Complexity Theory of Randomised Testing

PINGSHI YU, Imperial College London, United Kingdom
CHENGSONG TAN, Kaihong, China
NICOLAS WU, Imperial College London, United Kingdom
ALASTAIR DONALDSON, Imperial College London, United Kingdom

Randomised testing is a widely-used approach to software validation, yet despite years of practical development its theoretical foundations remain thin. In particular, the fundamental question of what it means for a set of inputs to be *generable* has gone unanswered in both the literature and folklore. We present, for the first time, complexity-theoretic foundations for random generators in software testing. We model generators as Turing machine transducers that consume random bits and produce string-encoded outputs, and show that the theoretically generable languages coincide exactly with the recursively enumerable languages. This has direct implications for testing at the boundaries of decidability, such as in the field of compiler testing. Turning to *efficient* generation, we show that the polynomial-time generable languages lie within *NP*, that certain important *NP*-complete languages admit efficient generators, and that—under standard cryptographic assumptions—there are languages in *P* for which no efficient generator exists: the complexity of efficienct generation and of efficient decision are not the same. We then show that space-bounded complexity is the natural framework for generators producing *correlated* samples, capturing methodologies such as coverage-guided fuzzing and symbolic execution. Beyond classification, we characterise efficient generability: a language has a polynomial-time generator iff it admits a *certificate scheme* over a verifier—so witness planting, the folklore technique behind generators to test SAT solvers, is in a sense the only route to efficient generation. Our theory also yields design principles for property-based testing libraries: we prove no library can compositionally derive efficient generators from logical predicates involving conjunction or negation, under standard assumptions. However, restricted classes like *NL* (equivalently, linear Datalog predicates) would admit such a compilation.

1 Introduction

At first sight, *software testing*, concerned with the observed behaviour of particular programs on particular inputs, is far removed from *complexity theory*, which distinguishes what can be computed in principle from what can be computed by an efficient algorithm. Yet, in *property-based testing* (PBT) [10, 33], where a *system under test* (SUT) is executed on many automatically generated inputs, a critical component to the success of a testing campaign is the *generator*, a randomised algorithm. Due to the intricacies of realistic generators, much work has gone into their implementation [31, 50], library tooling [10, 20, 33, 52], and on automatic *derivation* of generators from predicates [16, 17, 28]. But despite their importance, little is known about the *theoretical limits* of generator expressivity: what can be generated *at all*, and what can be generated *efficiently*?

We give the first complexity-theoretic formalisation of generators in randomised testing, establishing a number of results about what can and cannot be generated, and why. Our formulation is applicable to the majority of generators found in testing practice: generators are modelled as Turing machine transducers that consume bitstrings and must *eventually* produce string-encoded outputs. Once generators are cast within this framework, each generator associates with a formal language—the set of outputs it can produce. This allows rigorous investigation of the limits of generation in general, as well as under a variety of time and space constraints. In establishing our novel framework, we build upon previous theoretical work on language generation that predates PBT (and which in our view deserves more attention) [44, 45].

Authors' Contact Information: Pingshi Yu, p.yu22@imperial.ac.uk, Imperial College London, London, United Kingdom; Chengsong Tan, tanchengsong@kaihong.com, Kaihong, Shenzhen, China; Nicolas Wu, n.wu@imperial.ac.uk, Imperial College London, London, United Kingdom; Alastair Donaldson, alastair.donaldson@imperial.ac.uk, Imperial College London, London, United Kingdom.

```c
// [a=19, b=4]
void foo(int a, int b) {
  unsigned i = 0;

  while(i != a) {
    i += b;
  }
}
```

(a) Nonterminating C program.

```
(declare-const p Bool)
(declare-const r Bool)
(declare-const q Bool)
; from unsat generator
(assert (=> (and q r)
            (and (not q) r)))
; from sat generator
; by e.g. [p=F, q=T, r=F]
(assert (=> (and p q) r))
```

(b) (Un)satisfiable formulas.

```c
// [a=21219]
// halts in 100 steps
void bar(int a) {
  int i;
  for (i = 0; i < 100; i++) {
     if (i == a)
        break;
  }
}
```

(c) Terminating C program.

Fig. 1. Possible test cases produced by example generators.

To illustrate the kinds of questions our theoretical framework allows us to ask and answer, we give three examples in the context of programming languages research.

Example: testing a termination checker. Suppose we are presented with a termination checking tool for C-programs, which is claimed to be conservative: the tool should emit a "may not terminate" diagnostic for *at least* those (program, input) pairs where the program does not terminate on the input. To apply PBT to this tool we thus need a generator of elements from the set **NonTerm** $= \{(p, i) : p \text{ a C program}, i \text{ an input}, p \text{ does not terminate on } i\}$. If testing finds a (p, i) pair for which the checker does *not* emit a diagnostic, it has found a bug in the checker. Figure 1a illustrates one such pair that should be producible by the generator. While of course we can create a generator that produces many such pairs of examples (possibly including Figure 1a), our theory tells us that it is *impossible* to create such a generator without sacrificing some part of the **NonTerm** space: we show that the recursively enumerable (*RE*) languages are precisely the generable ones, and since **NonTerm** is not in *RE* (assuming an unbounded memory semantics for C), it cannot be generable.

Example: testing a SAT solver. To apply PBT to a SAT solver we would like two generators, generating satisfiable (unsatisfiable) formulas: if the solver claims that any are unsatisfiable (satisfiable) this is a bug. Figure 1b uses SMT-LIB syntax to sketch simple formulas that such generators might be expected to produce. Our theory confirms that generators covering the full space of satisfiable or unsatisfiable problems can *exist*, and that a generator of satisfiable problems that runs in polynomial time can be constructed (even though deciding satisfiability is NP-complete). However, we show that (unless $P = NP$) a fully general polynomial-time generator of unsatisfiable problems cannot exist. Furthermore, if we move from SAT to QBF instances by allowing quantifiers, a fully general polynomial-time generator cannot exist even for the satisfiable case.

Example: time-bounded differential compiler testing. As a final motivating example, a campaign for randomised differential compiler testing typically requires (program, input) pairs where the program *does* terminate on the input. To make such a campaign efficient we might further want the generator to provide a time bound for each generated program, avoiding the need for arbitrarily-selected timeout values. We would thus like to generate (p, i, k) triples comprising a program p that is guaranteed to terminate on input i within k time steps[1] (see Figure 1c). Although this input space *is* generable, we show that unless standard complexity-theoretic assumptions turn out to be false, a time-efficient generator *cannot* cover the set of all such programs: no generator of independent samples exist that uses $q(n)$ time to generate programs of size n, for some polynomial q. Not only that, polynomial generation remains out of reach even if the generator had access to an efficient SAT solver. The results established in Section 6 show that polynomial space $q(n)$ would

[1]Assuming that each basic operation of the program takes one time step.

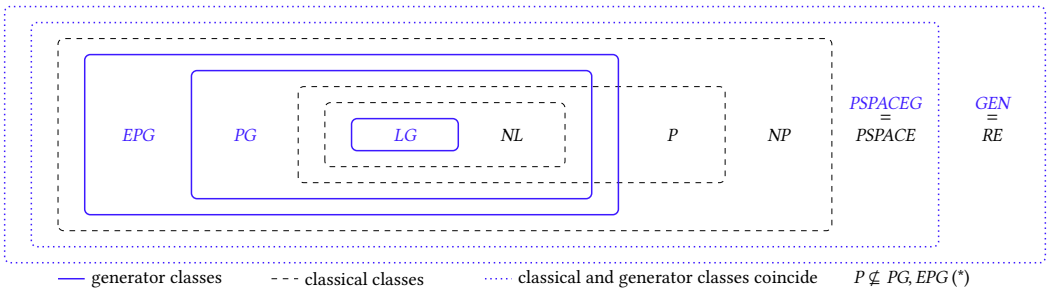

Fig. 2. Complexity relationships between the new generators classes and known classes. Purple dotted boundaries denote generator classes that do not coincide with classical ones; purple solid boundaries denote generator classes that do coincide with classical ones; black dashed boundaries denote classical complexity classes that do not coincide with generator classes. The generator classes are *LG*: logspace generable, *PG*: polynomial-time generable, *EPG*: expected polynomial-time generable, *PSPACEG*: polynomial-space generable, and *GEN*: all generable languages. *P* contains *NL*, intersects *PG* and *EPG*, and is contained in *NP*. (*) Non-containment of *P* is contingent on standard cryptographic assumptions.

also not be sufficient to produce all such programs of size n—limiting the use of feedback-driven generation techniques such as concolic execution [7, 8] and coverage-guided fuzzing [5] for this particular set. A practical solution would have to make some engineering trade-offs.

Beyond classification, our framework yields a *characterisation* of efficient generability: we show that a language has a polynomial-time generator if and only if it has a verifier admitting a *certificate scheme*—a way to sample certificates and reconstruct random instances around them. This shows that *some form* of witness planting (including one based on a trivial verifier)—the folklore technique behind generators for SAT and graph reachability—is used by *all* time-efficient generators.

Separately, a long-standing goal in the PBT literature is to automatically derive efficient generators from predicates [9, 16, 28]. We prove a negative result in this context: under standard assumptions, no library can *compositionally* map predicates from a logic containing conjunction or negation to polynomial-time generators—any homomorphic translation must give up either expressivity or efficiency. On the other hand, we show that restricted classes like *NL* (equivalently, predicates in linear Datalog) can in principle be compiled compositionally to polynomial-time generators.

By studying the power of generators under various resource constraints, we establish a wide range of results about testing that fall out naturally. Some confirm intuitions from testing folklore; others are novel insights that were previously inaccessible without such a formal model.

Contributions. In summary, our main contributions are:

- We provide the first complexity-theoretic foundations of generators for randomised testing (Section 2), and derive expressivity bounds applicable to all generators (Section 3).
- We propose a notion of time/space constraints for generators, and derive theoretical limits of efficient generation (Section 4, Section 5, Section 6).
- We establish general principles for constructing time-efficient generators, and limitations on the design of composable testing libraries (Section 7).

Throughout the paper, we introduce several generator-specific complexity classes: *LG* (log-space generable), *PG* (polynomial-time generable), *EPG* (expected polynomial-time generable), *PSPACEG* (polynomial-space generable) and *GEN* (everything that can be generated). Figure 2 serves as a roadmap to the complexity landscape established in our work.

2 Modelling Generators

A *generator* in randomised testing produces structured inputs for an SUT [35], e.g. inputs that the SUT is supposed to accept, or deliberately malformed inputs designed to stress the SUT. These structured inputs can be viewed as strings in a language. For generality, and to allow the application of complexity theory, we formulate generators as Turing machines, and their outputs as string encodings of data. We model randomness as an explicit input bitstring supplied to a deterministic Turing machine, where random selections are made during computation by reading from this bitstring. This is in line with how randomness is implemented in common randomised testing libraries such as Hypothesis [33].[2]

In practice, generation is driven by a *pseudorandom number generator* (PRNG), which we idealise as an inexhaustible supply of independent, unbiased random bits: a generator consumes some finite prefix of an infinite stream and produces an output. There are two caveats with the informal description, which we make formal below. First, we permit a generator that produces nothing for *every* stream, characterising generators of the empty set, and is useful for our theoretical model. Second, a generator being productive "given an infinite stream" is too strong if interpreted as "given *every* infinite stream". We show in Section 2.3 that reasonable generators can still fail to produce a value in extreme, albeit probability 0 events—so we require producivity only *almost surely*.

Although it is natural to define generators to take infinite streams as input, infinite streams are not a natural input space for Turing machines. To reconcile this, we proceed in three steps:

(1) We define *bitstring transducers*, which operate on *finite* bitstrings, together with a *halting* condition. A halting transducer either accepts, producing an output, or rejects, where a rejection only occurs when the transducer runs out of its finite input (Section 2.1).
(2) We lift a halting transducer to infinite streams. Because rejection only occurs after input exhaustion, acceptance is monotone under extension of the input, thus every accepting run belongs to an equivalence class represented by a unique minimal accepting prefix. This lets us define what a transducer does on an infinite stream, and lets us *measure* the set of accepted bitstreams (Section 2.2).
(3) We define the notion of *eventual productivity* as almost-sure acceptance of infinite streams under the measure of infinite sequences of fair coin-flips, and obtain the definition of a generator (Section 2.3).[3]

Preliminaries. Unless stated otherwise, all tapes of a *Turing machine* (TM) are assumed to be infinite (to the right), initially empty, with the head pointed at some fixed, designated initial cell. The alphabet on all tapes of Turing machines is finite, and includes the "$\sqcup$" symbol to represent blank spaces. The blank symbol will never be written. When we make the statement "tape with alphabet Γ", we implicitly mean it has alphabet $\Gamma \uplus \{\sqcup\}$, and all empty cells on the tape are populated by $\sqcup$ symbols. We say a tape is read-once-only if the head never moves left and the machine never writes to the tape. Similarly, a tape is write-once-only if the head never moves left, and after any write operation the tape head is immediately moved to the right. For a Turing machine M with a single input tape, we say "M ran on x", or "M ran with input x", or "$M(x)$", to mean the computation performed by M in the initial state, with the input tape populated using x starting from the initial cell. This also extends to machines with multiple input tapes. For example,

[2] It is also possible to model randomness using nondeterministic Turing machines, which would result in an equivalent model. We choose deterministic Turing machines with an explicit random input to stay in line with practical implementations, and because it makes later definitions more convenient. The results for the deterministic case translate directly to the nondeterministic case.

[3] In Appendix A.1 we discuss an alternative natural notion of eventual productivity based only on Turing machines.

on a machine M with two input tapes, $M(x, y)$ means the computation performed by M in the initial state, with the first input tape populated using x and the second using y.

2.1 Transducers on finite bitstrings

Definition 2.1 (Bitstring Transducer). *A* bitstring transducer T *is a Turing machine with three tapes: a read-once-only* input tape *with alphabet* $\{0, 1\}$*, a* working tape *with alphabet* Σ*, and a write-once-only* output tape *with alphabet* Γ*. For a bitstring* $b \in \{0, 1\}^*$*, if* $T(b)$ *accepts, the contents of the output tape (disregarding empty cells)* $x \in \Gamma^*$ *are said to be* produced *by* T*, denoted* $T(b) = x$*.*

If there is no ambiguity, we may say transducer for short.

Definition 2.2 (Random Selection). *During an execution, we say that the transducer* T *makes a* random selection *whenever a transition moves the input tape head right—that is, it reads a bit from the input bitstring.*

Multiple random bits can be combined together in a *procedure* to make more refined random selections (e.g. conditional dependence, or selections among more than two possibilities). Procedures are computations (potentially with side effects) that consume randomness in the sense of Definition 2.2, and generators are special cases of procedures that write values to the output tape. Procedures are used to modularise algorithmic descriptions and help with clarity of exposition.

Example 2.1. *A transducer* N *can select any* $n \in \mathbb{N}$ *at random via the following iterative procedure.* N *reads the input in pairs of bits. In each round it reads a* continuation bit*; if it is* 1*,* N *reads a further* data bit*, appends it to a buffer on the working tape, and begins a new round; if it is* 0*,* N *halts and outputs the buffer, interpreted as a binary numeral. Writing the input stream as* $b_0 b_1 b_2 \ldots$*, the transducer computes* $\sum_{i=0}^{k-1} b_{2i+1} 2^i$ *where* $k = \min\{i : b_{2i} = 0\}$*, whenever this* k *exists. Note that the only way* N *rejects is when it runs out of input bits, e.g. on* 1 *or* 101 *(the continuation bit promises another bit that is not supplied), or on* 10 *(no continuation bit is supplied).*

The accepting runs of Example 2.1 consume $2k + 1$ bits for some $k \geq 0$, and rejections occur due to the stream ending prematurely. This is the behaviour we now generalise to our *halting* condition.

Definition 2.3 (Halting Transducer). *A bitstring transducer* T *is* halting *if*

(1) *for every input* $b \in \{0, 1\}^*$*,* $T(b)$ *halts; and*
(2) T *rejects iff the input head attempts to read a blank cell (a cell containing* $\sqcup$*).*

The second clause prevents premature rejections and is a necessary precursor for the formal definition of "eventual productivity" below. From Definition 2.3, we can immediately observe the following, useful for the upcoming generalisation to infinite streams.

Observation 2.1 (Monotonicity). *Let* T *be halting and suppose* $T(b)$ *accepts. Then the run* $T(b)$ *reads no cell beyond the* $|b|$*-th, since by Definition 2.3 reading a blank would force rejection. Hence for every* b' *having* b *as a prefix, the run of* T *on* b' *is identical, so* $T(b')$ *accepts and* $T(b') = T(b)$*.*

2.2 From finite bitstrings to infinite streams

By Observation 2.1, the accepting behaviour of a halting transducer T is completely determined by the shortest bitstrings on which it accepts.

Definition 2.4. *For a halting transducer* T*, the* minimal accepted bitstrings A_T *are*

$$A_T = \{\, b \in \{0,1\}^* : T(b) \text{ accepts} \;\wedge\; \forall b'.\, b' \text{ a strict prefix of } b \implies T(b') \text{ rejects} \,\}.$$

Again by Observation 2.1, no element of A_T is a prefix of another, i.e. A_T is prefix-free, and T accepts a finite bitstring b exactly when some element of A_T is a prefix of b. Note that A_T is defined for conceptual convenience, and is not intended to be computed directly.

We can now assign a natural semantics of T on an infinite bitstream $bs \in \{0,1\}^\omega$: its behaviour is identical to $T(b)$, where $b \in A_T$ is a prefix of bs, if b exists.

Definition 2.5. *For a halting transducer T and an infinite bitstream bs, define $T(bs)$ as:*

(1) *$T(b)$, if $b \in A_T$ and b is a prefix of bs;*
(2) *divergent, if no $b \in A_T$ is a prefix of bs.*

This is well-defined: as A_T is prefix-free, at most one $b \in A_T$ is a prefix of bs.

To speak of the probability that T produces an output, we use the standard measure of infinite sequences of fair coin-flips, which models the PRNG that drives generators in practice.

Definition 2.6 (Distribution of Random Bitstreams). *Let $\mathcal{I}$ denote the distribution of bitstreams $\{0,1\}^\omega$, where for a random variable X distributed according to $\mathcal{I}$ (written $X \sim \mathcal{I}$), all bits of X are independently and identically distributed (i.i.d.) following a fair Bernoulli distribution* $\mathrm{Ber}(0.5)$. *For a set of bitstreams $B \subseteq \{0,1\}^\omega$, denote the probability of B under $\mathcal{I}$ by $\mu(B)$.*

Definition 2.7. *For a bitstring $b \in \{0,1\}^*$, define the* cylinder S_b *as:*

$$S_b = \{b +\!\!+ bs : bs \in \{0,1\}^\omega\}.$$

Definition 2.8. *The* accepted *infinite bitstreams of a halting transducer T are*

$$A_T^\omega = \bigcup_{b \in A_T} S_b.$$

Since A_T is prefix-free, the cylinders S_b for $b \in A_T$ are pairwise disjoint. A_T is also countable, thus A_T^ω is a countable disjoint union of cylinders, and is measurable with

$$\mu(A_T^\omega) = \sum_{b \in A_T} 2^{-|b|}.$$

This quantity is the probability that T, when driven by an idealised PRNG, produces an output.

