

Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning

Sameer Reddy

University of California, Berkeley, USA
sameer.reddy@berkeley.edu

Rohan Padhye

University of California, Berkeley, USA
rohanpadhye@cs.berkeley.edu

ABSTRACT

Property-based testing is a popular approach for validating the logic of a program. An effective property-based test quickly generates many diverse valid test inputs and runs them through a parameterized test driver. However, when the test driver requires strict validity constraints on the inputs, completely random input generation fails to generate enough valid inputs. Existing approaches to solving this problem rely on whitebox or greybox information collected by instrumenting the input generator and/or test driver. However, collecting such information reduces the speed at which tests can be executed. In this paper, we propose and study a black-box approach for generating valid test inputs. We first formalize the problem of guiding random input generators towards producing a diverse set of valid inputs. This formalization highlights the role of a *guide* which governs the space of choices within a random input generator. We then propose a solution based on reinforcement learning (RL), using a tabular, on-policy RL approach to guide the generator. We evaluate this approach, *RLCheck*, against pure random input generation as well as a state-of-the-art greybox evolutionary algorithm, on four real-world benchmarks. We find that in the same time budget, *RLCheck* generates an order of magnitude more diverse valid inputs than the baselines.

ACM Reference Format:

Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2019. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nmnnnn.nmnnnn>

1 INTRODUCTION

Property-based testing is a powerful method for testing programs expecting highly-structured inputs. Popularized by QuickCheck [16], the method has grown thanks to implementations in many different languages [1, 2, 7, 8, 39], including prominent languages such as Python [3], JavaScript [4], and Java [26]. Property-based testing allows users to specify a test as $\forall x \in \mathcal{X} : P(x) \Rightarrow Q(x)$, where \mathcal{X} is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nmnnnn.nmnnnn>

Caroline Lemieux

University of California, Berkeley, USA
clemieux@cs.berkeley.edu

Koushik Sen

University of California, Berkeley, USA
ksen@cs.berkeley.edu

a domain of inputs and P, Q are arbitrary predicates. The precondition $P(x)$ is some sort of *validity constraint* on x . A property testing tool quickly generates many inputs $x \in \mathcal{X}$ and runs them through a test driver. If the tool generates an x for which $P(x)$ holds but $Q(x)$ does not hold, then the test fails.

Using a property-based testing tool requires two main steps. First, the user needs to write a *parameterized test driver*, i.e., the programmatic representation of $P(x) \Rightarrow Q(x)$. Second, the user needs to specify the *generator* for $x \in \mathcal{X}$. A generator for \mathcal{X} is a non-deterministic program returning inputs $x \in \mathcal{X}$. Testing frameworks typically provide generators for standard types (e.g. primitive types or a standard collections of primitives). If \mathcal{X} is a user-defined type, the user may need to write their own generator.

For property-based testing to be effective, the generator must produce a *diverse* set of inputs $x \in \mathcal{X}$ satisfying the validity constraint $P(x)$. This is a central conflict in property-based testing. On the one hand, a simple generator is easier for the user to write, but not necessarily effective. On the other hand, a generator that produces diverse valid inputs is good for testing, but very tedious to write. Further, generators specialized to a particular validity function $P(x)$ cannot be reused to test other properties on \mathcal{X} with different validity constraints, say $P'(x)$. We thus want to solve the following problem: given a generator G of inputs $x \in \mathcal{X}$ and a validity function $P(x)$, automatically guide the generator to produce a variety of inputs x satisfying $P(x)$.

One way to solve this problem is to do whitebox analysis [13, 21–23] of the generator and/or the implementations of $P(x)$ and $Q(x)$. A constraint solver can be used to generate inputs $x \in \mathcal{X}$ that are guaranteed to satisfy $P(x)$, which also exercise different code paths within the implementation of $Q(x)$ [41]. Another set of approaches, adopted in recent greybox fuzzing-inspired work [30, 38], is to collect code coverage during test execution. This information can be used in an evolutionary algorithm that can generate inputs that are likely satisfy $P(x)$, while optimizing to increase code coverage through $Q(x)$. Both sets of approaches require instrumenting the program under test; thus, they are limited in terms of performance. Purely whitebox approaches have trouble scaling to complex validity functions due to the path explosion problem [15], and even greybox fuzzing techniques require instrumentation that can lead to slowdowns, thereby reducing the rate at which tests can be executed. This is in conflict with the QuickCheck approach, in which properties can be *quickly* validated without instrumenting the test

program. In this paper, we address this gap by investigating a *black-box approach* for guiding a generator G to produce a large number of diverse valid inputs in a *very short amount of time*.

