

[51] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? a multi-vocal literature review," *Information and Software Technology*, vol. 76, pp. 92–117, 2016.

[52] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Transactions on Software engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[53] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[54] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[55] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996, vol. 222.

[56] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[57] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," 2019.

[58] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[59] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," 2017.

[60] M. Böhme, "Stads: Software testing as species discovery," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 2, pp. 1–52, 2018.

[61] J. Brown and M. Lipow, "Testing for software reliability," in *Proceedings of the international conference on Reliable software*, 1975, pp. 518–527.

[62] J. D. Musa, "Sensitivity of field failure intensity to operational profile errors," in *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. IEEE, 1994, pp. 334–337.

[63] J. A. Whittaker and J. Poore, "Statistical testing for cleanroom software engineering," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, vol. 2. IEEE, 1992, pp. 428–436.

[64] J. A. Whittaker and J. Voas, "Toward a more reliable theory of software reliability," *Computer*, vol. 33, no. 12, pp. 36–42, 2000.

[65] J. Horgan and A. Mathur, "Software testing and reliability," *The Handbook of Software Reliability Engineering*, pp. 531–565, 1996.

[66] H.-G. Gross, *Component-based software testing with UML*. Springer Science & Business Media, 2005.

[67] P. A. Brooks and A. M. Memon, "Automated gui testing guided by usage profiles," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 333–342.

[68] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004, pp. 97–107.

[69] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 151–162.

[70] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, 2002.

[71] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[72] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[73] M. Olsthoorn, A. van Deursen, and A. Panichella, "Generating highly-structured input data by combining search-based testing and grammar-based fuzzing," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1224–1228.

[74] M. Z. Tsoukalas, J. W. Duran, and S. C. Ntafos, "On some reliability estimation problems in random and partition testing," *IEEE Transactions on software Engineering*, vol. 19, no. 7, pp. 687–697, 1993.

[75] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 2002.

[76] P. Ammann and J. Offutt, "Using formal methods to derive test frames in category-partition testing," in *Proceedings of COMPASS'94-1994 IEEE 9th Annual Conference on Computer Assurance*. IEEE, 1994, pp. 69–79.

[77] T. Y. Chen and Y.-T. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, 1996.

[78] R. Hamlet, "Theoretical comparison of testing methods," in *Proceedings of the ACM SIGSOFT'89 third symposium on Software testing, analysis, and verification*, 1989, pp. 28–37.

[79] T. Y. Chen, T. Tse, and Y.-T. Yu, "Proportional sampling strategy: A compendium and some insights," *Journal of Systems and Software*, vol. 58, no. 1, pp. 65–81, 2001.

[80] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the number and nature of faults found by random testing," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 3–28, 2011.

[81] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Transactions on software engineering*, no. 3, pp. 236–246, 1980.

[82] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2011, pp. 262–277.