2.3 Eventual productivity

We are now ready to formalise the "eventual productivity" requirement of a generator. The strongest interpretation of eventual productivity is that $T(bs)$ is defined for *every* $bs \in \{0,1\}^\omega$, equivalently $A_T^\omega = \{0,1\}^\omega$. However, this is too strong in practice. Consider the natural number transducer N of Example 2.1. It diverges on exactly the streams in:

$$X = \{b_0 b_1 \ldots : \forall i \in \mathbb{N}.\ i \text{ even} \Rightarrow b_i = 1\},$$

that is, streams in which every continuation bit is 1, so that N never stops asking for another data bit. Ruling out transducers like N would make it impossible to sample from an infinite set—any transducer with infinitely many possible outputs must have infinitely many accepting prefixes, and so there must exist streams on which the transducer is never productive. This would severely limit the applicability of our model.

Observe however, that the set X has trivial measure, as $\mu(X) = \lim_{i \to \infty} 2^{-i} = 0$. We therefore formalise *eventual productivity* as *almost-sure* acceptance.

Definition 2.9 (Generators). *A halting transducer G is a* generator *if $\mu(A_G^\omega) = 1$ or $\mu(A_G^\omega) = 0$.*

Observation 2.2. *$\mu(A_G^\omega) = 0$ iff $A_G = \emptyset$ iff G rejects every finite bitstring.*

Proof. Each S_b has $\mu(S_b) = 2^{-|b|} > 0$, so $\mu(A_G^\omega) = \sum_{b \in A_G} 2^{-|b|} = 0$ iff A_G is empty. By Observation 2.1, $A_G = \emptyset$ iff G accepts no finite bitstring. □

The two cases ($\mu(A_G^\omega) = 1$ or 0) are for the two behaviours permitted of generators: the first characterises almost-sure productivity; the second describes the case where all inputs are rejected (corresponding to a generator for the empty set). It is undecidable to check whether a given transducer is a generator, but this is not an issue since we show generator conditions on a case-by-case basis throughout this work.[4] For added intuition, it may help to consider the transducers that are *not* generators. These are the transducers that fail to halt on some finite bitstring, or that diverge on a non-negligible — but not co-negligible — set of streams. To illustrate the definition, we verify that it matches our intuitions on Example 2.1, and that rejection sampling yields true generators.

Proposition 2.1. *The transducer N of natural numbers described in Example 2.1 is a generator.*

Proof. N is clearly halting: on any finite bitstring it either meets a 0 continuation bit and accepts, or exhausts its input and rejects. The set of bitstreams *not* accepted by N is X as above, so $A_N^\omega = \{0,1\}^\omega \setminus X$ and $\mu(A_N^\omega) = 1 - \mu(X)$. Since $\mu(X) = 0$, $\mu(A_N^\omega) = 1$ and N is a generator. □

Definition 2.10 (Generator by Rejection Sampling). *Let f be a total computable predicate on $\{0,1\}^*$. The generator G_f based on rejection sampling on f is defined as follows.*

(1) *Sample an n (e.g. using Example 2.1), then sample a string w of length n.*
(2) *Compute $f(w)$. If $f(w)$ returns true, output w. Otherwise restart from step 1.*

Proposition 2.2. *If f is a computable predicate on $\{0,1\}^*$ such that f holds for at least one $w \in \{0,1\}^*$, then G_f is a generator.*

Proof. See Appendix A.2. □

3 A Framework for Generability

In testing, randomness is an essential tool to enable systematic exploration of the input space. The goal of randomised testing can be viewed as the validation of a certain property that must hold on the program when executed on the random inputs. The commonly practised paradigm of *property-based testing* (PBT) validates properties of the form $\forall x.Pre(x) \implies Post(x)$, which suits the vast majority of use cases. For instance, fuzzing can be seen as validating the property: "for all inputs x, the program should not crash". Differential testing can be seen as validating: "for all valid inputs x, the two implementations T, T' must have $T(x) = T'(x)$".

The key to the successful application of randomised testing is an effective generator, whose job is to generate inputs that satisfy the preconditions of the property, which then enables the property to be checked on the program. We now show a few example scenarios below of programs and their properties that may be tested, and will be referring back to them throughout this section.

Example 3.1 (Binary Search Trees). *Program P takes as input a binary search tree T and an index i, and outputs the ith smallest element within the tree. P should satisfy the following property, where $bst(T)$ is a predicate checking that T is a binary search tree, and i, j are integers:*

$$\forall T, i, j.bst(T) \wedge 0 \leq i \leq j < |T| \implies P(T,i) \leq P(T,j)$$

Example 3.2 (Counting SAT Assignments). *Program P takes as input a CNF boolean formula ψ and outputs the number of satisfying assignments of ψ. P should satisfy the following property, where sat is the predicate for the satisfiability of the formula, and $\psi[x/A]$ is the substitution of A for x in ψ:*

$$\forall \psi.sat(\psi) \implies P(\psi) > 0 \wedge P(\psi) = P(\psi[x_0/T]) + P(\psi[x_0/F])$$

[4] In a similar vein, the undecidability of the Halting problem does not prevent people from developing useful algorithms.

Example 3.3 (Tensor Decomposition). *Program P takes a tensor T over field F, and outputs a decomposition of T as a product of rank-1 tensors. P should satisfy the following:*

$$\forall T, r.\ rank(T) = r \implies length(P(T)) = r \ \wedge\ (\forall i < r.\ rank(P(T)_i) = 1) \ \wedge\ \bigotimes_{i<r} P(T)_i = T.$$

Example 3.4 (C Compiler). *Program P is a compiler for the C language, taking an optimisation flag (0 or 1) as the first argument and a C program as the second argument. P should satisfy the following property, where $wd(c)$ is a predicate checking c is a C program with well-defined semantics (assume it doesn't take inputs), and $P(o, c)()$ denotes executing the executable produced by compiling c using P on optimisation o:*

$$\forall c.wd(c) \implies P(0, c)() = P(1, c)()$$

In each case, for a property $Prop = \forall x.Pre(x) \implies Post(x)$, the ideal generator should implement a distribution on the elements of the set $Precond = \{x : Pre(x)\}$.

3.1 The Design Space of Generability

We first address some nuances in the design space of the formal language associated with a generator. This is an important step, since languages are the central object of study in any complexity theory.

Allow generators to sample only a subset of Precond? Sampling from a strict subset of *Precond* would mean there exist elements $x \in Precond$ that can never be generated. Such a generator can still find bugs, but never ones triggered only by inputs outside its range. Thus the generator of a subset of *Precond* would not be appropriate to *fully* test *Prop*.

As a pragmatic choice, it is common in practice to create generators that sample from only a subset of inputs that satisfies a property's precondition. For instance, when the precondition is seemingly too complex[5] to write a generator for, such as the property described in Example 3.4, generators typically focus on specific subsets of the input space heuristically understood to contain potential bug triggers [12, 31]. Although existing literature refers to these as "generators" of C-programs for validating compilers, the property they are testing for is more accurately understood as a different from the one in Example 3.4, with a more restrictive precondition:

$$\forall c.producible(c) \implies P(0, c)() = P(1, c)()$$

where *producible* is to denote the predicate corresponding to the targeted subset of inputs the generator produces. For example, restricted C-programs amenable to implemented analyses [50], or C-programs containing constrained forms of loops [32].

Allow generators to sample a superset of Precond? Generators sampling from a superset of *Precond* would be able to trigger all possible property violations, but risk false positives from inputs outside of *Precond*. Rejection sampling is required to filter the inputs, selecting only ones that satisfy the precondition. Depending on requirements, this filter can be applied either as a final step of generation, or after a bug-triggering input has been found to check for false-positivity.

While filtering works well when deciding the precondition is easy (Example 3.1), the precondition itself can be hard to decide, and is not in general related to the complexity of the algorithm under test. For example, checking the precondition is *NP*-hard in Example 3.2 and Example 3.3, and undecidable in Example 3.4 (deciding if a C program is well-defined is undecidable [11]). Example 3.3 is worth highlighting here: computing the rank of T is *NP*-hard, but if $rank(T)$ is known, the decomposition is computable in time polynomial in the size of the tensor. The runtime of the test campaign can

[5]Note this is only an empirical understanding, which is thus far not formalised. When is a set really *too complex* to generate? We will look to give theoretical answers to this question in the coming sections.

thus be dominated by the filtering stage. These filters also need to be provided separately, and when the filter is complex, it can be an additional source of complexity and bugs in the testing system.

Moreover, test case throughput is an important factor for the effectiveness of a testing campaign [4, 34]. Spending significant time deciding the validity of inputs can often mean fewer or missed bugs. If a precondition-satisfying input is produced rarely by a generator, for instance, using a generator of random ASTs (or even random strings) for Example 3.4, then it can be the case that no tests are ever run, due to the low probability of finding a satisfying input! Indeed, if the generator produces precondition-satisfying inputs with probability p, then one expects $1/p$ attempts to find a usable input—and $1/p$ can easily scale exponentially with the size of the generated test case. In the Example 3.4 case, a random AST would have vanishingly low probability of satisfying the language specification [23], and a random string even more so.

What distributions should generators implement? A generator implements a particular distribution over the set of inputs, and in practice impacts the effectiveness of a testing campaign. Randomised testing employs a wide range of distributions, which can also be adjusted as testing progresses [18, 36]. However, there is no canonical distribution around which to build a complexity theory. The uniform distribution (on elements of a given size) is commonly studied theoretically, but fail to capture the diversity of distributions used in practice. Indeed, in the design of generators, the distribution implemented is often an orthogonal concern to samplability itself.

Therefore, as a first step towards a complexity-theoretic model of generators, we remain agnostic to distributions, and aim to gain fundamental understanding of *which* sets can be generated at all. This serves as a prerequisite to the harder question of generability under *prescribed* distributions, and also ensures generality of our results. The "no-go" results we derive, a type of result common in complexity theory, apply to generators implementing *any* computable distribution, rather than being contingent on distributional assumptions. The only requirement we impose is *distributional support* [37]: the support of the implemented distribution must equal the whole set.

One natural concern is that the weak assumption of only distributional support may be overly permissive, since it allows distributions that assign negligible weight to parts of the set. However, distributional support is a stronger guarantee than it may first appear—once an input is producible, however improbably, practical techniques can often amplify the probability of producing desirable outcomes. For example, the bitstring input to a generator can be viewed as an optimisation parameter, and this insight was used in previous works [17, 33, 36] to tune generators, producing outputs that would have otherwise been infeasible due to their low probability weights.

3.2 Unconstrained Generability

In light of the insights from the above discussion, we give the following definition for the language associated with a generator.

Definition 3.1 (Generated Language). *For a generator G, the* language generated by G *is:*

$$\mathcal{L}(G) = \{x \mid \exists\, b \in \{0,1\}^+ \,.\, G(b) = x\}$$

The following is the most general class of languages that is generable. The only requirement is that there *exists* an algorithm (generator) to generate the elements.

Definition 3.2 (Generable Languages). *For an alphabet Σ, we define the set of* generable languages*, GEN, as follows:*

$$GEN = \{\mathcal{L}(G) : G \text{ is a generator}\}$$

A language L is generable *if $L \in GEN$.*

For a language L, we have defined our generators to associate with L a surjective computable function $\{0,1\}^* \to L$ that constructs instances of L from randomness. On the other hand, one of the most commonly studied objects in complexity theory is Turing recognisers, which implement a (potentially partial) predicate for some language L. Turing recognisers associate a language L with a function of type $\Sigma^* \to \mathbb{B}$, which accepts $x \in \Sigma^*$ iff $x \in L$.

The standard class of *RE*, or recursively enumerable languages, is the set of all languages whose elements can be recognised by a Turing machine (predicate).

Definition 3.3 (Recursively Enumerable Languages). *For a given alphabet Σ, the recursively enumerable languages are:*

$$RE = \{L \subseteq \Sigma^* : \exists \text{ a Turing machine } M.\ \forall x \in \Sigma^*, \quad x \in L \Leftrightarrow M \text{ halts and accepts } x\}$$

In Definition 3.3, M refers to a standard Turing machine that accepts/rejects an input, rather than the transducer style within Definition 2.1. M can be seen as a membership predicate for some set S, where $M(x)$ accepts if and only if $x \in S$ (but importantly may loop instead of rejecting if $x \notin S$). The *RE* sets/languages with a membership predicate algorithm that always halts are called *decidable*, and the complement of the decidable languages in $\mathcal{P}(\Sigma^*)$ is called *undecidable*.

The well-definedness of a C-program (*wd* in Example 3.4) is an undecidable problem, making it a "hard" member of *RE*. As noted earlier, the natural approach of rejection sampling cannot be used within any generator of well-defined C-programs (programs x satisfying $wd(x)$), since generators must terminate on any finite input, and the test for a string x satisfying $wd(x)$ may never terminate. The inability to use rejection sampling of course extends to all *RE* languages.

This raises the natural question: is it actually possible to write a generator for well-defined C-programs, or indeed any undecidable set? The question is not limited to a single generation algorithm that one may write: any conceivable procedure, or combinations of procedures whose set of outputs equals the set of well-defined C-programs would be a generator.

We now show that in the resource unconstrained case, the generable languages are precisely the recursive enumerable languages. This is consistent with the known fact that standard transducers have images equalling the *RE* languages—the additional constraints of a generator do not sacrifice any expressive power. However, for testing this already leads to interesting conclusions, which we discuss in more detail later in the section: programs such as compilers often do deal with undecideability, and this places fundamental constraints on the types of properties that can be fully tested on such programs. Later in Section 5 and Section 6, we will see that additional resource (time or space) constraints lead to situations where the generator and their corresponding decision complexity classes do not coincide.

Theorem 3.1 (Generability is Recursive Enumerability). *$RE = GEN$.*

Proof. We defer the full proof to Appendix A.4. Intuitively, $GEN \subseteq RE$ is due to the *RE* machine being able to enumerate through all finite bitstrings (as random seeds b) to check if the generator can produce some x. $RE \subseteq GEN$ is due to the generator being able to perform dovetailing to find a single accepted string by the recogniser, which can then be used as fallback for a nondeterministic guess of the membership certificate for some randomly chosen x. □

Consequences for PBT: *When testing a property for which the set of acceptable inputs is recursively enumerable, Theorem 3.1 implies that a generator for the complete input format exists—i.e., ambition in generator-writing is at least theoretically justified. However, properties whose preconditions are not recursively enumerable cannot be fully tested by any generator, no matter how it is engineered.*

For instance, it is possible to write a generator that will output, with non-zero probability, all instances of seemingly very hard problems, such as:

Definition 3.4. *The language of halting computations on Turing machines:*

$$HALT = \{\langle M, w\rangle : \textit{Turing machine } M \textit{ halts on input } w\}$$

The equality also yields a class of non-generable languages—ones that lie outside of *RE*.

Corollary 3.1. *All languages outside of RE are not generable.*

Non-*RE* languages are a large set, and one way to obtain languages in it is via complementing languages known to be in *RE*. A language L is undecidable if there is no Turing machine that halts on all inputs and accepts exactly the strings in L. Rice's theorem [39] is a well-known result that states all (non-trivial) semantic properties on Turing machines are undecidable. Hence, at least one of the language itself, or its complement, is non-*RE*. The result also extends from Turing machines to Turing-complete programming languages, and we will quote this version.

For a Turing-complete programming language K, let P_K be the set of encodings of well-defined programs written in K, and for $M \in P_K$ let $\mathcal{L}(M)$ be the set of strings w such that $M(w)$ executes without rejection (crashes or errors). A *semantic property* is a set $S \subseteq P_K$ that depends only on program semantics: if $\mathcal{L}(M) = \mathcal{L}(N)$ then $M \in S \iff N \in S$. It is *non-trivial* if $S \neq \emptyset$ and $S \neq P_K$, and its complement is $S^c = P_K \setminus S$. Rice's theorem [39] implies every non-trivial semantic property is undecidable, yielding the following:

Corollary 3.2. *Let $S \subseteq P_K$ be a non-trivial semantic property over programs written in a Turing complete language K. Then at least one of S or S^c is not in GEN.*

Proof. Suppose for a contradiction both S and S^c are generable, then both are also recursively enumerable by Theorem 3.1, thus S is actually decidable. This contradicts Rice's Theorem. □

These results are particularly applicable to software that can come face-to-face with non-*RE* languages, such as compilers. In this context, it means that there are properties of the compiler implementation that can never be completely tested, due to the lack of a generator for the property. This includes the use of rejection sampling. For example, in C++, termination is a property relevant for the well-definedness of programs, defined by the language specification. One might be interested in building a generator to test the compiler (its inputs being programs) for the following: programs that do not terminate should nonetheless not crash the compiler. However, although the terminating programs are generable, there are no generators of all non-terminating programs:

Corollary 3.3. *For a Turing-complete language K, the set of non-Halting programs, LOOP, is not generable, where*

$$LOOP = \{M \in P_K : M \textit{ takes no inputs and does not terminate}\}$$

Consequences for PBT: *You cannot fully test compiler properties with non-RE preconditions, e.g., the behaviour on non-terminating programs (Corollary 3.3). Any generator of a producible subset is testing a* different, weaker *property—a trade-off that should be made explicitly, not discovered accidentally.*

Of course, the existence of generators is only one of the concerns of PBT practitioners. We now turn to another important consideration: their *efficiency*.

4 Generator Families

We cannot wait all day for a generator to produce an output for an SUT: an effective generator needs to rapidly produce a large corpus of inputs for the SUT. Thus, as with algorithms, it is helpful to associate resource constraints with generators. But, unlike algorithms, generators do not have a natural input parameter on which the input size can be defined, and hence there also lacks a definition of resource constraint, which are based on the input size.

In practice, PBT libraries like QuickCheck indeed include an optional *size* parameter, which a generator implementation can use to control the sizes of the generated objects.

We introduce a similar parameter in our model, which allows a priori specification of the output size—from which we can then define the notion of resource usage for generators. Adding these constraints would allow our model to be applicable to generators that use the size parameter.

The following aims to capture the notion of parameterised generators, namely a family of generators that each generates a portion of the intended language (instances of a particular size), and where the union of their generated languages is the whole language.