In this paper, we first formalize the problem of guiding the random choices made by a generator for effective property testing as the *diversifying guidance problem*. Second, we notice that the diversifying guidance problem is similar to problems solved by reinforcement learning: given a sequence of prior choices (state), what is the next choice (action) that the generator should make, in order to maximize the probability of generating a new x satisfying $P(x)$ (get high reward)? We thus explore whether reinforcement learning can solve the diversifying guidance problem. We propose an on-policy table-based approach which adapts its choices on-the-fly during testing time. We call our technique *RLCheck*.

We implement *RLCheck* in Java and evaluate it to state-of-the-art approaches Zest [38] and QuickCheck. We compare the techniques on four real-world benchmarks used in the original evaluation of Zest: two benchmarks that operate on schema-compliant XML inputs, and two that operate on valid JavaScript programs. We find that *RLCheck* generates $1.4\times\text{--}40\times$ more diverse valid inputs than state-of-the-art within the same tight time budget. All methods are competitive in terms of valid branch coverage, but the simple RL algorithm we explore in *RLCheck* may be biased to certain parts of the valid input space. We also find that a basic addition of greybox feedback to *RLCheck* does not produce improvements that are worth the instrumentation slowdown.

In summary, we make the following contributions:

- We formalize the problem of making a generator produce many valid inputs as the *diversifying guidance problem*.
- We propose a reinforcement learning approach to solve the diversifying guidance problem, named *RLCheck*.
- We evaluate *RLCheck* against the state-of-the art Zest tool [38] on its valid test-input generation ability.
- We provide our code as open-source for replication.¹

2 A MOTIVATING EXAMPLE

Property-based testing tools [9, 16, 18, 20, 21, 39, 46] allow users to quickly test a program by running it with many inputs. Originally introduced for Haskell [16], property-based testing has since been ported to many other programming languages [1–4, 7, 8, 26, 39]. Using a property-based testing tool involves two main tasks.

First, the user must write a *parameterized test driver* which takes input x of some type X and runs test code checking a property $P(x) \Rightarrow Q(x)$. We say x is *valid* if it satisfies $P(x)$.

For example, in Figure 1, the test driver `test_insert` (Line 15) is given a binary tree `tree` and an integer `to_add` as input. If `tree` is a binary search tree (Line 16), the driver inserts `to_add` into `tree` (Line 17) and asserts that `tree` is still a binary search tree after the insert (Line 18). The `assume` at Line 16 terminates the test silently if `tree` is not a binary search tree. The `assert` at Line 18 is violated if `tree` is not a binary search tree after the insertion. Thus, the test driver implements $P(x) \Rightarrow Q(x)$ for the validity constraint $P(x) = "x$ is a binary search tree” and the post-condition $Q(x) = "after inserting `to_add`, x is a binary search tree” – by raising an assertion failure when $P(x) \Rightarrow Q(x)$ is falsified. In this example, the predicate $Q(x)$$

¹URL omitted for double-blind. Code for the generators in supplementary material.

```

1   from generators import generate_int
2
3   def generate_tree(depth=0):
4       value = random.Select([0, 1, ..., 10])
5       tree = BinaryTree(value)
6       if depth < MAX_DEPTH and
7           random.Select([True, False]):
8           tree.left = generate_tree(depth+1)
9       if depth < MAX_DEPTH and
10          random.Select([True, False]):
11             tree.right = generate_tree(depth+1)
12     return tree
13
14 @given(tree = generate_tree, to_add = generate_int)
15 def test_insert(tree, to_add):
16     assume(is_BST(tree))
17     BST_insert(tree, to_add)
18     assert(is_BST(tree))

```

Figure 1: Pseudocode for a property-based test. `generate_tree` generates a random binary tree, and `test_insert` tests whether inserting a given integer into a given `tree` preserves the binary search tree property. `random.Select(D)` returns a random value from `D`.

is specified explicitly via assertions. $Q(x)$ can also be implicitly defined, e.g. $Q(x)$ may be the property that the program does not crash with a segfault when executed on x .