Definition 4.1 (Implementation of a Family of TMs). *Let $(M_i)_{i\in C}$ be a family of TMs indexed by strings $C \subseteq \{0,1\}^*$. We say that M implements $(M_i)_{i\in C}$, written $M = (M_i)_{i\in C}$, if $(M_i)_{i\in C}$ is a uniform family as follows:*

- *M has an extra read-only input tape compared to each M_i.*
- *For all $i \in C$, $M(i) = M_i$. That is, with i on the input tape of M, M behaves exactly as M_i.*

Definition 4.2 (Family of Generators). *Let L be a language, and $(L_i)_{i\in C}$ a partition of L indexed by the set $C \subseteq \{0,1\}^*$. $G = (G_i)_{i\in C}$ is a generator for L if for each $i \in C$, $\mathcal{L}(G_i) = L_i$.*
The language generated by G, $\mathcal{L}(G)$ is defined as $\mathcal{L}(G) = \bigcup_{i\in\mathbb{N}} \mathcal{L}(G_i)$.

For the special case where $C = \{1^i : i \in \mathbb{N}\}$ is used to partition a set L into $L_i = \{w \in L : |w| = i\}$, we will also denote C as $\mathbb{N}$, and L_{1^n} as L_n, for brevity. This will be used when we consider length-based partitions of a language L in the following sections.

Using this, we can define the generable languages under space and time constraints.

Definition 4.3. *For a time constructible function $t : \mathbb{N} \to \mathbb{N}$, the generator family $G = (G_i)_{i\in\mathbb{N}}$ is time bounded by t if for all $i \in \mathbb{N}$, G_i terminates within time $t(i)$.*
Define the class of languages $GTIME(t(n))$ generable by some t-time bounded generators as:

$$GTIME(t(n)) = \{\mathcal{L}(G) : \forall i.G_i \text{ is time bounded by } t(i)\}$$

Definition 4.4. *For a space constructible function $s : \mathbb{N} \to \mathbb{N}$, the generator family $G = (G_i)_{i\in\mathbb{N}}$ is space bounded by s if for all $i \in \mathbb{N}$, G_i uses no more than $s(i)$ cells on its work tape.*
Define the class of languages $GSPACE(s(n))$ generable by some s-space bounded generators as:

$$GSPACE(s(n)) = \{\mathcal{L}(G) : \forall i.G_i \text{ is space bounded by } s(i)\}$$

The following is immediate for both time and space bounded generators:

Proposition 4.1. *Suppose $s : \mathbb{N} \to \mathbb{N}$ is space-constructible with $s(i) > \log(i)$, and $t : \mathbb{N} \to \mathbb{N}$ is time-constructible with $t(i) > i$, then:*

$$GSPACE(s(n)) \subseteq NSPACE(s(n)) \qquad GTIME(t(n)) \subseteq NTIME(t(n))$$

Proof. By simulation. For a full proof, see Appendix A.5. □

4.1 Alternative Notions of Size

In our formalisation, a generator G for a language L will produce, on input n, a random element $x \in L$ of length *exactly* n. We discuss here some alternative possible formalisations of instance size, inspired by testing practice.

4.1.1 On generating elements of size $|x| \leq n$. Some generators in practice produce, given an $n \in \mathbb{N}$, elements of size $|x| \leq n$ rather than $|x| = n$. These generators can fit within our framework by observing that such generator G of the language L can be mapped to an equivalent generator of the following language:

$$L_{pad} = \{x\#0^i : x \in L \wedge i \in \mathbb{N}\}$$

where 0^i denotes i repetitions of the reserved element 0, and # is a dedicated separator element. If L_{pad} is generable by the $|x| = n$ scheme (for a given n, produce elements of length exactly n), then L is generable by the $|x| \leq n$ scheme (for a given n, produce elements of length at most n). Moreover, observe that the decision complexity of L_{pad} is the same as L, hence the later no-go results from connections to decision complexity classes also applies for $|x| \leq n$ generation schemes.

4.1.2 Other parameterisations of size. Apart from string length, there can also be other concievable notions of size for a language L, in forms of some computable function $sz : L \to \mathbb{N}$. For instance, for CNF formulas, the number of clauses or variables can both be useful notions of size.

However, it is difficult to define notions of complexity on metrics like the clauses or variables count. This is because formulas with a fixed number of clauses or variables can, in general, describe formulas of arbitrary size in terms of their string representations. Grouping these together in a complexity-theoretic formalisation would mean assigning the same measure of complexity to all such elements—yet the arbitrary lengths of their string representations would genuinely take vastly different (particularly time) resources to produce.

For sz based on the number of clauses or variables, the string representations of x with $sz(x) = m$ has unbounded ratios: $\forall n \in \mathbb{N}.\exists m \in \mathbb{N}.\exists x \in L_m.\frac{|x|}{m} > n$, where $L_m = \{x \in \Sigma^* : sz(x) = m\}$. Alternatively, consider another sz where $sz(x)$ is within some range of $|x|$, e.g.

$$(1 - \delta)|x| \leq sz(x) \leq (1 + \delta)|x|$$

In these cases, we can recover the fixed-length paradigm by padding all elements with $sz(x) = m$ to a string of length exactly $(1 + \delta)m$: the sz-m elements are generable in the sz-based paradigm if and only if length $(1 + \delta)m$ elements are generable in the fixed-length paradigm.

5 Polynomial-Time Generators

In this section, we discuss generators that run in polynomial time, which corresponds to the class of realistically implementable generators.

We note that prior work [44, 45] also considered polynomial-time generators, and actually predates PBT [10] itself. In the course of establishing a number of new results, we will also walk through some previously known results (Corollary 5.1, Proposition 5.1, Proposition 5.3) within our framework to highlight the links between modern-day PBT and complexity theory.

An equivalent version of the following definition was first introduced in Definition 3.1 of [44].

Definition 5.1 (Polynomial-Time Generators). *The generator family $G = (G_i)_{i \in \mathbb{N}}$ runs in polynomial time if there exists a polynomial p such that for all $i \in \mathbb{N}$, G_i terminates in time $p(i)$.*

Remark 1. We do not need the empty set clause of Definition 3.1 in [44], because we define the generator of an empty set as one that rejects all inputs, and we can use its rejection within $p(i)$ steps (on a length $p(i)$ input bitstring) to determine the emptiness of $\mathcal{L}(G_i)$. On the other hand, if $\mathcal{L}(G_i) \neq \emptyset$, then it can never reject when provided with a length $p(i)$ input bitstring.

Definition 5.2. *The class of languages generable by some polynomial-time generator is:*

$$PG = \bigcup_{i \in \mathbb{N}} GTIME(n^i)$$

Generators in randomised testing often use *rejection sampling* to produce generators for properties that are difficult to write explicit generators for. However, this approach trades ease in development for runtime during generation. If a property t is satisfied by the generator with probability p, then the expected samples (time) to find such an instance is $1/p$—and this can easily be exponential if t occurs rarely enough. The polynomial time constraint implies that for realistic generators, the use of such rejection sampling is only possible in controlled cases where $1/p$ is at most a polynomial.

We will first see that the polynomial time generable languages are wholly contained within *NP*.

Corollary 5.1. *PG* $\subseteq$ *NP*.

Proof. Consequence of Proposition 4.1. □

Consequences for PBT: *Corollary 5.1 shows that NP membership of the precondition is a necessary condition for efficient generation: if a precondition is PSPACE-hard or coNP-hard (e.g. quantified formulas, unsatisfiable formulas), there is no hope in seeking a fast, fully general generator—one must resort to rejection sampling or restrictions.*

Example 5.1. *If NP* $\neq$ *PSPACE, then the following PSPACE-complete language is not generable in polynomial time:*

$$QBF = \{\phi : \phi \text{ is a satisfiable quantified boolean formula}\}$$

where a quantified boolean formula is a propositional boolean formula with quantifiers.

Example 5.2. *If NP* $\neq$ *coNP, then the following coNP-complete language is not generable in polynomial time:*

$$TAUT = \{\phi : \phi \text{ is true for all assignments}\}$$

The natural follow up question is whether the containment *PG* $\subseteq$ *NP* is strict. We see that *PG* does contain some of the hardest problems in *NP*, namely *CNF-SAT*.

Proposition 5.1. *CNF-SAT* $\in$ *PG*.

Proof. We give a sketch of the key ideas, and defer full detail to Appendix A.6. Consider a generator that first samples a random assignment $\mathbf{x}$, then generates the formula ϕ based on $\mathbf{x}$ clause-by-clause. For each clause, we pick a random literal to evaluate to *True*, referring to the earlier chosen assignment $\mathbf{x}$. The other literals in the clause are chosen at random. By construction, as ϕ is in CNF, each clause of ϕ is satisfied by $\mathbf{x}$, and thus $\phi[\mathbf{x}]$ also evaluates to *True*.

Any satisfiable *CNF-SAT* formula ψ has a chance of being sampled this way: if ψ is satisfied by $\mathbf{y}$, then the above procedure has a chance of choosing $\mathbf{y}$ and then constructing ψ around $\mathbf{y}$. □

Consequences for PBT: *A precondition that is hard to* decide *can still be easy to* generate*: you can fully test properties on a #SAT solver (Example 3.2) without ever solving SAT, for example by using the generator in Proposition 5.1. Thus, testing difficulty cannot be inferred from decision difficulty.*

However, having an efficient generator for an *NP*-hard language does not directly yield a way to sample from easier languages, due to the lack of natural reductions for generators (as far as we know). The reduction used in *NP*-hardness embeds the easier language into the harder language. For generators, we would like the opposite: to surjectively map from instances of the hard language to the easier one, which is what is needed to implement a generator for the easier language.

In fact, despite having a generator for an *NP*-complete problem, we will show below an example of a language in *NP* (in fact, *P*) that is unlikely to have a polynomial-time generator.

5.1 *PG* vs. *NP*

We now introduce a concrete problem in *NP* that is likely not in *PG*, as its membership in *PG* would contradict cryptographic assumptions that have withstood decades of scrutiny.

Following [43], we will base our construction on so-called collision-resistant hash functions.

Definition 5.3 (Hash Functions [43]). *A hash function h is a collection of functions $(h_n)_{n\in\mathbb{N}}$, $h_n : \{0,1\}^{n+1} \to \{0,1\}^n$, such that there exists a computable function $h : \mathbb{N} \times \{0,1\}^+ \to \{0,1\}^+$ with $h(n,x) = h_n(x)$ for all $n \in \mathbb{N}$, and h runs in time polynomial in n.*

Definition 5.4 (Collision-Resistant Hash Functions). *Hash function h is considered* collision-resistant *if there exists no polynomial-time bounded randomised algorithm $A : \mathbb{N} \to \{0,1\}^* \times \{0,1\}^*$ and polynomial q such that for infinitely many $n \in \mathbb{N}$:*

$$\mathbb{P}(A(n) = (x,y) \wedge x \neq y \wedge h_n(x) = h_n(y)) > \frac{1}{q(n)}$$

If a randomised collision-finding algorithm of Definition 5.4 exists for the polynomial p-bounded hash function h, then one can find size-n hash collisions of h in expected polynomial time.

Remark 2 (Unkeyed hashes and uniform attacks). In cryptographic literature [27], *keyed* hash families $(h_k)_{k\in K}$ are more commonly used, with security against non-uniform attacks. We instead use *unkeyed* hash functions (Definition 5.3) with security against *uniform* attacks (Definition 5.4). Unkeyed hashes capture practical algorithms like SHA-256, but cannot resist non-uniform attacks: a hash collision is guaranteed to exist for each n by the pigeonhole principle, and it is possible to hard-wire them into non-uniform circuits. Uniform attacks [40] are a formalisation of the security guarantees in practice: collisions exist, but there exist no efficient algorithms to find them. As we shall see later, the uniformity of generators and the resistance of unkeyed hashes against uniform attacks are precisely what give us the non-generability result of Proposition 5.2.

Define the following language for a hash function h:

Definition 5.5. *For hash function h:*

$$\mathit{COLLISION}_h = \{(x,y) \in \{0,1\}^+ \times \{0,1\}^+ : x \neq y \wedge |x| = |y| \wedge h_{|x|}(x) = h_{|x|}(y)\}$$

Clearly, for any hash function h (collision-resistant or not), given (x,y), a Turing machine M can execute $h(x)$ and $h(y)$, and accept if they compute the same hash. Since h executes in time polynomial in its input size, M will terminate in polynomial time.

Lemma 5.1. *For all hash functions h, $\mathit{COLLISION}_h \in P$.*

Proposition 5.2. *If for all hash functions h, $\mathit{COLLISION}_h \in PG$, then collision resistant hash functions do not exist.*

Proof. Suppose $\mathit{COLLISION}_h \in PG$, then there exists a polynomial-time generator $G = (G_i)_{i\in\mathbb{N}}$ with $\mathcal{L}(G) = \mathit{COLLISION}_h$. For a given $n \in \mathbb{N}$, we can find a collision of h_n by running G_{2n+1} (assuming 1 character is used as a separator) to obtain a pair $(x,y) \in \mathit{COLLISION}_h$ of size $2n+1$. Note this is possible due to the uniformity of $(G_i)_{i\in\mathbb{N}}$. Since $|x| = |y|$ we know that $|x| = |y| = n$, and hence the pair found is a collision of h_n: $h_n(x) = h_n(y)$. As G_i runs in polynomial time, we have therefore a randomised algorithm that finds hash collisions in polynomial time, for infinitely many n, and with success probability 1. □

Theorem 5.1 (Cryptographic Separation). *If collision-resistant hash functions exist, then $PG \subsetneq NP$. In fact, $P \not\subseteq PG$.*

Consequences for PBT: *Simple software may be infeasible to test efficiently, in full generality. Easy-to-decide preconditions (here, checking a hash collision in Theorem 5.1) can still be difficult to generate satisfying inputs for. Easy decision complexity is neither necessary nor sufficient for generability. (For a classification of what is efficiently generable, see our certificate scheme of Theorem 7.1.)*

So, the power of efficient generators has an interesting relationship with the classical decision classes: *PG* contains some of the hardest problems in *NP*, yet it also does not contain some problems in *P*. We thus see that the complexity of the language for generation is a related but different concept from the complexity of the language for decision algorithms.

Some generators in testing literature also make use of *SAT* solvers [6, 32, 46, 48], to increase expressivity. Here we quote a result first shown in [44]: with a *SAT* oracle, one can generate all *NP* problems in a polynomial number of steps.

Let Σ_i^P denote the ith level of the polynomial hierarchy with an *NP* base machine. For a generator class X with constraints Y and language L, define X^L as the class of languages generable by generators with constraints Y and oracle access to a decider for L. For a generator complexity class X and a decision complexity class C, define $X^C = \bigcup_{L \in C} X^L$.

Proposition 5.3. $\Sigma_1^P = NP \subseteq PG^{NP} \subseteq NP^{NP} = \Sigma_2^P$

Proof. A case from a more general result in [45]. See Appendix A.8 for a proof of this case. □

Consequences for PBT: *Although using a SAT/SMT solver during generation allows NP preconditions to be satisfied, Proposition 5.3 shows it is not a panacea: for example, PSPACE-hard or EXPTIME-hard preconditions remain out of reach even with a perfect SAT or SMT solver, respectively.*

5.2 Expected Polynomial-Time Generators

Since sampling is inherently a random process, another feasible resource constraint on generators is via the *expected time* to produce a sample, rather than a deterministic time bound. The following defines *expected* polynomial time generators:

Definition 5.6. *The generator family $G = (G_i)_{i \in \mathbb{N}}$ runs in expected polynomial time if there exists a polynomial p such that for all $i \in \mathbb{N}$, G_i terminates in expected time $p(i)$. Moreover, for all $w \in \mathcal{L}(G_i)$, there exists at least one bitstream where w is produced before time $p(i)$.*

The condition that there exists *at least one* path of polynomial length for any w ensures that there are no fundamentally difficult instances in the language.[6] Our definition does not collapse to *PG*, since execution paths longer than polynomial are still permitted. Intuitively, expected polynomial time describes repeated retries of a polynomial process with bounded chance of failure: a generator version of *BPP*. On lucky runs, the process takes polynomial time, which corresponds to the existence of a polynomial-length path.

Definition 5.7. *The class of languages generable by some expected polynomial-time generator is:*

$$EPG = \{\mathcal{L}(G) : G \text{ is an expected polynomial-time generator}\}$$

Proposition 5.4. $PG \subseteq EPG \subseteq NP$.

Proof. A generator that terminates in deterministic time $p(i)$ will also terminate in expected time at most $p(i)$, thus $PG \subseteq EPG$. $EPG \subseteq NP$ follows by simulation—for a full proof, see Appendix A.7. □

[6] Such an issue of hard instances hiding in long runs was also noted in [25], where interestingly, the modelling choice made by Jerrum et al. [25] was to *ignore* the expected-time case and only focus on deterministic time bounds.

Consider the language *PRIMES*, defined as follows:

Definition 5.8. *Assume x is interpreted in its binary representation, define:*

$$PRIMES = \{x \in \{0,1\}^* : x > 1 \wedge \forall y. 1 < y < x \implies gcd(x,y) = 1\}$$

We show that $PRIMES \in EPG$: an interesting case as it is an open problem in mathematics to determine whether there exists a *PG* algorithm for *PRIMES* [49]. That is, an algorithm which *always* produces a prime number of a certain size within polynomial time. However, it does have a natural algorithm running in *expected* polynomial time, and also does not follow the structure established for *PG* in Theorem 7.1. It would thus be tempting to conjecture that $PG \subsetneq EPG$.

Proposition 5.5. *$PRIMES \in EPG$.*

Proof. This is a straightforward application of the fact that $PRIMES \in P$ [1], and the Prime Number Theorem on the density of primes. Together, these allow us to use rejection sampling for primes between 2^n and 2^{n+1}. See Appendix A.9 for a full proof. □

Consequences for PBT: *Giving up worst-case polynomial time guarantees can at times result in simpler generator designs, and potentially allow more sets to be sampled (but is an open problem whether this is the case [49]). However, care must be taken to ensure the runtime is truly polynomial in expectation (as in Proposition 5.5), and that no element is intrinsically hard to sample: otherwise the efficiency and soundness (distributional support) guarantees no longer hold.*

Akin to boosting the probability of deciding a language in probabilistic complexity classes, we can make a similar statement on boosting the probability of generating an instance for *EPG*.

Proposition 5.6. *For $L \in EPG$, and any $\epsilon > 0$, there exists an algorithm A that produces a sample $x \in L$ of length n with probability $> 1 - \epsilon$, in polynomial time of n and $\log(1/\epsilon)$.*

Proof. By considering long runs and using Markov's inequality to bound the non-generation probability. See Appendix A.10 for a full proof. □

This boosting result means that for *EPG* languages, we can obtain samples with arbitrarily high probability in polynomial time. Thus, this would also rule out the membership of $COLLISION_h$ in *EPG*, unless collision-resistant hash functions do not exist.

Proposition 5.7. *If collision-resistant hash functions exist, then $EPG \subsetneq NP$. Moreover, $P \nsubseteq EPG$.*

Proof. Suppose for a contradiction that for all hash functions h, $COLLISION_h \in EPG$, then there exists an expected polynomial time generator that generates $COLLISION_h$. By Proposition 5.6, for any $k \in \mathbb{N}$, we can obtain a generator that produces a length-n sample of $COLLISION_h$ with probability at least $1/k$, within polynomial time in n (with ϵ fixed to $1/k$, the $\log(1/\epsilon)$ term becomes a constant factor in the polynomial). Since this is a procedure that violates the collision resistance property (Definition 5.4) of h, no hash functions are collision resistant, which is a contradiction. □

6 Space Bounded Generators

Space-bounded generators have a limited amount of space to work with, but can exhibit arbitrary running time. Whilst this seems like a rather loose restriction on generators, we note that these constraints actually correspond to a large class of generators implemented in practice.

Space-bounded generation can model test-case generation algorithms that produce not only independent outputs in a single run (like in the case of *PG* or *EPG*), but also past-dependent outputs in an iterative fashion. That is, algorithms where previous outputs can *influence* future produced

outputs, for instance by caching some data collected from past outputs to introduce dependence in the generation algorithm.

For a given randomised algorithm A that produces the correlated sequence $(x_1, \ldots, x_n, \ldots)$ one by one, we can define a G that first samples for an $n \in \mathbb{N}$ (e.g., using Example 2.1), then simulates A until the first $n-1$ outputs have been emitted, before writing the final x_n on the output tape. Clearly, x is a possible output of the algorithm A if and only if $x \in \mathcal{L}(G)$, and that G uses the same amount of *space* as A to produce x.

Many prominent examples of practical generators naturally fit this framework of sequence-producing generators. These include coverage-guided fuzzing [5] and concolic execution [8, 15]—which we will discuss in more detail later on in Section 6.2. In these cases, the time taken by the generator is of orthogonal relevance, as many of these methods are designed to run *indefinitely*—for instance, as a background process to continuously find bugs [5, 7]. Thus, for these generation algorithms, space bounds can be a more natural theoretical constraint to study.

There is a caveat with using space-bounded generators verbatim to model these algorithms: many such algorithms call the SUT, which may be arbitrary computation. Thus, it would make sense to *separate* the generation algorithm from the SUT call: the generation process should be independent of the SUT it is applied to. This issue will be addressed later in Section 6.2.

We start by defining the specific space-bounded generation classes considered in this section. The first is on logspace generators, which have access to pointers and counters up to the size of the generated output. Logspace computations still make sense, since generators have read-once-only input and write-once-only output, preventing them from using the input/output tapes for storing more information than their space bound permits. Though seemingly restrictive, we will see later that logspace still suffices for generating instances of useful predicates, like the set of satisfiable 2-SAT formulae or directed graphs with guaranteed connectivity between two chosen nodes.

Definition 6.1. *The class of log-space generable languages are denoted as LG.*

$$LG = \bigcup_{c \in \mathbb{N}} GSPACE(c \log(n))$$

We also consider languages generable within polynomial space. This class is permissive and covers the majority of the state-dependent generation algorithms seen in practice: storing more than a polynomial amount of data in the size of the test input is impractical in most applications.

Definition 6.2. *The class of languages generable within polynomial space is defined as:*

$$PSPACEG = \bigcup_{i \in \mathbb{N}} GSPACE(n^i)$$

We begin with some general facts on space-bounded generators, connecting them to existing models of time-bounded generators and decision procedures.

The first proposition relates space-bounded decision procedures and time-bounded generators, where a time-bounded generator with *much more* time can generate samples from a space-bounded NDTM. This is in analogy to a standard result on space-time tradeoffs between NDTMs and DTMs for decision procedures.

Proposition 6.1. *For space-constructible $s : \mathbb{N} \to \mathbb{N}$,*

$$NSPACE(s(n)) \subseteq GTIME(2^{O(s(n))})$$

Proof. Deferred to Appendix A.13. Intuitively, the extra time on the generator allows for the nondeterministic machine to be simulated by enumerating configurations. Then, by using a complete decision language within $NSPACE(s(n))$ (itself decidable on the generator as a procedure) that

decides whether an accepting suffix exists for M, it is possible to generate, for any $L \in NSPACE(s(n))$, any $x \in L$ via repeated queries to this complete language. □

Proposition 6.1 together with Proposition 4.1 relates space and (much higher) time bounds for generators, analogous to the classic time-space tradeoff theorem for deciders.

Corollary 6.1 (Time-Space Tradeoff). *For a space constructible function $s : \mathbb{N} \to \mathbb{N}$ with $s(n) \geq \log(n)$,*

$$GSPACE(s(n)) \subseteq GTIME(2^{O(s(n))})$$

For the decision version of Corollary 6.1, a configuration-counting argument is the standard proof. This argument does not carry over immediately, since space-bounded generators can be driven into cycles for an unbounded amount of time by their *input*. In Appendix A.11, we discuss the issue with the configuration counting argument in more detail and give a fix in an alternative, direct proof of Corollary 6.1.

6.1 Logspace Generators

Proposition 4.1 and Proposition 6.1 together mean that every *NL* language is generable in polynomial time (i.e., has a *PG* algorithm).

Corollary 6.2. *$LG \subseteq NL \subseteq PG$.*

This implies properties with preconditions such as 2-SAT, or connected directed graphs, will also have efficient polynomial-time generators. This also gives a practical sufficient condition for *PG* membership: if the predicate can be decided by an *NL* machine, then an efficient polynomial-time generation algorithm is guaranteed to exist.

We can apply a known result [2] to show that there exists an algorithm in *EPG* uniformly generating any $L \in LG$. The main idea is to view logspace generators as logspace transducers (adapted to our setting), which Arenas et al. [2] proved have uniform samplers with a bounded probability of failure. It is straightforward to adapt such uniform samplers to a generator in *EPG*.

Proposition 6.2. *If $L \in LG$, then there exists a EPG generator of uniform distributions.*

Proof. We defer the proof to Appendix A.12. □

Consequences for PBT: *When a precondition is decidable in NL (e.g. reachability, 2-SAT, and most syntactic well-formedness checks) an efficient generator is* guaranteed *to exist, and (by Proposition 6.2) even a uniform one.*

We also see that even in the restricted setting of *LG*, we are still able to generate *NL*-complete languages, analogous to how *PG* contains *NP*-complete problems.

Definition 6.3. *Reachability on directed graphs is defined as:*

$$REACH = \{\langle G, s, t\rangle : \exists \text{ path from } s \text{ to } t \text{ in } G\}$$

Proposition 6.3. *$REACH \in LG$.*

Proof. The construction is based on certificate sampling and instance reconstruction, which is generalised by Theorem 7.1 later on. We give a sketch here and defer the details to Appendix A.14. We represent graphs as an edge-list, and partition *REACH* by the total number of vertices and edges. First, select the source and target as s and t, then generate a random walk from s to t, outputting edges as they are used. Then, add edges to the graph in Erdos-Renyi fashion, again outputting them directly after selection. Every $x \in REACH$ has a nonzero probability of being sampled. □

6.2 Generators With Feedback

Testing techniques like concolic execution and coverage-guided fuzzing produce a (potentially infinitely long) sequence of test inputs, where the results from executing previous inputs are used to guide the generation of the next using the results from running the SUT. Since the SUT can perform arbitrary computation, we separate the SUT from the *generation technique* in our model.

Definition 6.4 (Function Oracle Turing Machine). *A TM M_f with oracle access to $f : \Sigma^* \to \Sigma^*$ is a standard TM with an additional read-write query tape t_q, a read-only answer tape t_a, and a designated state s_o. When M_f enters state s_o with x written on the query tape t_q, the contents on the answer tape t_a will be replaced by $f(x)$. The language recognised/generated by M_f is denoted $\mathcal{L}(M_f)$.*

For a Turing machine M, we denote M_f as the same Turing machine with access to an oracle for f. When there is no ambiguity, we may drop the subscript f from M_f for brevity. If M is a generator, we will denote it with G instead, i.e. G_f is a generator G with oracle access to f.

This oracle machine is different from the ones discussed in Proposition 5.3: here oracle access is to a function f whereas the previous one gives oracle access to a decider $f : \Sigma^* \to \{T, F\}$.

The execution of the SUT, together with the computation of data guiding future generation, can be viewed as an oracle available to the generator. We present simplified versions of two prevalent testing methodologies and show how they fit into our framework, noting that real implementations often employ variants and optimisations to reduce time and space usage. Nonetheless, our eventual result still holds: any space-bounded generation process, no matter how it is guided, can only generate languages within some complexity class, thus ruling out all languages outside of the class.

Example 6.1 (Concolic Execution). *Concolic execution for an SUT S is an iterative process that first generates an input x_0, then tracks the control flow path taken by the run $S(x_0)$, collecting path constraints p_0 along the way. Then, new inputs x_i are generated to ensure that they reach control-flow paths unexplored by previous inputs $x_0, \ldots, x_{i-1}$, usually through an SMT solver to produce x_i that satisfies an alternative path constraint, which forces unexplored paths to be taken.*

Here, the SUT oracle $f : \Sigma^* \to \Sigma^*$ accepts a generated input x_i, executes the SUT on x_i, and returns the path information $p_i = f(x_i)$ observed during that execution. The generator G has oracle access to f. After generating each output x_i, G calls $f(x_i)$ and stores the paths taken within the codebase as a list $p_0 = f(x_0), \ldots, p_i = f(x_i)$. To produce the element x_{i+1}, G invokes a solver to find an input x_{i+1} that satisfies the path condition p_{i+1}, distinct from all previous paths $p_0, \ldots, p_i$.

Example 6.2 (coverage-guided fuzzing). *coverage-guided fuzzing produces new inputs by mutating previous inputs that are likely to gain new coverage. Historical input x is selectively stored as y_k together with the code coverage p_k obtained by y_k. To generate x_{i+1} that may achieve new coverage, inputs y that were close to achieving new coverage are selected for mutation. If new coverage has been found by x_{i+1}, then it is stored along with the previous inputs that expanded coverage: $y_1, \ldots, y_k$.*

Here, the SUT oracle $f : \Sigma^* \to \Sigma^*$ accepts inputs x_i and outputs the coverage data p_k for the codebase achieved by x_i. The generator has oracle access to f, storing historical inputs $y_1, \ldots, y_k$. Using these, it produces new inputs by mutating existing inputs for new coverage.

Since the oracle contains computation of the SUT, they can realistically be of any complexity (e.g. compilers), and is independent of the generation complexity. Therefore, we do not constrain resource usage of the oracle. However, as the output still needs to be parsed and stored by the generator, the *size* of the output from the oracle call can affect the complexity of generation. For an oracle, we therefore define the following constraint:

Definition 6.5. *A function $f : \Sigma^* \to \Sigma^*$ is length-bounded by $s : \mathbb{N} \to \mathbb{N}$ if for all $x \in \Sigma^*$, $|f(x)| \leq s(|x|)$.*

Complexity classes can be defined analogously for Turing machines with oracle access. Here, the oracle input tape is under the same space bounds as work tapes.

Definition 6.6. *For space constructible function $s : \mathbb{N} \to \mathbb{N}$, and a function $f : \Sigma^* \to \Sigma^*$, define the classes of languages decidable/generable within s space as follows:*

$$DSPACE(s(n))^f = \{\mathcal{L}(M_f) : M \text{ is space-bounded by } s\}$$
$$GSPACE(s(n))^f = \{\mathcal{L}(G_f) : G \text{ is space-bounded by } s\}$$

Note that the space-bound s applies both to the work tape and the query tape.

Definition 6.7. *For a function $f : \Sigma^* \to \Sigma^*$,*

$$PSPACE^f = \bigcup_{c \in \mathbb{N}} DSPACE(n^c)^f \qquad PSPACEG^f = \bigcup_{c \in \mathbb{N}} GSPACE(n^c)^f$$

Both techniques described in Example 6.1 and Example 6.2 can use infinite space in the limit. However, real-world applications are bounded by resource limits, and using polynomial space as a proxy for realism, we ask: what constraints placed on coverage-guided fuzzing and concolic execution ensure that the procedures remain in *PSPACEG*?

Coverage-Guided Fuzzing. Since the codebase is fixed, the coverage information has constant size regardless of the input sizes, thus the output of the oracle is always within a polynomial bound (in fact, constant size). Each new coverage-finding input is stored alongside its coverage information, which consumes space on the work tape. Within a polynomial space bound p, generators of outputs of sizes up to n can store $p(n)/n$ previous inputs (if stored naively).

Concolic Execution. The data returned by the oracle consist of the path taken by the generated input x_i of size (up to) n. For the oracle to stay within polynomial space bounds, the path taken must therefore have at most polynomially many control flow statements (which is achievable with instrumentation and timeout, for instance). As the previous paths taken are stored by the generator, there can be at most a polynomial number of historical paths stored.

The above examples are illustrative for demonstrating the kind of constraints that polynomial space induces on naive implementations of popular generation algorithms. In reality, optimisations would be used on implementations for either technique [3, 38]: the calculus shifts accordingly for the number of paths or test cases one is able to store, but the bottleneck remains in the polynomial space bounds. We show below the limits of generation capability for such feedback-driven approaches when a polynomial space bound is adhered to.

The previous complexity results for space-bounded generators still hold with oracle access.

Proposition 6.4. *Suppose $s : \mathbb{N} \to \mathbb{N}$ is both space and time constructible with $s(n) \geq \log(n)$, and $f : \Sigma^* \to \Sigma^*$ is length-bounded by $2^{O(n)}$. Then:*

$$GSPACE(s(n))^f \subseteq NSPACE(s(n))^f \subseteq GTIME(2^{O(s(n))})^f$$

Proof. Deferred to Appendix A.15. Similar to the non-oracle version of Proposition 6.1. Extra care is needed when counting the configurations of oracle machines: the read-only answer tape and f being a function are the two factors that keep the number of configurations at $2^{O(s(n))}$. □

Proposition 6.5. *For $f : \Sigma^* \to \Sigma^*$ length-bounded by $2^{O(n)}$, $PSPACEG^f = PSPACE^f$*

Proof. Deferred to Appendix A.17. $PSPACEG^f \subseteq PSPACE^f$ follows from a variant of Savitch's theorem adapted for our oracle machines. $PSPACE^f \subseteq PSPACEG^f$ follows since rejection sampling can be performed whilst reusing space. □

If the oracle f itself is computable in polynomial space, then both $PSPACE^f$ and $PSPACEG^f$ collapse to *PSPACE*. We thus have the following as a corollary:

Theorem 6.1. *PSPACEG = PSPACE.*

If the SUT itself runs in polynomial space, then the polynomial-space generable properties for the SUT are exactly the *PSPACE* decidable ones. If the SUT does not run in polynomial space, then the polynomial-space generable properties are exactly those with input sets decidable by a *PSPACE* machine with oracle access to the SUT.

Consequences for PBT: *Theorem 6.1 implies feedback-driven techniques—coverage-guided fuzzing, concolic execution—are bounded by PSPACE. For example, unless the SUT itself is powerful enough to solve EXPTIME-complete problems, EXPTIME-complete sets remain out of reach regardless of how the feedback loop is engineered. Corollary 6.1 says that any input set a feedback-driven technique can sample in polynomial space is also reachable by a (slower) generator of independent samples.*

Example 6.3. *If PSPACE ≠ EXP, then the following problems are not generable in polynomial space.*

(1) *Board positions of generalised size-n Chess/Shogi/Draughts games with a winning strategy for the player that moves first.*
(2) *For any programming language, the set of programs that terminate in k steps, with the step count specified in binary.*

7 Designing Generators and Generator Libraries

In this section, we introduce some practical insights that come from our theoretical framework. We first provide canonical constructions for building efficient generators (that are members of *PG*), and then apply the results we established in generation complexity to gain understanding of the limitations of compositional library designs for randomised testing.

7.1 Conditions for *PG* Membership

We propose two characterisations of *PG* languages, which formalise the main known techniques for obtaining *PG* generators.

7.1.1 PG Via Basis Extension. The first technique is based on the ability to generate *default* elements of the language. A difficulty with generating inputs accepted by *NP* deciders is that the majority of the branches may fail. Thus, a generator would need to find a branch of the *NP* computation that succeeds in order to confirm a string's membership—which by naive methods leaves an exponential number of branches needing to be checked. However, if one has the ability to efficiently find a default element as a fallback, a generator is free to try potentially-failing computations, allowing it to generate the remainder of the *NP* language.

Definition 7.1. *For the language L, define the functions that compute a default element of L as:*

$$DEFAULT(L) = \{f : \{1\}^* \to L : L_i \neq \emptyset \implies f(1^i) \in L_i, L_i = \emptyset \implies f(1^i) = \epsilon\}$$

Proposition 7.1. *For $L \in NP$, $DEFAULT(L) \cap FP \neq \emptyset$ if and only if $L \in PG$, where FP denotes polynomial-time computable functions.*

Proof. The main idea is that being able to compute a default element x means we can simulate a branch of the NDTM, falling back to x if the branch rejects. For a full proof, see Appendix A.16. □

We can extend the structure of such techniques to a sufficient condition for generating samples from an *NP* language, by asserting extra structural conditions on the language for efficient extension from a base element.

Proposition 7.2. *Let L be a language and $\preccurlyeq$ be a polynomial-time computable partial order on Σ^*. Let $K = \{s \in L : \nexists s' \in L \setminus \{s\}, s' \preccurlyeq s\}$ be the set of minimal elements in L according to $\preccurlyeq$. $L \in PG$ if the following conditions are met:*

(1) ***Upward Closure**: For any $s \in L$ and string s', if $s \preccurlyeq s'$, then $s' \in L$.*
(2) ***Samplable Minimal Basis**: K contains every minimal element of L, and is in PG.*
(3) ***Efficient Extension**: For every element $s \in K$, there exists a polynomial-time generator G_s such that $\mathcal{L}(G_s) = \{s' : s \preccurlyeq s' \wedge |s'| = |s|\}$.*

In other words, if there exists a core set of instances in L that is efficiently generable, and one can efficiently extend from the core instances to any instances in L, then the set is efficiently generable.

Proof. Deferred to Appendix A.18. □

This provides a technique for obtaining *PG* sampling algorithms, and indeed many *NP*-hard languages satisfy this property. We give examples applications of Proposition 7.2 in Appendix A.19.

7.1.2 PG Via Certificate Sampling. Many *NP* decision problems are hard because of the need to find a potentially small set of certificates for the instances. However, we have seen that many hard *NP* problems nonetheless have efficient generators. The problem of generating satisfying instances of an *NP* language becomes easier than decision when there is a means of constructing instances from certificates: ensuring that the generated instance is in the language by construction.