Second, the user must specify how to generate random inputs for the test driver. This must be done by writing or specifying a *generator*, a non-deterministic function returns a random input of a given type in each of its executions. Property-based testing frameworks typically provide generators for basic types such as primitive types and predefined containers of primitives (e.g. `generate_int` in Figure 1). If the test function takes a user-defined data structure, such as the `tree` in Figure 1, Line 15, the user writes their own generator. For many types, writing a basic generator is fairly straightforward. In Figure 1, `generate_tree` generates a random binary tree by (1) choosing a value for the root node (Line 4), (2) choosing whether or not to add a left child (Line 7) and recursively calling `generate_tree` (Line 8), and (3) choosing whether or not to add a right child (Line 10) and recursively calling `generate_tree` (Line 11). We have deliberately kept this generator simple in order to have a small motivating example.

The user can now run `test_insert` on many different trees to try and validate $P(x) \Rightarrow Q(x)$: the `assume` in Line 16 effectively filters out invalid (non-BST) inputs. Unfortunately, this rejection sampling is not an effective strategy if $P(x)$ is too strict. If the generator has no knowledge of $P(x)$, it will, first of all, very rarely generate valid inputs. So, in a fixed time budget, very few valid inputs will be generated. The second issue is that the generator may not generate very diverse valid inputs. That is, the only valid inputs the generator has a non-negligible probability of generating may be *very small* valid inputs; these will not exercise a variety of behaviors in the code under test. For example, out of 1000 generated

binary trees, the generator in Figure 1 only generates 20 binary search trees of size ≥ 3 , and only one binary search tree of size 4 and 5, respectively. Overall, the generator has very low probability of generating complex valid inputs, which greatly decreases the efficacy of the property-based test.

Fundamentally, the input generated by a generator is governed by the choices taken at various *choice points*. For example, in Figure 1, at Line 4, the generator makes a *choice* of which integer value to put in the current node, and it chooses to make a left or right child at Lines 7 and 10, respectively. Depending on the prior sequence of choices taken by the generator, only a subset of the possible choices at a particular choice point may result in a valid input. For example, if $P(x)$ is the binary search tree invariant, when generating the right child of a node with value n , the only values for the child node that can result in a valid BST are those greater than n . While narrowing the choice space in this manner is straightforward for BSTs, manually encoding these restrictions is tedious and error-prone for complex real-world validity functions.

Overall, we see that the problem of guiding G to produce many valid inputs boils down to the problem of narrowing the choice space at each choice point in the generator. We call this the *diversifying guidance problem*. We formalize this problem in Section 3 and propose a solution based on reinforcement learning in Section 4.

3 PROBLEM DEFINITION

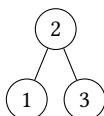
In property-based testing, a generator G is a non-deterministic program returning elements of a given space \mathcal{X} . For example, in Figure 1, \mathcal{X} is the set of binary trees of depth up to `MAX_DEPTH` with nodes having integer values between 0–10, inclusive.

In particular, a generator G 's non-determinism is entirely controlled by the values at returned at different *choice points* in the generator. A choice point p is a tuple (ℓ, C) where $\ell \in \mathbb{L}$ is a program location and $C \subseteq \mathcal{C}$ is a finite domain of choices. For example, there are three choice points in the generator in Figure 1:

- (Line 4, [0, 1, ..., 10]): the choice of node value;
- (Line 7, [True, False]): whether to generate a left child; and
- (Line 10, [True, False]): whether to generate a right child.

During execution, each time the generator reaches a choice point (ℓ, C) , it makes a choice $c \in C$. Every execution of the generator, and thus, every value produced by the generator, corresponds to a sequence of choices made at these choice points, say c_1, c_2, \dots, c_n .

For example, the execution through `generate_BST` in Figure 1 which produces the tree:

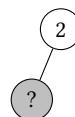


corresponds to the following sequence of choices c_1, c_2, \dots, c_9 :

Choice Index	Choice Taken	Choice Point	
c_1	2	(Line 4, [0, 1, ..., 10])	313
c_2	<code>True</code>	(Line 7, [True, False])	314
c_3	1	(Line 4, [0, 1, ..., 10])	315
c_4	<code>False</code>	(Line 7, [True, False])	316
c_5	<code>False</code>	(Line 10, [True, False])	317
c_6	<code>True</code>	(Line 10, [True, False])	318
c_7	3	(Line 4, [0, 1, ..., 10])	319
c_8	<code>False</code>	(Line 7, [True, False])	320
c_9	<code>False</code>	(Line 10, [True, False])	321

As the generator executes, each time it reaches a choice point $p = (\ell, C)$, it will have already made some choices c_1, c_2, \dots, c_k . Traditional QuickCheck generators, like the one in Figure 1, will simply choose a random $c \in C$ at choice point p regardless of the prefix of choices c_1, c_2, \dots, c_k .

Going back to our running example, suppose the generator has reached the choice point choosing the value of the left child of 2, i.e. choosing what to put in the ? :



That is, the generator has made the choices $[c_1 = 2, c_2 = \text{True}]$, and must now choose a value from 0–10 at the choice point in Line 4. The generator is equally likely to pick any number from this range. Since only 2 of the 11 numbers from 0–10 are smaller than 2, it has at most an 18% chance of producing a valid BST.

To increase the probability of generating valid inputs, the choice at this point should be made not randomly, but according to a *guide*. In particular, according to a guide which restricts the choice space to only those choices which will result in a binary search tree. First, we formalize the concept making choices according to a guide.

Definition 3.1 (Following a Guide). We say that a generator G follows a guide $\gamma : C^* \times P \times \mathbb{N} \rightarrow C$ if: during its t^{th} execution, given a sequence of past choices $\sigma = c_1, c_2, \dots, c_k$, and the current choice point $p = (\ell, C)$, the generator G makes the choice $\gamma(\sigma, p, t)$.

Suppose we have a validity function $v : \mathcal{X} \rightarrow \{\text{True}, \text{False}\}$ which maps elements, output by the generator, to their validity. For example, `is_BST` is a validity function for the generator in Figure 1. The *validity guidance problem* is the problem of finding a guide that leads the generator to produce valid elements of \mathcal{X} :

Definition 3.2 (Validity Guidance Problem). Let G be a generator producing elements in space \mathcal{X} . Let $v : \mathcal{X} \rightarrow \{\text{True}, \text{False}\}$ be a validity function. The *validity guidance problem* is the problem of creating a guide γ such that:

if G follows γ , then $v(x) = \text{True}$ for any $x \in \mathcal{X}$ generated by G .

Note that a solution to the validity guide problem is not necessarily useful for testing. In particular, the guide γ could simply hard-code a particular sequence of choices through the generator which results in a valid element $x \in \mathcal{X}$. Instead, we want to generate valid inputs with diverse *characteristics*. For example, we may want to generate unique valid inputs, or valid inputs of different lengths, or valid inputs that exercise different execution paths in

313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391

393 the test program. We use the notation $\xi(x)$ to denote an input's
 394 characteristic of interest, such as identity, length, or code coverage.
 395

396 **Definition 3.3** (diversifying guidance problem). Let G be a
 397 generator producing elements in space \mathcal{X} . Let $v : \mathcal{X} \rightarrow \{\text{True}, \text{False}\}$ be a
 398 validity function and ξ be a characteristic function. The *diversifying*
 399 *guidance problem* is the problem of creating a guide γ such that:
 400 if G follows γ and $X_T \subseteq \mathcal{X}$ is the set of inputs generated by G after
 401 T executions, $|\{\xi(x) : x \in X_T \wedge v(x)\}|$ is maximized.

402 If ξ is the identity function, then a solution to the diversifying
 403 guidance problem is a guide which maximizes the number of unique
 404 valid inputs generated.
 405

4 METHOD

406 In this section we describe our proposed solution to diversifying
 407 guidance problem. In particular, our proposed guide uses reinforce-
 408 ment learning to makes its choices. We begin with a basic back-
 409 ground on Monte Carlo Control [45], the table-based reinforcement
 410 learning method used in this paper. We then describe how it can
 411 be used to solve the diversifying guidance problem.