This is the core insight used by many existing generators of *NP*-complete languages, which is based on their verifiers: first sample a random certificate, and then sample a random instance that is verified by the certificate. We generalise this and show that the reverse also holds. That is, every generator for L is associated with some canonical verifier V for L, a method to sample certificates of V, and a method to recover random instances from certificates. Thus, every generator can be viewed as an efficient algorithm to recover instances for each canonical certificate (i.e., the certificate consisting of bitstrings).

Definition 7.2 (Polynomial-Time Verifier). *L has a polynomial-time verifier V if:*

- *V takes inputs $w\#c$, where $w \in \Sigma^*$ and $c \in \Sigma^+$ is a certificate.*
- *V terminates in polynomial time of its input size.*
- *There exists a polynomial p where:*
  - *If $w \in L$, then there exists a certificate c with $|c| \leq p(|w|)$ such that V accepts $w\#c$.*
  - *If $w \notin L$, then for all certificates c, V rejects $w\#c$.*

Definition 7.3 (Certified Inputs). *For a verifier V, and a certificate c, define the certified inputs of c, $V_f(c)$ as:*

$$V_f(c) = \{w \in \Sigma^+ : V(w\#c) \text{ accepts}\}$$

Definition 7.4 (Certificate Generator). *Verifier V for L has a certificate generator S_V if there exists a polynomial q, and $S_V = (S_n)_{n\in\mathbb{N}}$ is a uniform family of generators where:*

(1) *Each S_i is such that $L_i = \Sigma^i \cap \bigcup_{c\in\mathcal{L}(S_i)} V_f(c)$.*
(2) *Each S_i runs in time $q(i)$.*
(3) *If L_i is empty, then S_i rejects.*

Note that in the first condition, we constrain $\bigcup_{c\in\mathcal{L}(S_i)} V_f(c)$ to the subset of length-i strings. This is necessary as a certificate may verify instances of many lengths. As a concrete example, consider a verifier of SAT, taking certificates to be the variable assignment. A specific assignment can be the certificate for formulas of arbitrary length.

Definition 7.5 (Certificate Recovery). *Verifier V for L has a certificate recoverer R_V if there exists a polynomial p in two variables, and $R_V = (R_i)_{i\in\mathbb{N}}$ is a uniform family of TMs where:*

(1) *Each R_i has a read-only input tape for certificates $c \in \Sigma^+$, a read-only input tape of random bits, and an output tape.*
(2) *For all c, $R_i(c)$ is a generator for $L_i \cap V_f(c)$.*
(3) *For all c, $R_i(c)$ runs in time $p(i, |c|)$.*

Definition 7.6 (Certificate Scheme). *V has a certificate scheme (S_V, R_V) if there exist a certificate generator S_V and also a certificate recoverer R_V.*

Theorem 7.1 (Generators are Certificate Schemes). *The following are equivalent:*

(1) *L is generable in polynomial time ($L \in PG$).*
(2) *There exists a polynomial-time verifier V for L, and V has a certificate scheme.*

Proof. We defer the full proof to Appendix A.20. Intuitively, a certificate scheme yields a polynomial time generator by composing the components together. Conversely, given a *PG* generator G, one can derive a verifier V_G by treating the certificate as the input bitstring of G. Then, one can validate that there is a certificate scheme for G based on the verifier V_G, by sampling for a random bitstring certificate of length $p(n)$, where p is the polynomial time-bound of G. □

Consequences for PBT: *A strategy for building an efficient generator for a language is to study its* verifier *rather than its decider, by sampling a certificate first, then building a random instance around it. Theorem 7.1 shows that* every *efficient generator actually uses some instance of this strategy.*

7.2 Compositional Libraries For Generators

As established earlier, while rejection sampling gives an easy way to automatically derive generators that satisfy a specific predicate, it can quickly become intractable as the probabilities of data satisfying the predicate drop. Moreover, if the predicate is difficult to verify, then the validation step during test-case generation can itself become the dominant factor for a testing campaign.

Due to this, property-based testing libraries often also provide combinators for users to specify custom generators, rather than relying on rejection sampling to produce inputs. A desirable characteristic for such libraries is *compositionality*. In the context of testing, one clear place for compositionality involves the duality of generators and predicates: predicates and generators both correspond to formal languages on the inputs. Where predicates interpret a language L as a function of type $\Sigma^* \to \mathbb{B}$, generators interpret L as a surjective computable function $\{0,1\}^* \to L$ that constructs instances of the language from randomness.

Logical predicates are easily specified using existing languages, such as propositional logic, described by the following grammar, where *Atomic* refer to base predicates taken as fact (e.g. $\leq, \equiv$):

$$T := Atomic \mid T \wedge T \mid T \vee T \mid T \to T \mid \neg T$$

Compositionality in this context is the requirement for a function f to map the logical language into generators homomorphically, such that f follows the same structure as the logical language:

$$f(a : Atomic) = f_a(a) \qquad f(t_1 \to t_2) = gImplies(f(t_1), f(t_2)) \qquad f(\neg t) = gNot(f(t))$$
$$f(t_1 \wedge t_2) = gAnd(f(t_1), f(t_2)) \qquad f(t_1 \vee t_2) = gOr(f(t_1), f(t_2))$$

where f_a maps the base predicate a to a generator of values that satisfies the predicate a, and *gAnd*, *gOr*, *gImplies*, *gNot* are the functions that implement the generator homomorphisms for logical operators $\wedge, \vee, \to, \neg$ respectively.

We now show that, based on intuitions imported from complexity theory via our formalisation, such a compositional design for efficient generators is not possible in general—if the logical language for predicates can express predicates equivalent in power to the operations $\neg$ or $\wedge$.

To put it another way: in a compositional library of generators/predicates, either the generators cannot have polynomial runtime guarantees, or the logical operations must be limited in expressivity—it cannot contain anything equivalent to $\neg$ or $\wedge$ (that is, *gAnd* and *gNot* cannot have implementations for generators).

Firstly, we show that some operators can be implemented composably for efficient generators.

Proposition 7.3. *PG is closed under concatenation, Kleene star, and union.*

Proof. We defer the proof to Appendix A.21. □

On the other hand, we see that some logical operators on predicates cannot have composable implementations on generators (under standard complexity constraints).

Theorem 7.2 (Efficient Generators are not Composable). *If NP ≠ coNP, then PG is not closed under complementation. If collision-resistant hash functions exist, then PG is not closed under intersection.*

Proof. Complementation follows from Corollary 5.1 and Proposition 5.1. If CNF-UNSAT = CNF-SAT^c $\in$ *PG*, then CNF-UNSAT $\in$ *NP* by Corollary 5.1. Thus *NP* = *coNP* and we have a contradiction. See Appendix A.22 for a proof of the intersection case. □

Consequences for PBT: *No PBT library can offer a general gAnd or gNot combinator with polynomial-time efficiency guarantees. Theorem 7.2 means library designers must choose: restrict the predicate language (e.g. to NL/linear Datalog, where compositional compilation is possible), or accept rejection sampling—which our results show is then close to optimal.*

Remark 3 (What is still possible?). Theorem 7.2 does not rule out compositionality on restricted predicates, or on alternative logical languages that do not entail intersection and negation operators. For instance, we know that *NL* $\subseteq$ *PG*, and *NL* is closed under the logical operations discussed in this section (complementation, intersection, union). Thus, it would be feasible in theory to design a composable generator library, restricted to *NL* predicates $Pred : L \to \mathbb{B}$ that produces polynomial-time composable generators for *NL* languages. Descriptive complexity gives a path to achieving this: *NL* Turing machines have an exact correspondence with linear Datalog programs. Thus, it would be possible in theory to write a compiler that composes linear Datalog predicates together, and compiles them into polynomial-time generators of their accepted languages.

8 Related Work

The closest work to ours is [45], which introduced complexity classes for polynomial-time generators and observed several similar results to ours: polynomial-generable languages being in *NP*, some *NP*-complete problems are polynomial-time generable, and placing oracle-based generation within the polynomial hierarchy. They also show a conditional separation for *PG* $\subsetneq$ *NP*, assuming a tally language (one with at most one string per length n) exists in $NP \setminus P$—or equivalently, $E \neq NE$. Tally language generators, however, have no realistic interpretation in testing: they are decision algorithms in disguise that cannot produce different random instances. In Section 5.1 we show that non-sparse languages can also be non-generable, under a cryptographic assumption not known to be related to the sparse languages assumption. This both elucidates generability for the testing context and offer independent complexity-theoretic intuition linking generability and cryptography. Moreover, [44, 45] do not consider other resource constraints (unconstrained, expected polynomial time, log-space and polynomial-space bounds), generability conditions via core languages and verifiers, or connections to cryptography—unsurprisingly, perhaps, as their work predates PBT, a

common application of randomised testing. Our work can thus be viewed as both an extension and a modern reinterpretation of their earlier work, with practical applications to randomised testing.

A classic line of work [24, 25] studies *uniform* generation and repetition-free enumeration in a different setting: given a relation $R \subseteq X \times Y$ and an $x \in X$, one seeks a sampler for $Sol(x) = \{y \in Y : (x, y) \in R\}$. Jerrum et al. [25] showed that for self-reducible R, almost uniform generation of $Sol(x)$ is as hard as approximately counting $|Sol(x)|$; more recently, Arenas et al. [2] showed that if R is checkable in logspace, then $Sol(x)$ is almost-uniformly sampleable, approximately countable, and enumerable with polynomial delay. Their setting differs from ours in two ways: they sample certificates for instances $x \in L$, whereas we sample instances of L itself; and they require uniformity while permitting failure, whereas we require only full support on L but forbid failure. The two lines of work nonetheless intersect in Proposition 6.2, where interpreting logspace generators in the framework of Arenas et al. [2] yields the expected polynomial-time uniform generability for LG languages. Our framework is currently distribution agnostic, and extending it to prescribed distributions on L, perhaps along the lines of Jerrum et al. [25], is an avenue for future work.

Levin's theory of average-case complexity [30] studies the hardness of problems whose instances are drawn from *P-samplable* distributions [22]. A polynomial-time generator $G = (G_n)_{n \in \mathbb{N}}$ for L implements a full-support P-samplable distribution for L, by composing the G_n with a distribution supported on $\mathbb{N}$. The focus differs, however: Levin [30] studies the hardness of deciding L given *some* distribution μ (not necessarily full-support), whereas we ask which languages *admit* a full-support sampler at all, and what structure it must have (Section 7.1). Average-case hardness of language-distribution pairs also underpins cryptographic security, e.g. in constructing one-way functions. Generating instances around planted witnesses in Proposition 5.1 for SAT (first observed in [44]) is an instance of the classic planted-assignment construction [13, 26]. In Theorem 7.1, we show that every polynomial-time generator is a certificate scheme: witness planting is not merely a technique for efficient generation, but in some sense the only one.

The duality between predicates and generators has previously been investigated in a programming languages and verification context. In [9, 28], generators are derived from implementations of deciders by techniques based on *narrowing* from logic programming folklore. In our notation, for predicate P, the derived generator G_P generates the same language that P recognises: $\mathcal{L}(G_P) = \mathcal{L}(P)$. However, the unconstrained use of rejection sampling in these derivations means the correspondence carries no complexity guarantees, and Section 7.2 limits the complexity of generators from such constructions: no compositional mapping from a logical language containing conjunction or negation to generators can preserve polynomial-time generation guarantees, under standard assumptions. Libraries have also been developed to facilitate proofs on generators within theorem provers [37]. However, these proofs are on a per-generator basis, primarily on soundness (corresponding to our notion of full support on L), and do not make statements on the complexity of generation—which is our focus.

Although software testing was the motivation for our work, random generators of the kind we study are used throughout computational sciences. Many fields rely on non-trivial generators of random data, often as the core part of a larger algorithm or methodology. *Markov Chain Monte Carlo* (MCMC) is a widely used technique to produce random samples from sets whose membership has a high decision complexity [41]. Designing a Markov chain that converges to the desired distribution on some set can be viewed as a generation problem in our formalisation. Our theory of space-bounded generation for correlated outputs (Section 6) yields statements on the feasibility of designing chains for a given set. Chemical space generation is another field where combinatorial databases for molecules are built, with applications to drug discovery and reaction pathways. For the database to be useful, its elements must satisfy graph-theoretic constraints of molecules [19, 42], feasibility constraints from physical simulations [21, 51], and synthesisability constraints imposed

by the domain of application [29]. The combinatorial explosion makes throughput critical for these databases, which are often accessed through sampling rather than enumeration. Our theory provides results connecting the complexity of the constraints (defining the set of molecules) with both the feasibility and efficiency of building such a chemical space (Section 3, Section 5 and Section 6). Many more examples exist. In each case, the generator is the core component of a larger pipeline producing randomised outputs, drawn from a set with nontrivial membership constraints: exactly the central object of study in this work.

9 Conclusion

We formalised generability as a complexity-theoretic notion by modelling generators as resource-bounded transducers, and studied how constraints shape what data can be generated. Without resource bounds, we see that generable languages coincide exactly with the recursively enumerable languages, while under polynomial-time bounds we obtain a structured hierarchy of generability classes contained within *NP*, with logspace being a more restrictive condition than polynomial-time for generators. Moreover, we find that space-based constraints can be applied naturally to generators of *correlated* inputs, and derive the limits of their expressivity. We also give characterisations of the polynomial-time generable languages, showing they correspond exactly to a subclass of verifiers. Applying our theory to testing libraries, our framework can rule out whole classes of designs from the realm of possibility. Altogether, our results provide a foundational framework for reasoning about generators in property-based testing using tools from complexity theory. More broadly, our approach also opens avenues for transferring complexity-theoretic techniques to the study of testing, while exposing potential new complexity questions centered on generation.

References

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. 2004. PRIMES is in P. *Annals of mathematics* (2004), 781–793.

[2] Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. 2019. Efficient logspace classes for enumeration, counting, and uniform generation. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 59–73.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th international conference on software engineering*. 1083–1094.

[4] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2020), 79–86.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[6] Alessandro Disney Bruni, Tim Disney, and Cormac Flanagan. 2011. A peer architecture for lightweight symbolic execution. *Universidad de California, Santa Cruz* (2011).

[7] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 63–74.

[8] Cristian Cadar and Martin Nowack. 2021. KLEE symbolic execution engine in 2019: C. Cadar, M. Nowack. *International Journal on Software Tools for Technology Transfer* 23, 6 (2021), 867–870.

[9] Koen Claessen, Jonas Duregård, and Michał H Pałka. 2014. Generating constrained random data with uniform distribution. In *International Symposium on Functional and Logic Programming*. Springer, 18–34.

[10] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

[11] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. *ACM SIGPLAN Notices* 47, 1 (2012), 533–544.

[12] Karine Even-Mendoza, Cristian Cadar, and Alastair F Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering* 27, 6 (2022), 129.

[13] Vitaly Feldman, Will Perkins, and Santosh Vempala. 2015. On the complexity of random satisfiability problems with planted solutions. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*. 77–86.

[14] Daniel Gabric and Jeffrey Shallit. 2021. Borders, palindrome prefixes, and square prefixes. *Inform. Process. Lett.* 165 (2021), 106027.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[16] Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C Pierce. 2026. The Search for Constrained Random Generators. *Proceedings of the ACM on Programming Languages* 10, PLDI (2026), 2060–2084.

[17] Harrison Goldstein and Benjamin C Pierce. 2022. Parsing randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 89–113.

[18] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 78–88.

[19] Ralf Gugisch, Adalbert Kerber, Axel Kohnert, Reinhard Laue, Markus Meringer, Christoph Rücker, and Alfred Wassermann. 2015. MOLGEN 5.0, a molecular structure generator. In *Advances in Mathematical Chemistry and Applications: Volume 1 Revised Edition*. Bentham Science Publishers, 113–138.

[20] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 45–48.

[21] Scott A Hollingsworth and Ron O Dror. 2018. Molecular dynamics simulation for all. *Neuron* 99, 6 (2018), 1129–1143.

[22] Russell Impagliazzo and Leonid A Levin. 1990. No better ways to generate hard NP instances than picking uniformly at random. In *FOCS*, Vol. 90. 31st.

[23] ISO. 1998. *ISO/IEC 14882:1998: Programming languages — C++*. 732 pages. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.. http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%2D1998;http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%3A1998;http://www.iso.ch/cate/d25845.html;https://webstore.ansi.org/

[24] Mark Jerrum. 2003. *Counting, sampling and integrating: algorithms and complexity*. Springer Science & Business Media.

[25] Mark R Jerrum, Leslie G Valiant, and Vijay V Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical computer science* 43 (1986), 169–188.

[26] Ari Juels and Marcus Peinado. 2000. Hiding cliques for cryptographic security. *Designs, Codes and Cryptography* 20, 3 (2000), 269–280.

[27] Jonathan Katz and Yehuda Lindell. 2007. *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC.

[28] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[29] Itai Levin, Michael E Fortunato, Kian L Tan, and Connor W Coley. 2023. Computer-aided evaluation and exploration of chemical spaces constrained by reaction pathways. *AIChE journal* 69, 12 (2023), e18234.

[30] Leonid A Levin. 1986. Average case complete problems. *SIAM J. Comput.* 15, 1 (1986), 285–286.

[31] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. doi:10.1145/3428264

[32] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (jun 2023), 22 pages. doi:10.1145/3591295

[33] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[34] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[35] Ali Mili and Fairouz Tchier. 2015. *Software testing: Concepts and operations*. John Wiley & Sons.

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[37] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C Pierce. 2015. Foundational property-based testing. In *International Conference on Interactive Theorem Proving*. Springer, 325–343.

[38] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. 861–875.

[39] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[40] Phillip Rogaway. 2006. Formalizing human ignorance: Collision-resistant hashing without the keys. In *International Conference on Cryptology in Vietnam*. Springer, 211–228.

[41] Vivekananda Roy. 2020. Convergence diagnostics for markov chain monte carlo. *Annual Review of Statistics and Its Application* 7, 1 (2020), 387–412.
[42] Lars Ruddigkeit, Ruud Van Deursen, Lorenz C Blum, and Jean-Louis Reymond. 2012. Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17. *Journal of chemical information and modeling* 52, 11 (2012), 2864–2875.
[43] Alexander Russell. 1995. Necessary and sufficient conditions for collision-free hashing. *Journal of Cryptology* 8, 2 (1995), 87–99.
[44] Laura A. Sanchis. 1990. On the complexity of test case generation for NP-hard problems. *Inform. Process. Lett.* 36, 3 (1990), 135–140.
[45] Laura A Sanchis and Mark A Fulk. 1990. On the efficient generation of language instances. *SIAM J. Comput.* 19, 2 (1990), 281–296.
[46] Phillip Schanely. 2022. CrossHair: An analysis tool for Python that blurs the line between testing and type systems. https://github.com/pschanely/CrossHair. GitHub repository, accessed 19 January 2026.
[47] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.
[48] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 583–594.
[49] Terence Tao, Ernest Croot III, and Harald Helfgott. 2012. Deterministic methods to find primes. *Math. Comp.* 81, 278 (2012), 1233–1246.
[50] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532
[51] Pingshi Yu, Alistair J Sterling, and Jotun Hein. 2021. A Novel Automated Screening Method for Combinatorially Generated Small Molecules. *Journal of Chemical Information and Modeling* 61, 4 (2021), 1637–1646.
[52] Pingshi Yu, Nicolas Wu, and Alastair F Donaldson. 2025. Ratte: Fuzzing for Miscompilations in Multi-Level Compilers Using Composable Semantics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 966–981.

A.1 Alternative notion of eventual productivity

Another candidate for the notion of "eventually productive" may be the following—which in any case would be a desirable necessary property for generators:

Definition A.1.1 (Extensible Transducer). *A bitstring transducer T is extensible if for any $b \in \{0,1\}^*$ where $T(b)$ rejects, there exists an $e \in \{0,1\}^+$ such that $T(b +\!\!+ e)$ accepts.*

On the surface, this seems to describe the *eventually productive* property, since no computations are ever "dead", due to the extension e being always available for any rejected b.

However, it is possible that the extension e becomes increasingly rare as the computation advances. Consider the following case:

Example A.1.1. *Define the bitstring transducer H to accept and output the binary representation of the string "`nice`", if and only if the input has the form $y +\!\!+ y +\!\!+ z$ for some finite bitstring $y \in \{0,1\}^+$ and suffix $z \in \{0,1\}^*$.*

H clearly satisfies Definition A.1.1. However, when inputs are produced by a uniform-like source of randomness (such as a PRNG), the probability of the input having the form $y +\!\!+ y +\!\!+ z$ becomes increasingly rare as the length of the input increases.

That is, when the input bitstring is sampled from $\mathcal{I}$, Example A.1.1 has a strictly non-zero chance of never terminating.

Proposition A.1.1. *The transudcer H defined in Example A.1.1 has $\mu(A_H^\omega) < 1$.*

Proof. Observe first that since A_G^ω is a union of disjoint sets S_b, we have:

$$\mu(A_G^\omega) = \sum_{b \in A_G} \mu(S_b)$$

We will compute an upper bound for $\mu(A_H^\omega)$. A_H contains bitstrings of length $2k, k \in \mathbb{N}^{>0}$ of the form xx, for some $x \in \{0,1\}^*$, and such that no prefix of xx is of the form $yy, y\ in \{0,1\}^*$.

Below we enumerate all members $b \in A_H$ with length $|b| = 2k$ for $k = 1, 2, 3, 4$, as well as the total probability of their bitstrings being sampled, $\sum_{|b|=2k} \mu(S_b)$:

k	Bitstring	Total Probability
1	0 0 1 1	$\frac{2}{2^2}$
2	01 01 10 10	$\frac{2}{2^4}$
3	010 010 011 011 100 100 101 101	$\frac{4}{2^6}$
4	0110 0110 0111 0111 0100 0100 1011 1011 1000 1000 1001 1001	$\frac{6}{2^8}$

Observe that for any $k \geq 2$, if $b \in A_H$ and $|b| = 2k$ then b cannot have 00 or 11 as a prefix (and thus must have prefix either 01 or 10). For $k \geq 2$, the number of bitstrings of length $2k$ within A_H is at most $2 * 2^{k-2} = 2^{k-1}$. Since there are 2^{2k} length $2k$ strings, we know $\sum_{|b|=2k} \mu(S_b) \leq \frac{2^{k-1}}{2^{2k}}$.

It follows that:

$$\mu(A_H^\omega) = \sum_{b \in A_H} \mu(S_b) = \sum_{k=1}^{\infty} \sum_{|b|=2k} \mu(S_b) \leq \frac{2}{2^2} + \sum_{k=2}^{\infty} \frac{2^{k-1}}{2^{2k}} = \frac{1}{2} + \sum_{k=2}^{\infty} 2^{-(k+1)} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4}$$

Hence, the probability of B terminating is at most $\frac{3}{4} < 1$.

In fact, a more exact result exists in literature. Let the number of "square" bitstrings without a square prefix of length $2k$ be the sequence a_k. In our case, we have $a_k = 2^{2k} \sum_{|b|=2k} \mu(S_b)$. In [14], the sum is computed as $\sum_{k=1}^{\infty} c_k = \sum_{k=1}^{\infty} a_k 2^{-2k} \approx 0.72996$, compared to our bound of 0.75. □

It is still possible to have *practically* non-productive computations even if the transducer satisfies Definition A.1.1: it is too weak, and thus the stronger condition of measure-theoretic *eventual productivity* is needed. We will see that the measure-theoretic condition used is indeed stronger than Definition A.1.1:

Proposition A.1.2. *If G is a generator, then G is extensible.*

Proof. Suppose the generator G is not extensible. Then there exists a bitstring b where $G(b)$ rejects and for all strings $e \in \{0,1\}^+$, $G(b +\!\!+ e)$ rejects, and thus $b \notin A_G$. We know $\mu(S_b) > 0$. Therefore $\mu(A_G^\omega) \leq 1 - \mu(S_b) < 1$, which is a contradiction. □

A.2 Proof of Proposition 2.2

We will use the following definitions and lemmas in our proof.

Definition A.2.1. *For a generator G and a bitstring $b \in \{0,1\}^*$, define $s : \{0,1\}^* \to \mathcal{P}(\{0,1\}^*)$, the subset of bitstrings that concatenate with b to reach A_G, as:*

$$s(b) = \{e \in \{0,1\}^* : (b +\!\!+ e) \in A_G\}$$

Definition A.2.2. *For generator G and a bitstring b, define $e : \{0,1\}^* \to \{X\} \cup \{0,1\}^*$, the minimum extension, if one exists, such that $G(b +\!\!+ e(b))$ accepts as follows.*

1. *If $G(b)$ accepts, then $e(b) = \epsilon$. (Empty string)*
2. *If $G(b)$ rejects, then*
   (a) *If $s(b) \neq \emptyset$: $e(b) = \min(s(b))$, where the* min *is taken lexicographically.*
   (b) *If $s(b) = \emptyset$: $e(b) = X$*

Such an f exists, since the set $s(b)$ is either empty, or bounded below (in lengths of elements) by 0, thus a lexicographically minimal element of $s(b)$ exists for all non-empty $s(b)$. We only take the minimum when $s(b)$ is non-empty, and $e(b)$ is defined for all other cases.

Lemma A.2.1. *Let G be a generator. If for all bitstrings $b \in \{0,1\}^*$, $e(b) \neq X$ and there exists $c \in \mathbb{N}$ such that $|e(b)| \leq c$, then G will terminate with probability 1.*

Proof. For event E, let $\mathbb{P}(E|b)$ denote the probability of E occurring, given that b has been consumed on the input tape of G. Clear if $G(b)$ accepts, then $\mathbb{P}(G \text{ accepts in } x \text{ steps}|b) = 1$ for any x. Thus assume $G(b)$ rejects. Note that

$$\mathbb{P}(G \text{ accepts in } |e(b)| \text{ steps}|b) \geq 2^{-|e(b)|}$$

where the inequality is due to the potential for multiple extensions of the same length as $e(b)$. Thus

$$\mathbb{P}(G \text{ does not accept in } |e(b)| \text{ steps}|b) \leq 1 - 2^{-|e(b)|}$$

Suppose G does not terminate on bitstream bs, we can partition bs into the concatenation of a countable set of finite bitstrings, $bs = b_1...$, using f:

$$b_0 = \epsilon$$
$$|b_i| = |e(b_0 b_1 .. b_{i-1})| \qquad i > 1$$

Define $x_i = b_1...b_i$, we can decompose the events that G does not accept bs as follows, and apply the earlier inequality to get:

$$\begin{aligned}
&\mathbb{P}(G \text{ does not accept } bs) \\
=&\prod_{i=1}^{\infty} \mathbb{P}(|b_i| = |e(x_{i-1})| \cap G \text{ does not accept on } b_i | x_{i-1}) \\
\leq&\prod_{i=1}^{\infty} (1 - 2^{-|e(b_i)|})
\end{aligned}$$

We see that if each $|e(b_i)|$ is bounded by some constant c, then the infinite product is guaranteed to collapse to zero. □

Corollary A.2.1. *The generator of natural numbers described in Example 2.1 will terminate with probability 1.*

Proof. Observe that for any sequence $b \in \{0,1\}^+$ that does not terminate for the natural numbers generator, one of the two extensions 000 or 00 will result in the procedure terminating. Thus $|e(b)| \leq 3$ and Lemma A.2.1 applies. □

Corollary A.2.2 (Rejection Sampling is Sound). *Any procedure F that performs rejection sampling based on predicate $f : \{0,1\}^* \to \{T, F\}$ where there exists $w \in \{0,1\}^*$ with $f(w) = T$ will terminate with probability 1.*

Proof. The set of strings that f returns T for is fixed. Let w be the shortest such string such that $f(w) = T$. Thus the shortest extension that causes F to terminate has size bound $|w|$, and Lemma A.2.1 applies. □

When designing generators, to ensure termination, it is enough to argue if there always exists a *short* sequence of choices that leads to termination. If the shortest sequence grows with execution time, then the generator has a chance of non-termination.

A.3 Generator Acceptance and its Link To The Real Numbers

Observation A.3.1. *For a generator G, consider the set of all finite bitstrings x such that $G(x)$ accepts. Let this set be A, then A has the following properties:*

(1) *If $x \in A$, then for all $w \in \{0,1\}^*$, $w +\!\!+ x \in A$.*
(2) *For all $y \in \{0,1\}^*$, there exists a $w \in \{0,1\}^*$ such that $y +\!\!+ w \in A$.*

Consider the map $f : \{0,1\}^* \to [0,1]$, defined as:

$$f(0 :: w) = 2^{-1}(0 + f(w)) \qquad f(1 :: w) = 2^{-1}(1 + f(w))$$

that is, $f(b) = 0.b$, where b here is interpreted as digits in base-2.

Clearly f is an injection. The set A can thus be viewed as regions on the interval $[0,1]$, through the mapping f.

A is defined on finite sequences, however we can extend A to be a set A^+ of infinite sequences by defining:

$$A^+ = \{bs \in \{0,1\}^\omega : \exists b.b \text{ is a finite prefix of } bs \cap b \in A\}$$

We can similarly define f^+ on A^+ by taking the fixed point of the computation f (i.e. the result $f^+(bs)$ would be an infinite sum, and would equal the number $0.b \in [0,1]$). $f^+(A^+)$ has the nice property that the probability measure $\mu(f^+(A^+))$ is precisely the probability that G eventually terminates.

We observe that $f(A)$ (and $f^+(A^+)$) has the following "dense" property: for any subinterval $(a,b) \subseteq [0,1]$, $f(A) \cap (a,b) \neq \emptyset$, and furthermore contains infinitely many elements of $f(A)$.

A.4 Proof of Theorem 3.1

The proof will be given in two parts, for the two sides of the inclusion.

Proposition A.4.1. *$RE \subseteq GEN$*

Proof. Suppose $L \in RE$. We show that there exists a generator G such that $\mathcal{L}(G) = L$. Let M be a Turing machine that accepts x in a finite number of steps iff $x \in L$.

If $|L|$ is finite, then there exists a generator that produces one of the finite elements of $x \in L$ at random. Thus we assume now $|L|$ is infinite. We describe the generator G for L as follows:

(1) Compute some x_0 via dovetailing.
(2) Sample natural numbers n and m (e.g. via the procedure in Example 2.1).
(3) Sample a random string x of length m, and simulate $M(x)$ for up to n steps.
(4) If M accepts during this simulation, then output x, otherwise output x_0.

We now show that G satisfies the conditions of a generator. Because we are working in the case where $L \neq \emptyset$ (in fact $|L|$ is infinite), dovetailing is guaranteed to find some element $x_0 \in L$ in finite time. As the simulation of $M(x)$ is for a bounded number of steps, the sampling procedure will always halt on any finite bitstrings. For almost sure termination on bitstreams, note G is deterministic subject to fixing the results of the two calls to Example 2.1. Thus since Example 2.1 terminates almost surely, so does G.

It clear that G halts in an accepting state if and only if it has generated some $x \in L$, from which it follows that $\mathcal{L}(G) \subseteq L$. For any $x \in L$ there exists n such that M accepts x in n_0 steps. Both x and the corresponding $n > n_0$ has a set of finite bitsrings that results in them being selected in steps 2 and 3. Thus x and n being selected occurs with nonzero probability. We therefore have $\mathcal{L}(G) = L$ as required. □

Proposition A.4.2. *$GEN \subseteq RE$.*

Proof. Suppose $L \in GEN$, and let G be a generator such that $\mathcal{L}(G) = L$.

We construct a Turing machine M that terminates in an accepting state given input $x \in L$, and loops infinitely given input $x \notin L$, as follows.

Given an input x, M enumerates finite-length bitstrings in lexicographic order (0, 1, 00, 01, 10, etc.). For a bitstring b:

- M simulates the execution of G in an initial configuration where the input tape of G contains b. By the definition of a generator (Definition 2.1), this simulation of G is guaranteed to lead to G halting.
- If G halts in an accepting state having generated x (the input to M), M halts in an accepting state.
- If instead G halts in a non-accepting state or in an accepting state having generated some $x' \neq x$, M proceeds to consider the next bitstring in lexicographical order.

By Definition 3.1, for $x \in \mathcal{L}(G)$ there exists a bitstring b such that $G(b) = x$. Therefore, when executed on input x, M will eventually encounter such a b, in which case M will halt, accepting x.

Conversely, for $x \notin \mathcal{L}(G)$ there is no bitstring b such that $G(b) = x$. Therefore, when executed on input x, M will loop forever, simulating the execution of G on increasingly-long bitstrings. □

A.5 Proof of Proposition 4.1

Proof. We construct a non-deterministic TM M for a generator G. Give string x with $|x| = n$, we decide if it belongs to $\mathcal{L}(G)$ on M by simulating G on M, branching nondeterministically whenever a random bit is revealed, with a branch for each of the possible outcomes. When the simulation of G terminates, accept if it is equal to x, and otherwise reject. Thus, x is accepted by M if and only if the simulation of G produced x on any branch, that is, if and only if $x \in \mathcal{L}(G)$. Since G is time bounded by $t(n)$ / space bounded by $s(n)$, each branch of M uses at most $log(n)t(n) + n \in O(t(n))$ time, and/or at most $s(n) \in O(s(n))$ space (assuming space is reused after writing each symbol of x). Thus both claims hold. □

A.6 Proof of Proposition 5.1

Proof. We will define a generator $G_{m,k,p}$ for each (m, k, p), where m is the number of clauses, k is the number of literals, p is the length of the representation for each variable, with $m < k$. Note all CNF-SAT formulas have $m < k$, as if the number of clauses is more than the number of literals ($m > k$), then some clauses must be empty leaving the formula unsatisfiable and therefore not a member of CNF-SAT. Since we fix the number of clauses and literals, we know the exact length of the formula as long as the encoding of the clauses and literals are known. Denote the encoding-dependent length of the formulas by $l(m, k, p)$, which is polynomial-time computable for any reasonable encodings. For instance, $l(m, k, p) = m + (k + 2)p$ is the length of the encoding of the formula as follows: a list of m clauses will contain m length-1 separator characters; $(k + 1)p$ characters will be consumed by p literals of length k each, with an extra character to encode negation; k characters will be consumed by length-1 separators between literals. Thus each string in $\mathcal{L}(G_{m,k,p})$ have the same length, and the language of the generators $L_{m,k,p} = \mathcal{L}(G_{m,k,p})$ forms a partition of the langauge CNF-SAT.

With these, we can define generators for CNF-SAT formulas with length exactly n by first computing the set $I_n = \{(m, k, p) : m + (k + 2)p = n\}$, selecting a random $(m, k, p) \in I_n$ and running the corresponding $G_{m,k,p}$ to obtain a random formula of length n.

It now suffices to show that generators $G_{m,k,p}$ of satisfiable CNF formulas exist for any choice of m, k, p with $m < k$, in time polynomial in $l(m, k, p)$. We define $G_{m,k,p}$ as follows:

(1) Choose an $n \leq k$ to be the number of variables.
(2) For each variable $x_i, 1 \leq i \leq n$, choose a random boolean assignment.
(3) Keep counters m', k' initially set to m, k respectively.
(4) Choose an integer $q \leq k' - m'$ to be the number of literals included in the clause, with the exception of $m' = 1$, in which case $q = k'$. Decrement counters $k', m' := k' - q, m' - 1$.
(5) We form the literals $b_1, ..., b_q$ for the clause by selecting each literal to be either x_j or $\neg x_j$ for some randomly x_j. Choose a number $1 \leq i \leq q$, and fix b_i to be the *True* based on the preselected assignment.
(6) Add the clause $b_1 \cup ... \cup b_q$ to the formula ϕ.

Throughout the procedure, we only make random selections from finite sets. Each choice of q ensures that the remaining clauses will contain at least one literal. The runtime is clearly polynomial in $l(m, k, p)$. ϕ is satisfiable by construction as under the randomly chosen assignment during generation, each clause contains at least one *True* literal.

If ϕ is a satisfiable formula in CNF form with m clauses, k total literals and n distinct variables, then *some* selection of $G_{m,k,p}$ will generate ϕ. □

A.7 Proof of Proposition 5.4

Proof. For $G = (G_i)_{i\in\mathbb{N}}$ generating L in expected polynomial time, we need to construct a NDTM M that decides L. We define M as follows. For input x with $|x| = i$:

(1) Simulate G_i on each branch for at most $p(i)$ steps.
(2) On each of the branches, branch again nondeterministically whenever a bit on the random input tape is read, for its possible contents, 0 or 1.
(3) When G_i enters an accepting state, accept if the output tape contents is equal to x, otherwise reject.

Without loss of generality, assume that $x \in L_i \subset L$. Then we know there exists a bitstream bs where x is accepted within $p(i)$ steps. Thus there would also be a branch on M where x is produced by the simulation of G_i and thus x would be accepted.

On the other hand, if $x \notin L$, then $x \notin L_i$ for any $i \in \mathbb{N}$, and no simulations of any G_i would produce x. Thus all branches, and hence also M, would reject x. □

A.8 Proof of Proposition 5.3

Proof. $PG^{NP} \subseteq NP^{NP}$: follows since the NP^{NP} machine can simulate the PG^{NP} oracle machine nondeterministically.