4.1 Reinforcement Learning

412 We begin by defining a version of the problem solved by reinforce-
 413 ment learning which is relevant to our task at hand. We use a
 414 slightly nontraditional notation for consistency with the previous
 415 and next sections. What we call *choices* are typically called *actions*,
 416 and what we call a *learner* is typically called an *agent*.

417 We assume an learner in some environment. The learner can
 418 perceive the *state* s of the environment, where s is in some set of
 419 states \mathcal{S} . At the first point in time, the learner is at an initial state
 420 $s_0 \in \mathcal{S}$. At each point in time, the learner can make a choice $c \in C$
 421 which will bring it to some new state $s' \in \mathcal{S}$. Eventually, the agent
 422 gets into some terminal state $s_{\text{term}} \in \mathcal{S}$, indicating the end of an
 423 *episode*. An episode is the sequence of (*state*, *choice*) pairs made
 424 from the beginning of time up to the terminal state, i.e.:
 425

$$e = (s_0, c_0), (s_1, c_1), \dots, (s_T, c_T)$$

426 Where the choice c_T in state s_T brings the learner to the terminal
 427 state s_{term} . Finally, we assume we are given a reward r for a given
 428 episode e . A larger reward is better.

429 The problem to solve is the following. Given a state space \mathcal{S} ,
 430 choices C , and reward r , find a policy π which maximizes the
 431 expected reward to the learner. That is, find a π such that if the
 432 learner, at each state $s \in \mathcal{S}$, makes the choice $c = \pi(s)$, then the
 433 expected reward $\mathbb{E}_{\pi, e}[r]$ from the resulting episode e is maximized.

434 **4.1.1 Monte Carlo Control.** One approach to solving the policy-
 435 learning problem above is by on-policy Monte Carlo Control [45].
 436 The technique is *on-policy* because the policy the learner is optimiz-
 437 ing is the same one it is using to control its actions. Thus, a Monte
 438 Carlo Control learner L defines both a policy π , where $\pi(s)$ outputs
 439 a choice c for the given state s , as well as an update procedure that
 440 improves π after each episode.

441 Algorithm 1 shows pseudocode for a Monte Carlo Control (MCC)
 442 learner L . In the algorithm, we subscript the choice space, state
 443 space, and Q and $counts$ with L to emphasize these are independent
 444 ones, which the discount factor implicitly does.

445 **Algorithm 1** A Monte Carlo Control learner L . Implements a policy
 446 π_L and an update function UPDATE_L which updates π_L towards the
 447 optimal policy after each episode.

448 **Input:** choice space C_L , state space S_L , and ε_L

449 1: $e_L \leftarrow []$ ▷ initialize episode
 450 2: **for** $(s, c) \in S_L \times C_L$ **do**
 451 3: $counts_L[s, c] \leftarrow 0$
 452 4: $Q_L[s, c] \leftarrow 0$
 453 5: **procedure** $\pi(\text{state } s)$
 454 6: **if** $\text{UNIFORMRANDOM}() < \varepsilon$ **then**
 455 7: $c \leftarrow \text{RANDOM}(C)$
 456 8: **else**
 457 9: $c \leftarrow \arg \max_{c \in C_L} Q_L[s, c]$ ▷ break ties arbitrarily
 458 10: $e_L \leftarrow \text{APPEND}(e_L, (s, c))$
 459 11: **return** $choice$
 460 12: **procedure** $\text{UPDATE}(\text{reward } r)$
 461 13: $T \leftarrow \text{LEN}(e_L)$
 462 14: **for** $0 \leq t < T$ **do**
 463 15: $s, c \leftarrow e_L[t]$
 464 16: $Q_L[s, c] \leftarrow \frac{r + Q_L[s, c] \cdot (counts_L[s, c])}{counts_L[s, c] + 1}$ ▷ update avg. reward
 465 17: $counts_L[s, c] \leftarrow counts_L[s, c] + 1$
 466 18: $e_L \leftarrow []$

467 for each MCC learner. We will drop the subscript L when talking
 468 about a single learner. The basic idea is as follows.