$NP \subseteq PG^{NP}$: We use *SAT* as the *NP* oracle. For fixed $n \in \mathbb{N}$ and any $L \in NP$ with NDTM M time-bounded by p, we encode M using the parametric *SAT* formula ϕ_n, provided by the proof of *NP*-completeness of *SAT* [47]. ϕ_n has n variables reserved for the initial contents of the tape, and for $x \in \{0,1\}^{\leq n}$, $\phi_n(x)$ denotes the formula with those variables substituted for by the bits of x.

Note that if $|x| = n$, then by construction, $\phi_n(x)$ is satisfiable if and only if M accepts x within $p(n)$ steps. And if $|x| < n$, then $\phi_n(x)$ is satisfiable if and only if there exists an extension e with $|e| = n - |x|$, and $M(x +\!\!+ e)$ accepts.

Using the *SAT*-solver, we describe a generator G_n that samples a length-n $x \in L$ as follows (if one exists):

(1) Suppose s is the string already generated, and is initially the empty string.
(2) Run the *SAT* solver for both $\phi_n(s +\!\!+ 0)$ and $\phi_n(s +\!\!+ 1)$.
(3) If neither accepts, reject; if only $\phi_n(s +\!\!+ e)$ accepts, set $e' := e$; if both accepts, then choose at random one of 0 or 1 to be e'.
(4) Return to step 2 with string $s := s +\!\!+ e'$.
(5) Write s to the output tape when $|s| = n$.

Clearly this will terminate within n iterations, and each iteration involves at most 2 calls to the *SAT* solver, for a total of $O(n)$ calls. By construction, for any n, any length-n $x \in L$ can be generated in this fashion, thus $\mathcal{L}(G) = L$. □

A.9 Proof of Proposition 5.5

Proof. By Bertrand's Postulate (theorem), there is always a prime between n and $2n$. Thus, we know that for any n, the set $PRIMES_n = \{x \in PRIMES : 2^{|n|} \leq x < 2^{|n|+1}\}$ is nonempty.

We know from [1] that $PRIMES \in P$. Let its decider be a function $f : \{0,1\}^* \to \mathbb{B}$ that runs in polynomial time $p(n)$. For each $n \in \mathbb{N}$, define a generator G_n as follows.

(1) Choose a random number x between $[2^n, 2^{n+1} - 1]$ by sampling for n bits.
(2) If $f(x)$ returns *Accept*, output x.
(3) Otherwise, repeat from step 1.

The density of primes in the range $[1, k]$ is $O(1/log(k))$. Assuming almost-uniform distribution of the primes, their density in the range $[2^n, 2^{n+1}-1]$ is at least $O(1/log(2^{n+1})) = O(\frac{1}{n+1})$. Therefore,

an expected $O(n+1) = O(n)$ iterations is needed for the algorithm to find a prime number. Since f runs in polynomial time p, and we make $n+1$ expected calls to f, the expected runtime of the algorithm is $O(np(n))$, which is still polynomial.

Additionally, for every prime $q \in [2^n, 2^{n+1} - 1]$, there exists a bitstream bs which leads us to selecting $x = q$ in step 1 of the algorithm, and generate x in time $O(p(n))$, lower than $O(np(n))$. Thus *PRIMES* $\in$ *EPG*. □

A.10 Proof of Proposition 5.6

Proof. Suppose L is generated by the expected polynomial time generator $G = (G_i)_{i\in\mathbb{N}}$, with expected time of $p(i)$ for G_i to produce a sample.

Let the runtime of G_i be a random variable X_i with $\mathbb{E}(X_i) = p(i)$. By Markov's inequality, $\mathbb{P}(X_i > 3\mathbb{E}(X_i)) < \frac{\mathbb{E}(X_i)}{3\mathbb{E}(X_i)} = \frac{1}{3}$. That is, the probability of that X_i runs for $3\mathbb{E}(X_i) = 3p(i)$ steps without producing an output is at most 1/3. Therefore, $\mathbb{P}(X_i \leq 3p(i) \geq \frac{2}{3}$.

For each G_i, we define algorithms H_i (these are operationally generators, but are not strictly generators since they have a nonzero probability of producing nothing—and generators require almost sure production of a value) to repeat for k iterations the execution of G_i, running each execution for $3p(i)$ steps. H_i therefore produces a sample of L_i with probability at least $1 - \frac{1}{3}^k$, and runs in time $3kp(i)$.

To exceed probability $1 - \epsilon$, we have: $1 - \frac{1}{3}^k > 1 - \epsilon \implies \epsilon > \frac{1}{3}^k \implies log(\frac{1}{\epsilon}) < klog(3)$. Thus choosing $k = \lceil \frac{log(\frac{1}{\epsilon})}{log(3)} \rceil$ has the desired success probability.

It remains to show that H_i can produce all elements $w \in L_i$ with nonzero probability. This holds because by definition, for all $w \in L_i$, there exists a bitstream bs that leads to G_i producing w in at most $p(i)$ steps. This path can be executed by H_i, since H_i simulates G_i for $3p(i) > p(i)$ steps on each run, and will produce w if bs is on the random tape when G_i is being simulated. □

A.11 Configuration Counting Argument for Corollary 6.1

Note that for Corollary 6.1, an initial direct proof might be to proceed with a configurations-counting argument similar to the decision version, where the space-bounded machine is also automatically bounded since it must terminate. However, due to the fact that generators take input, the situation is more complicated: the input, although read-only, allows the space-bounded machine to stall without exploring its state configuration space fully. Whereas with the same inputs, the time-bounded machine must terminate in bounded time. As a concrete example, consider a logspace generator W which implements *waiting* for the right bitstring input, as follows:

Example A.11.1. *Suppose the logspace generator $W = (W_i)_{i\in\mathbb{N}}$ implements the following for each i:*

1. *If the input tape header points to a* 1*, move the input header forward, and the rest of the machine remains in the starting configuration.*
2. *Otherwise, proceed with a logspace generation algorithm for $\mathcal{L}(W_i)$.*

The waiting process of step 1 does not consume any space: nothing is written onto the work tape. Hence W consumes logspace overall. Clearly, W can take more than polynomial number of steps for generation, if given a sufficiently long prefix of 1*s on the input bitstring.*

We give an alternative proof (similar to the proof of decision procedures) to highlight the intuition behind the the time-space tradeoff for generators, and its similarities/differences to the decision case.

Alternative Proof to Corollary 6.1. Suppose we are given a logspace generator $G = (G_i)_{i\in\mathbb{N}}$. We describe a polynomial-time sampling procedure F_i that works for each $\mathcal{L}(G_i)$.

Definition of the DFA: for each G_i, we produce (based on the transition function of G_i) a DFA D_i. We first define configurations of a generator H:

Definition A.11.1 (Configuration of H). *A configuration c of a generator H, is an element $c = (q, b_i, m, w_{work}, c_o) \in Q \times \{0,1\} \times \{0,1,\ldots,k\} \times \Sigma^* \times \Gamma$, where q is the internal state of H, b_i is the current symbol on the input tape, m is the worktape head position, w_{work} is the worktape content, c_o is the current symbol on the output tape. The length of the worktape content is k.*

Define $n = i$. Note that as the work tape of G_i is space bounded by $f(n)$ for a logarithmic function $f \in O(log(n))$, we only need to consider up to $f(n)$ elements on the worktape within any configuration of G_i. We define the nodes of D_i to be all configurations of G_i, with up to $f(n)$ elements on the work tape. That is, states of D_i are from the set $Q \times \{0,1\} \times \{0,1,\ldots,k\} \times \Sigma^{f(n)} \times \Gamma$. Define the starting state of D_i to be c_0. c_0 has two outward transitions, consuming symbols 0 and 1 respectively, heading towards the two possible starting configurations of G_i: the starting configurations with either a 0 or a 1 as the first symbol on the input tape. Define also the accepting state of D_i to be c_A, a newly added configuration of G_i, with ϵ transitions $c_A^k \to c_A$ from all accepting configurations c_A^k of G_i (these are configurations with $q \in Q$ in the accepting state).

Define a transition on D_i of $c \xrightarrow{1} c'$ (that consumes a 1), if c to c' is a valid step based on G_i's transition function, and c' is identical to c, except the contents of the input cell is replaced by a 1. Define $c \xrightarrow{0} c'$ and $c \xrightarrow{\llcorner\!\lrcorner} c'$ in a similar way. Define a transition $c \xrightarrow{\epsilon} c'$ for configuration changes that do not move the input tape forward. The use of ϵ transitions are for conceptual convenience and do not make our machine an NFA: since G_i is deterministic, at most one such ϵ-transition is possible from each configuration, and that ϵ-transitions cannot coexist with other types of transitions. They can equivalently be replaced by some other reserved symbol, such as $\square$, however such a formulation would be more verbose.

Note that since $|Q|, |\Sigma|, |\Gamma|$ are constants and f is a logarithmic function, the total number of nodes on D_i are bounded by $p(n)$, for some polynomial p. There can be at most a polynomial number of edges, where a loose bound is the number of edges on a complete graph. Thus, the total number of nodes and edges of D_i is bounded by a polynomial $q(n)$.

Proposition A.11.1. *D_i is a DFA.*

Proof. The edges of D_i are defined exactly according to the transition function of G_i. Since the nodes of D_i covers all accessible configurations of the TM (only a single cell for the input and output tapes are needed as the header cannot move back), each transition of G_i that does not move the input header corresponds to at most one edge $c \xrightarrow{\epsilon} c'$ on D_i. As G_i is a deterministic TM (when fixing the input bitstring), there is at most one ϵ edge from any configuration.

For each transition that moves the input tape forward, there is also a unique next state once the revealed character b is fixed. Thus, there cannot exist more than one of each kind of 0, 1, and $\llcorner\!\lrcorner$ edge, from each node on D_i.

Altogether, from each configuration there is at most one edge for each consumable symbol $\{0, 1, \llcorner\!\lrcorner\}$, making D_i a DFA. $\square$

Polynomial Generation Algorithm: for our *PG* algorithm F_i, we describe a traversal of the DFA to find an accepting path, corresponding to an accepting execution of G_i. We will use the following definitions:

Definition A.11.2. *A transition $c \xrightarrow{s} c'$ ($s \in \{0,1,\epsilon\}$) is said to* write *the symbol $x \in \Gamma$, if c points to an empty cell on the output tape, and c' points to a cell with x on the output tape. We also say such a transition is a* writing transition.

Definition A.11.3. *For the execution path $p = c_1 \dots c_k$ on M_i, the output of the path p is the output symbol from all writing transitions used, in order of appearance in the execution path.*

Starting from c_0, perform a randomised BFS, without revisiting any nodes, until reaching the accepting state c_A. This BFS results in a random, loopless path from c_0 to c_A. The runtime of this algorithm is bounded above by $q(n)$, the total number of transitions and nodes in M_i, and thus runs in polynomial time of n.

Correctness: we will show the following: 1) any loopless path from c_0 to c_A on M_i corresponds to an input b and a valid execution $G_i(b)$, and 2) any string generable by G_i can be generated by F_i. That is, $\mathcal{L}(G_i) = \mathcal{L}(F_i)$.

Proposition A.11.2. *Any path p from c_0 to c_A on M_i corresponds to an input b and an execution $G_i(b) = w$, which produces the same output w as p.*

Proof. Any path from c_0 to c_A, by definition, corresponds to an accepting execution on G_i. Since M_i is a DFA, the path p corresponds uniquely to a string $t_1 \dots t_k$ where $t_j \in \{0, 1, \epsilon\}$, representing the label of each transition that has been taken (since the path was accepting, by definition of G_i there cannot be an blank symbol consumed). Define b to be the subsequence consisting of only symbols from the set $\{0, 1\}$. Since G_i is deterministic, and M_i simulates the execution of G_i, $G_i(b)$ will visit precisely the configurations within p during its execution, and produce the same output as p. □

Proposition A.11.3. *Any string generable by G_i is the output of some* loopless *path from c_0 to c_A in M_i.*

Proof. First, note that if the empty string is being generated, then any loops on the path can be removed to obtain another accepting path. Thus assume now that the generated string is nonempty.

Suppose for a contradiction that a loop $r = c_1 \dots c_k$ on M_i exists within an accepting path p, with $c_1 = c_k$, and that it contains some writing transition $c_j \xrightarrow{s} c_{j+1}$ (assuming WLOG that $j \neq k$).

If $p = c_0 \dots r \dots c_A$, then repeating the loop r within p also yields an accepting path through M_i. That is, $p_m = c_0 \dots c_1 (r')^m \dots c_A$, with $m \in \mathbb{N}$ are all accepting paths, where $r' = c_2 \dots c_k$, and $(r')^m$ denotes the repetition of r' m times.

Each p_m also corresponds to an accepting execution on G_i. Since there are writing transitions within r, and G_i is write-once-only, there are at least m characters on the output tape of G_i in the corresponding execution of p_m. In particular, we can choose $m > n$ to reach a contradiction, since G_i is a generator that can only output strings of length $n = i$. □

Our algorithm F_i samples a random loopless accepting path on M_i, and the above propositions together shows that every string from $\mathcal{L}(F_i)$ can be sampled this way.

With this, we can form our *PG* generator $F = (F_i)_{i \in \mathbb{N}}$ to perform, on input i:

(1) Construct M_i described above.
(2) Run the sampling procedure to generate a random string recognised by M_i.

which has a nonzero chance of generating each instance of $\mathcal{L}(G_i)$ in polynomial time. Thus $\mathcal{L}(G)$ is in *PG*, as required. □

A.12 Proof of Proposition 6.2

Proof. First we make some necessary definitions.

Definition A.12.1. *For a relation $R \subseteq X \times Y$, define:*

$$Im(x) = \{y \in Y : (x, y) \in R\} \qquad Dom(R) = \{x \in X : \exists y.(x, y) \in R\}$$

Definition A.12.2. *M is a logspace transducer for the relation $R \subseteq X \times Y$ if, for all $x \in Dom(R)$, $M(x)$ is a generator for $Im(x)$. That is, $\mathcal{L}(M(x)) = Im(x)$.*

Specifically, logspace generators for L are logspace transducers for their length partitions, $(L_i)_{i\in\mathbb{N}}$.

Example A.12.1. *A log-space generator $G = (G_i)_{i\in\mathbb{N}}$ is a nondeterministic logspace transducer for the following relation:*

$$R_G = \{((i, 1^i), x) \in C \times \Sigma^* : \exists b \in \{0,1\}^*.G(i, b) = x\}$$

Proposition A.12.1 (Uniform Sampling Of Logspace Transducers [2]). *If the relation R can be implemented by a logspace transducer, then there exists a polynomial-time sampler, such that on input x, outputs with success probability $> \frac{1}{2}$ a uniform sample from $Im(x)$.*

This leads to a straightforward *EPG* algorithm for generating LG, by applying this procedure repeatedly until a success is reached. □

A.13 Proof of Proposition 6.1

Proof. We will use two lemmas in the proof. The first is the known result of time-space tradeoff for decision machines:

Proposition A.13.1 (Decision Time-Space Tradeoff). *For space constructible $s : \mathbb{N} \to \mathbb{N}$ with $s(i) > log(i)$,*

$$NSPACE(s(n)) \subseteq DTIME(2^{O(s(n))})$$

And the following:

Lemma A.13.1. *For space constructible $s : \mathbb{N} \to \mathbb{N}$, and language $L \in NSPACE(s(n))$, define the decision problem $PARTIAL(L)$ as follows:*

$$PARTIAL(L) = \{(x, 1^k) : \exists r.(x +\!\!+ r) \in L \ \wedge \ |r| = k\}$$

Then $PARTIAL(L) \in NSPACE(s(n))$

Proof. We give an NDTM M with space bound s that decides $PARTIAL(L)$. Assume L is decided by the NDTM D. Given an input $(x, 1^k)$ with total length $n = |x| + k$, guess nondeterministically on each branch a string r of length k. Due to space constraints when $s(n) < n$, the string r is not stored on the work tape but produced on-demand by nondeterministic branching for each symbol in Σ when the input tape header moves forward. We do not need to worry about the input tape header moving backwards and needing to ensure our choices are consistent, since this is assumed if $s(n) < n$.

Then, simulate branches of D on the string $x +\!\!+ r$, accepting if $D(x +\!\!+ r)$ accepts. If there exists a $r \in \{0,1\}^k$ such that $x +\!\!+ r \in L$ then there is a branch that chooses this r, and also there exists an accepting branch on the ensuing simulation of D on $x +\!\!+ r$, thus $(x, 1^k)$ is accepted. Conversely, if there doesn't exist such an r then all branches on the ensuing simulation will reject for all choices of r, thus in this case M will reject $(x, 1^k)$. Hence M decides $PARTIAL(L)$.

As D runs in $s(n)$ space, the simulation of D on M will also run in $s(n)$ space. We have argued earlier that guessing the string r do not require storing the guess onto the work tape, thus space usage remains $s(n)$ for M. □

Given an NDTM M that runs in space $s(i)$ on each branch with language $L = L(M)$, we need to produce a generator $G = (G_i)_{i\in\mathbb{N}}$ such that for each i, G_i runs in time $2^{O(s(n))}$ and has $\mathcal{L}(G_i) = L_i$, the strings of L with length exactly i.

Proposition A.13.1 and Lemma A.13.1 together implies that $PARTIAL(L)$ is decideable in time $O(2^{O(s(n))})$, thus size-i instances of $PARTIAL(L)$ can be decided on the generator G_i. Let its decider be P. G_i proceeds as follows:

(1) Run $P(\epsilon, 1^i)$. If this run rejects, then G_i also rejects.
(2) Initialise $o = \epsilon$.
(3) Generate a random permutation of the m symbols $c \in \Sigma, c_1, \ldots, c_k$.
(4) For $c_i = c_1, \ldots, c_k$, run $P(o +\!\!+ c_i, 1^{i-|o|})$ until one of the c_is accepts. Write this c_i to the output tape, and apply the update $o := o +\!\!+ c_i$.
(5) If $|o| = i$, then terminate. Otherwise continue from step 2.