469 We are trying to learn a policy π for state space S and choices C .
 470 The policy is ε -greedy: with probability ε it makes random choices
 471 (Line 7), otherwise it makes the choices that will maximize the
 472 value function, Q (Line 9).

473 The value function $Q[s, c]$ models the expected reward at the end
 474 of the episode from the choice c in state s . It is initialized to 0 for
 475 each (s, c) pair (Line 4), so the first episode follows a totally random
 476 policy. $Q[s, c]$ is exactly the *average rewards* seen for each episode
 477 e containing (s, c) . Thus, at the end of each episode e , for each
 478 $(s, c) \in e$ (Line 15), the running average for the rewards observed
 479 with action (s, c) is updated to include the new reward r (Line 16).

480 If the reward function producing r is *stationary* (i.e., fixed), then
 481 it can be shown that this update procedure always improves the
 482 policy. That is, if π is the original policy, and π' is the policy after the
 483 update, the expected reward from a learner following π' is greater
 484 than or equal to the expected reward from a learner following π .
 485 Sutton and Barto [45] provide a proof.

486 We draw attention to some specifics of our implementation, that
 487 diverge from what may appear in textbooks.

488 **4.1.2 Algorithmic Changes.** Firstly, we update an episode with a
 489 single reward r which is distributed to all state action pairs. This is
 490 because, as will be seen in later sections, we only observe rewards
 491 at the end of an episode i.e there are no intermediate rewards
 492 provided in our method. Secondly we do not use a discount factor
 493 on the reward r . This is because the sequence of choices in an input
 494 generation, do not lend themselves to a natural absolute ordering.
 495 We cannot assume later decisions are more important than earlier
 496 ones, which the discount factor implicitly does.

531 4.2 ***RLCheck***: MCC with Diversity Reward

532 We now return to our problem space of generating inputs with a
 533 generator G . Notice that the *guides* we defined in Definition 3.1
 534 have a similar function to the learners in Section 4.1: given some
 535 state (σ, p, t) make a choice c .

536 This leads to the natural idea of implementing a guide as an MCC
 537 learner, rewarding the learner with some $r(x)$ after the generator
 538 produces input x . However, note that for the guide, at each choice
 539 point $p = (\ell, C)$, only a subset of choices $C \subseteq C$ can be taken.
 540 Further, each choice point has a unique task: for example, choosing
 541 whether to generate a left child (Figure 3, Line 9) or a right child
 542 (Figure 3, Line 13). Thus, it is natural to define a separate learner
 543 L_p for each choice point p , and call UPDATE_{L_p} once for each learner
 544 after every execution of the generator.

545 Finally, in Section 3, we defined a guide using a sequence $\sigma \in C^*$
 546 to influence its actions, while in Section 4.1, we assumed a finite
 547 set of states S . Thus, we need a state abstraction function:

548 **Definition 4.1** (State Abstraction Function). A *state abstraction*
 549 function $A : C^* \rightarrow S$ for a generator G is a deterministic function
 550 mapping an arbitrary-length choice sequence σ to a finite state
 551 space S . A can rely on G to retrieve, for any $c_i \in \sigma$, the choice point
 552 p at which c_i was made.

553 We will return to the state abstraction function in Section 4.3.
 554 We can now define a Monte Carlo Control Guide.

555 **Definition 4.2** (Monte Carlo Control Guide). Assume a generator
 556 G producing inputs in \mathcal{X} , a state abstraction function A , and a
 557 reward function $r : \mathcal{X} \rightarrow \mathbb{R}$. A Monte Carlo Control Guide γ
 558 consists of a set of Monte Carlo control *learners*, $\{L_p\}$. Each learner
 559 L_p is associated with a choice point $p = (\ell, C)$ in G .

560 Let $\pi_{L_p}^{(t)}$ be L_p 's policy after $t - 1$ calls to UPDATE_{L_p} (ref. Algo-
 561 rithm 1). Then γ is:

$$562 \gamma(\sigma, p, t) = \pi_{L_p}^{(t)}(A(\sigma)).$$

563 Finally, after G produces an input x , the guide γ calls $\text{UPDATE}_{L_p}(r(x))$
 564 for each of its learners L_p .

565 Now, to use a Monte Carlo Control guide (MCC guide) to solve
 566 the diversifying guidance problem, only (1) the state abstraction
 567 function A (ref. Section 4.3) and (2) the reward function r need to
 568 be specified. We construct a reward function as follows.

569 Let v be the validity function and ξ the characteristic function of
 570 interest. If X be the set of inputs previously generated by G , then
 571 let $\Xi = \{\xi(x') : x' \in X\}$ be the set of characteristics of all the
 572 previously generated inputs. Then the reward function r is:

$$573 r(x) = \begin{cases} R_{\text{unique}} & \text{if } v(x) \wedge \xi(x) \notin \Xi \\ 574 R_{\text{valid}} & \text{if } v(x) \wedge \xi(x) \in \Xi \\ 575 R_{\text{invalid}} & \text{if } \neg v(x) \end{cases} \quad (1)$$

576 Our technique, ***RLCheck***, is thus: make a generator G follow an
 577 MCC Guide with the reward function r above.

578 Note that this reward function is *nonstationary*, that is, it is not
 579 fixed across time. If $X = \emptyset$, then generating any $x \in \mathcal{X}$ such that
 580 $v(x)$ holds will result in the reward R_{unique} ; re-generating the same x
 581 in the next step will only result in the reward R_{valid} . This means the
 582 assumptions underlying the classic proof of policy improvement do
 583

584 not hold [45]. Thus, ***RLCheck***'s guide is not guaranteed to improve
 585 to an optimal policy. Instead, it practices a form of online learning,
 586 adjusting its policy over time.

587 4.3 State Abstraction

588 A key element in enabling MCC to solve the diversifying guidance
 589 problem is the state abstraction function (Definition 4.1), which
 590 determines the current state given a sequence of past choices. The
 591 choice of A impacts the ability of the MCC guide to learn an effective
 592 policy. On one extreme, if A collapses all sequences into the same
 593 abstract state (e.g., $A(\sigma) = 0$), then a learner L_p essentially attempts
 594 to find a single best choice $c \in C_{L_p}$ for choice point p , regardless
 595 of state. On the other extreme, if A is the identity function (i.e.,
 596 $A(\sigma) = \sigma$), then the state space is infinite; so for every previously
 597 unseen sequence of choices σ , the learner's policy is random.

598 The ideal A is the abstraction function that maximizes expected
 599 reward. However, computing such an A is not tractable, since it
 600 requires inverting an arbitrary validity function $v(x)$. Instead, we
 601 apply the following heuristic: in many input generators, a good
 602 representation for the state S_n after making the n^{th} choice c_n is
 603 some function of a past subsequence of choices that *influence* the
 604 choice c_n . The meaning of influence depends on the type of input
 605 being generated and the nature of the validity function.

606 For example, Figure 2 shows a partially generated binary tree
 607 on the left. On the right, we show the choices made in the binary-
 608 tree generator (ref. Fig. 1) leading to this partial tree ($c_1 = 2, c_2 =$
 609 ***True***, $c_3 = 1, c_4 = \text{False}, c_5 = \text{False}, c_6 = \text{True}$), arranged by *influence*,
 610 where a choice in the construction of a child node is *influenced* by
 611 choices constructing its parent node.

612 With this influence heuristic, the best value for the next choice
 613 c_7 , which determines the value assigned to the right child, should
 614 depend on the choice c_1 , which decided that the root node had
 615 value 2, as well as the choice c_6 , which made the decision to insert
 616 a right child. The best value for this choice c_7 does not necessarily
 617 depend on choices $c_2 - c_5$, which were involved in the creation of the
 618 left sub-tree. Therefore, the state S_6 , in which the choice c_7 is to be
 619 made, can be represented as a sequence $[f_v(c_1), f_r(c_6)]$. Here, f_v is
 620 a function associated c_1 's choice point (the node-value choice point
 621 at Line 4 of Fig. 1) and f_r is a function associated with c_6 's choice
 622 point (the right-child choice point at Line 10 of Fig. 1). In Figure 2,
 623 the state S_6 after applying these functions is $[2, R]$; we will define
 624 the functions f_v and f_r for this figure later in this section.