If $L_i = \emptyset$, then $P(\epsilon, 1^i)$ will reject on step 1, thus $\mathcal{L}(G_i) = \emptyset = L_i$. Otherwise there must exist some choice of $s_1, \ldots, s_i$ at each step, such that $s = s_1 +\!\!+ \ldots +\!\!+ s_i \in L_i$. It is safe to fix the c_is by writing to the output tape since the run of P ensures there exists an extension from that point onwards which belongs in L_i. A different random permutation is necessary for each iteration to ensure all strings in L_i have a chance of being sampled: if the same permutation is used for each step then only the first string from the permutation-induced lexicographical ordering will be produced by the algorithm. Altogether we have $\mathcal{L}(G_i) = L_i$ and hence $\mathcal{L}(G) = L$

At most $|\Sigma|$ calls to P are made on each iteration, together with the extra call of step 1. Thus $i|\Sigma| + 1 \in O(i)$ calls to P are made in total. Auxillary procedures take time $O(i)$. Since $s(i) > log(i)$, the total runtime is $O(i2^{O(s(i))} + i) = O(2^{O(s(i))})$. Therefore $L = \mathcal{L}(G) \in GTIME(2^{O(s(n))})$. □

A.14 Proof of Proposition 6.3

Proof. The construction is based on certificate sampling and instance reconstruction (a la Theorem 7.1). We first define the partition to work on. Let $C = \{(v, e) \in \mathbb{N} \times \mathbb{N} : e \leq v^2\}$, where the tuple $(v, e) \in C \subseteq \{0, 1\}^*$ is interpreted as the binary encoding of two numbers, with v representing the number of vertices and e the number of edges. $l : C \to \mathbb{N}$ is defined as $l((v, e)) = \lceil log(v) \rceil + 2e * \lceil log(v) \rceil + 2\lceil log(v) \rceil$. $l((v, e))$ computes length of string needed to encode graph with v vertices (encoded as a natural number), e edges (encodes as an edge list), along with two nodes i, j in the graph. $l^{-1} : \{1\}^* \to \mathcal{P}(C)$, where $l^{-1}(1^n)$ is defined as a logspace machine that enumerates all possible pairs of integers v, e in their binary representation, with $v, e \leq n$, and checks that $l((v, e)) = n$—output if true, otherwise continues onto the next candidate. l^{-1} runs in logspace of input 1^n, since we are using binary representations.

We can iterate through the elements from the set $\{(v, e) : l(v, e) = n\}$ in logarithmic space, choosing a random element to stop at and run some $G_{v,e}$. It suffices to show that each $G_{v,e}$ exists and runs in logarithmic space of $n = l(v, e)$.

We now describe a procedure that, when given (v, e), generates a graph G with v vertices (labelled by natural numbers), e edges, and two nodes i, j within the graph. The generated G will, by construction, have a directed path between i and j.

Denote $n = l((v, e))$. The procedure is as follows:

(1) Output v in unary, to denote the number of vertices in the graph.
(2) Select two random nodes v_i, v_j with $i, j \leq v$.
(3) Generate a path of length $l \leq v$ via a random walk that starts from v_i and ends at v_j:
    (a) On each step of the random walk, we write the edge taken to the output tape.
    (b) When v_j is reached, stop the random walk.
    (c) If, by the time we traversed $l - 1$ nodes and still have not reached v_j, go directly to v_j on the next step.
(4) Add e random edges to the graph in Erdos-Renyi fashion. Output each chosen edge immediately.

(5) Output the earlier i, j.

The generated graph G has a path from v_i to v_j by construction, and thus the full outputted instance is a member of *REACH*.

For any given directed graph H with v vertices, e edges, and a path from x to y there exists a bitstring that makes the following choices:

(1) Chooses exactly $i = x, j = y$ to be the start/endpoints of the random walk.
(2) Chooses exactly the path from x to y on H in the random walk.
(3) Fills in exactly the rest of the edges of H in step 4.

and thus $\langle H, x, y\rangle$ is generable by the above procedure.

Throughout the procedure, we store the binary representation of v, the chosen vertices i, j, the counter for the random walk up to v in length, and the counter for the number of edges up to v^2. All of which are representable in space $O(log(n))$. □

A.15 Proof of Proposition 6.4

Proof. $GSPACE(s(n))^f \subseteq NSPACE(s(n))^f$ follows from the same simulation argument as for the non-oracle case.

For $NSPACE(s(n))^f \subseteq GTIME(2^{O(s(n))})^f$, we need the following lemma for the time-space tradeoff on oracle machines:

Lemma A.15.1. *For length-bounded $f : \Sigma^* \to \Sigma^*$, $NSPACE(s(n))^f \subseteq DTIME(2^{O(s(n))})^f$*

Proof. Take the nondeterministic oracle machine M_f, space-bounded by the polynomial s and recognising some $L \in NSPACE(s(n))^f$. Suppose we are given an $x \in \Sigma^*$ with $|x| = n$. Note that the query tape has size at most $s(n)$, and that the contents of the query tape fixes the contents of the answer tape. Since the largest size query that can be made is of length $s(n)$, the answer tape can record at most $2^{O(s(n))}$ nonempty cells (as f is length-bounded by $2^{O(n)}$), which is indexable by a pointer of size $O(s(n))$. As the answer tape is read-only, the number of possible configurations on both the query and the answer tapes is $2^{O(s(n))}$. The work tape and the input tape contribute another $2^{O(s(n))}$ configurations to the total state space, meaning the overall possible configurations for M_f remain $2^{O(s(n))}$. Querying edges on the configuration graph can be done by querying M_f's definition. Hence, a standard graph algorithm through the configuration graph can decide whether $x \in L$ in time $2^{O(s(n))}$. □

An identical construction to the proof of Proposition 6.1 then follows: as the oracle variant of $PARTIAL(L)$ is still within $NSPACE(s(n))^f$, the bit-by-bit construction can be used to construct all elements of $L \in NSPACE(s(n))^f$ at random. □

A.16 Proof of Proposition 7.1

Proof. Clearly if $L \in PG$ then $DEFAULT(L) \in FP$ by running the generator $G$for L.

Suppose now $DEFAULT(L) \in FP$. With a computable default element as a fallback, we are free to sample the *NP* computation even though there is a chance of failure (i.e. finding a rejecting branch). This is similar in idea to the proof of Proposition A.4.1. The following procedure $G = (G_i)_{i\in\mathbb{N}}$ is an algorithm for sampling L, given an *NP* machine M that decides L in polynomial time. Assume $DEFAULT(L)$ is implemented by the machine D which, on input 1^i, will output either ϵ or a string in L_i. Define G_i as:

(1) Sample a random string x of length i.
(2) Simulate a single branch of $M(x)$. Output x if $M(x)$ accepts.
(3) Otherwise, run $D(1^i)$. If $D(1^i) = y \neq \epsilon$ then output y, otherwise reject.

G_i runs in polynomial time since both M and D are polynomial-time bounded. Clearly all of $x \in L_i$ is samplable this way. □

A.17 Proof of Proposition 6.5

Proof. For $PSPACEG^f \subseteq PSPACE^f$, we know from Proposition 6.4 that $PSPACEG^f \subseteq NPSPACE^f$. We show that Savitch's theorem still holds for the oracle case: $NPSPACE^f \subseteq PSPACE^f$, which implies $PSPACEG^f \subseteq PSPACE^f$. Suppose M_f is an NDTM deciding some $L \in NPSPACE(s(n))^f$, and space-bounded by the polynomial s. Given any $x \in \Sigma^*$ with $|x| = n$, we want to decide whether $x \in L$. In the proof of Lemma A.15.1 we see that the number of configurations on M_f remains $2^{O(s(n))}$. Thus, using the divide-and-conquer method detailed in the proof of Savitch's theorem, we get a polynomial $O(s^2(n))$ space deterministic algorithm for L, using the same oracle f, implying $L \in PSPACE^f$.

$PSPACE^f \subseteq PSPACEG^f$: suppose M_f decides a language $L \in PSPACE^f$. G_f is defined as follows: on input n, choose a enumeration of the strings $x \in \Sigma^n$, and simulate the $PSPACE^f$ machine on x. If $M_f(x)$ accepts then accept and output x with probability $1/2$. If all strings x are rejected by M_f, then G_f rejects. If all xs are exhausted before we chose to accept, loop back to the beginning of the enumeration. Clearly $x \in L$ has length n if and only if x has nonzero probability of being generated. □

A.18 Proof of Proposition 7.2

Proof. We show that L is in PG by constructing a generator $G_L = (G_{L,i})_{i\in\mathbb{N}}$ for it. Since $K \in PG$, let $G_K = (G_{K,i})_{i\in\mathbb{N}}$ be the generator for K, where $G_{K,i}$ runs in time $p_K(i)$ for some polynomial p_K. Also, let G_s be the generator from the Efficient Extension condition for $s \in K$. Since G_s is polynomial-time, it runs in time $p_E(|s|)$ for some polynomial p_E.

For any index $n \in \mathbb{N}$, we define $G_{L,n}$ as follows:

(1) Use $G_{K,n}$ to generate a sample $s \in K$. Note that since $(L_n)_{n\in\mathbb{N}}$ partitions L by size, and $K \subseteq L$, $G_{K,n}$ generates elements in $K \cap L_i$. Thus $|s| = n$.
(2) Run the extension generator G_s to obtain a string s'.
(3) Output s'.

The total runtime is dominated by the runtime of $G_{K,i}$ and G_s. Since both are polynomial in n, the composed generator runs in polynomial time. By the property of G_s, $s \preccurlyeq s'$. Since $s \in K \subseteq L$, by Upward Closure, $s' \in L$. Also $|s'| = |s| = n$, so $s' \in L_i$. For any $w \in L_i$, since K contains all minimal elements of L, there exists some $s \in K$ such that $s \preccurlyeq w$. Since we restrict extensions to same-length strings (by the definition of G_s), we must have $|s| = |w| = n$, so $s \in K \cap L_i$. $G_{K,i}$ generates s with non-zero probability. G_s generates w from s with non-zero probability (since $w \in \{s' : s \preccurlyeq s' \wedge |s'| = |s|\}$). Thus, $G_{L,i}$ generates w with non-zero probability.

Therefore, G_L is a polynomial-time generator for L, and $L \in PG$. □

A.19 Examples of Generators via Basis Extension

Example A.19.1 (CNF-SAT revisited). *We show that CNF-SAT is in PG using Proposition 7.2. We define the size of a formula by the number of variables n and clauses m, using a fixed-length encoding. We define the partial order $\preccurlyeq$ such that $\phi_1 \preccurlyeq \phi_2$ if ϕ_1 and ϕ_2 have the same number of clauses, and each clause in ϕ_2 contains the literals of the corresponding clause in ϕ_1 (i.e., ϕ_2 is obtained by adding literals to ϕ_1).*

- ***Upward Closure**: Adding literals to a disjunctive clause makes it strictly easier to satisfy. Thus if ϕ_1 is satisfiable, ϕ_2 is also satisfiable.*

- ***Samplable Minimal Basis**: The minimal elements K are satisfiable formulas where no literals can be removed from clauses while maintaining the structure (i.e., unit clauses). This corresponds to consistent partial assignments. We can sample these by generating random consistent assignments for the n variables, and using the active literals as the unit clauses.*
- ***Efficient Extension**: Given a consistent assignment (formula of unit clauses), we can extend it to a full formula of the target size by adding random literals to each clause.*

Example A.19.2 (Hamiltonian Path). *Let HAMPATH be the set of graphs (encoded as adjacency matrices of size n^2) that contain a Hamiltonian path. We define $\preccurlyeq$ by subgraph inclusion: $G_1 \preccurlyeq G_2$ if $E(G_1) \subseteq E(G_2)$.*

- ***Upward Closure**: Adding edges to a graph preserves the existence of a Hamiltonian path.*
- ***Samplable Minimal Basis**: The minimal elements are graphs consisting of exactly one Hamiltonian path (and no other edges). These can be sampled efficiently by generating a random permutation of the vertices.*
- ***Efficient Extension**: Given a graph G consisting of a Hamiltonian path, we can extend it to a random graph G' of the same size (adjacency matrix representation) by setting additional entries in the adjacency matrix to 1 with some probability.*

A.20 Proof of Theorem 7.1

Proof. (2) $\implies$ (1) is clear, as the composition of the certificate generator with the certificate recoverer leads to a polynomial-time generator of L. That is, suppose the recovery scheme is (S_V, R_V). Then $G_i = R_i \circ S_i$ is the generator for L_i that runs in time $p(i, q(i))$, which is polynomial in i. And thus L has a polynomial-time generator.

To show (1) $\implies$ (2), we need to construct a verifier V, along with its associated recovery scheme (S_V, R_V), from a polynomial-time generator $G = (G_i)_{i\in\mathbb{N}}$ of L.

Define V to use bitstrings as certificates. On input $w\#c$ with $|w| = n$, simulate G_n on input c until termination, and accept if the output of G_n is w, reject otherwise. Since G_n will terminate in time $p(n)$, for a polynomial p, it will consume at most $p(n)$ random bits during any run. We define the certificates c of w to be any bitstream of length up to $p(n)$ that leads to w being produced by G_n. If $w \in L$ then there exists a bitstream (certificate) c that leads to $V(w\#c)$ accepting. On the other hand, if $w \notin L$, then $V(w\#c)$ will always reject for any bitstrings c. Thus all conditions of V being a polynomial-time verifier for L is satisfied.

We now need to show that the certificate recovery scheme (S_V, R_V) exists. Producing a random certificate is easy, as we are using bitstrings as certificates. By definition, G_i terminates in time $p(i)$. For $S_V = (S_i)_{i\in\mathbb{N}}$, we define each S_i as follows:

(1) Simulate G_i to determine if L_i is empty. Reject if $L_i = \emptyset$.
(2) Output a random bitstring of length $p(i)$.

To see that S_V is indeed a certificate generator, suppose firstly $w \in L_i = \mathcal{L}(G_i)$. Then there exists some $p(i)$-length bitstring c where $G_i(c) = w$, hence $V(w\#c)$ accepts by definition, and $w \in V_f(c)$. We know c is samplable from S_i since it samples all bitstrings of length $p(i)$, hence $c \in \mathcal{L}(S_i)$. Thus $w \in V_f(c)$, with $c \in \mathcal{L}(S_i)$. If $L_i = \emptyset$ then S_i indeed rejects. Therefore we have $L_i \subseteq \Sigma^i \cap \bigcup_{c\in\mathcal{L}(S_i)} V_f(c)$.

By definition of V, if $V(w\#c)$ accepts then $w \in L$. Hence $V_f(c) \subseteq L$, and $\Sigma^i \cap V_f(c) \subseteq L_i$ for any c. It follows $\Sigma^i \cap \bigcup_{c\in\mathcal{L}(S_i)} V_f(c) \subseteq L_i$.

Altogether, we have condition 1 of the certificate generator. Condition 2 of the certificate generator clearly holds since S_i takes exactly $p(i)$ steps, which is a polynomial in i.

For R_V, we define for each $i \in \mathbb{N}$, $R_i(c, b) = G_i(c)$. That is, each R_i will simulate G_i on the certificate c it received on the certificate input tape, and ignore its own random tape. Condition 2

of certificate recoverer holds since $L_i \cap V_f(c)$ contains the single string $G_i(c)$, which is the only output of $R_i(c, b)$ for any choices of bitstring b. Finally, condition 3 of certificate recoverer holds as G_i accepts within $p(i)$ steps, which is a polynomial in parameters i and $|c|$ in the trivial way. Thus R_V also satisfies all conditions of a certificate recoverer. □

A.21 Proof of Proposition 7.3

Proof. Suppose $G = (G_i)_{i\in\mathbb{N}}, H = (H_i)_{i\in\mathbb{N}}$ with $\mathcal{L}(G), \mathcal{L}(H) \in PG$. Assume G and H run in polynomial time bounds p and q respectively.

Concatenation. Define $G +\!\!+ H$ as follows. For index i: Sample a random $j \in 1, \ldots, i-1$ and define $k = i - j$. Run G_j first, followed by H_k on an empty work tape immediately after G_j terminates. If any of G_j or H_k rejected, set $j := j + 1$ and retry. If all of $j \in 1, \ldots i-1$ failed, then reject. The total time is bounded by $i(p(i) + q(i))$ to generate an input of size i, which is still polynomial in i.

Kleene Star. For G^*, given an index i, we need to produce a length i string that is the concatenation of shorter strings from $\mathcal{L}(G)$. At first sight, this seems difficult, since the partition function for the integer i grows at rate $O(2^{\sqrt{n}})$, and each partition needs to be checked to determine whether $\mathcal{L}(G_i^*) = \emptyset$.

However, we can define a polynomial time generator for $\mathcal{L}(G_i^*)$ by reducing to the Knapsack problem. First run all $G_1, \ldots, G_i$ in sequence to obtain a sequence of $l_1, \ldots, l_m$ string lengths where for all $j \in 1, \ldots m$, $\mathcal{L}(G_{l_j}) \neq \emptyset$. Then run the dynamic programming algorithm of Knapsack to extract a random way to sum up to i using $l_1, \ldots, l_m$. Reject if no solution exists. Finally, run each G_{l_j} in sequence, concatenating the outputs from each run to produce G_i^*. This algorithm clearly produces some string in $\mathcal{L}(G_i^*)$ if one exists. On the other hand, each partition $l_1, \ldots, l_m$ of i has nonzero probability of being extracted from the Knapsack algorithm, and for each j, any string $x_j \in \mathcal{L}(G_{l_j})$ has nonzero probability of being produced by their respective generators. Thus, all members of $\mathcal{L}(G*_i)$ also have nonzero probability of being sampled.

Determining the sequence of l_js takes time $O(ip(i))$, Knapsack runs in time $O(mi) = O(i^2)$, and the final output stage takes time bounded above by $O(ip(i))$. Thus the total runtime remains polynomial in i.

Union. For index i, select one of G_i or H_i at random to run first. If the first machine rejects, then run the second machine. Reject if both runs reject, otherwise output what is written on the output tape from the first accepting run. This generates all strings $x \in \mathcal{L}(G_i) \cup \mathcal{L}(H_i)$. □

A.22 Proof of Theorem 7.2

Proof. Suppose for a contradiction that given $L_1, L_2 \in PG$, $L_1 \cap L_2 \in PG$. We will show that $PG \supseteq NP$, which contradicts Theorem 5.1. Take any $L \in NP$, and define the following languages:

$$L^* = \{00x : x \in L\} \qquad A = \{01y : y \in \{0,1\}^*\} \qquad B = \{10y : y \in \{0,1\}^*\}$$

Observe that if $L^* \in PG$, then also $L \in PG$ by taking a suffix.

We claim both $L^* \cup A \in PG$ and $L^* \cup B \in PG$. To see this, note that $DEFAULT(L^* \cup A) \in FP$ as the function $f : \{1\}^* \to (L^* \cup A)$ defined as $f(\epsilon) = f(1) = f(11) = \epsilon$, and $f(1^k) = 010^{k-2}$ for $k \geq 2$ computes the default elements of $L^* \cup A$ in time linear in k. By Proposition 7.1, $L^* \cup A \in PG$. $L^* \cup B \in PG$ follows by a similar argument.

By assumption, $L^* = (L^* \cup A) \cap (L^* \cup B) \in PG$, where the equality comes from the fact that $L^* \cap A = A \cap B = L^* \cap B = \emptyset$. Thus, $L^* \in PG$, which implies $NP \subseteq PG$, and we get the desired contradiction. □