625 An additional consideration when representing state as a se-
 626 quence derived from past choices is that such sequences can be-
 627 come very long. We need to restrict the state space to being finite.
 628 Again, a reasonable heuristic is to use a trimmed representation of
 629 the sequence, which incorporates information from up to the last w
 630 choices that influence the current choice. w is a fixed integer that
 631 determines the size of a sliding window.

632 We can build a state abstraction function that follows these con-
 633 siderations in the following manner. First, build a choice abstraction
 634 function f_p for each choice point p , which maps each c to an abstract
 635 choice. Then, for $\sigma = c_1, c_2, \dots, c_n$, build $S_n = A(\sigma)$ so that:

$$636 S_n = \begin{cases} \emptyset & \text{if } \sigma = \emptyset \\ 637 \text{tail}_w(S_k :: f_p(c_n)) & \text{for some } k < n \text{ otherwise,} \end{cases}$$

$S_1 = 2$	$S_5 = 2$	$c_1 = 2$
$S_1 = 2$	$S_5 = 2$	$c_2 = \text{True}$
$S_2 = 2, L$		$c_6 = \text{True}$
	$S_6 = 2, R$	
$c_3 = 1$		$c_7 = ?$
$S_3 = 2, L, 1$	$S_4 = 2, L, 1$	
$c_4 = \text{False}$	$c_5 = \text{False}$	

Figure 2: A partially-generated binary tree (left) and its corresponding choice sequence arranged by influence (right).

```
1 def concat_tail(state, value):
2     return (state + [value])[-WINDOW_SIZE:]
3
4 def gen_tree(state, depth=0):
5     value = guide.Select([0, ..., 10], state, idx=1)
6     state = concat_tail(state, value)
7     tree = BinaryTree(value)
8     if depth < MAX_DEPTH and \
9         guide.Select([True, False], state, idx=2):
10        left_state = concat_tail(state, "L")
11        tree.left = gen_tree(left_state, depth+1)
12    if depth < MAX_DEPTH and \
13        guide.Select([True, False], state, idx=3):
14        right_state = concat_tail(state, "R")
15        tree.right = gen_tree(right_state, depth+1)
16
17 return tree
```

Figure 3: Pseudo-code for a binary tree generator which follows guide and builds a tree-based state abstraction.

where $::$ is the concatenation operator and $\text{tail}_w(s)$ takes the last w elements of s . Assume c_n was taken at choice point p .

We can build both very basic and very complex state abstractions in this manner.

For example, we can get $A(\sigma) = c_{n-w+1}, \dots, c_{n-1}, c_n$ by taking $f_p = id$ for all and choosing $k = n - 1$ always. This would be a simple sliding window of the last w choices.

The states S_1 - S_6 that annotate the edges in Figure 2 are derived using the choice point abstraction functions $f_v(c) = c$ for the value choice point, $f_r(c) = R$ for the right child choice point, and $f_l(x) = L$ for the left child choice point. The k is chosen as $k = \text{"largest } k < n \text{ which is a choice from the parent node"}$. While programmatically deriving this k from a choice sequence σ is tedious, it is quite easy to do inline in the generator. The generator Figure 3 shows a modified version of the generator from Figure 1, which updates an explicit state value at each to compute exactly this state abstraction function (Lines 6, 9, 13); it also uses guides to select arbitrary values (Lines 5, 9, 13).

4.3.1 Case study. We evaluate the effect the state abstraction function has on the ability of *RLCheck* to produce unique valid inputs for the BST example. We evaluate three state abstraction functions:

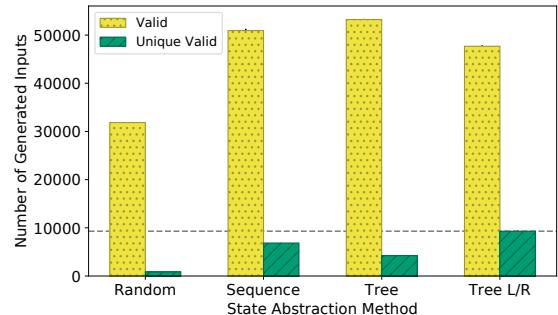


Figure 4: Number of (unique) valid inputs generated, by state abstraction. “Random” is a no-RL baseline.

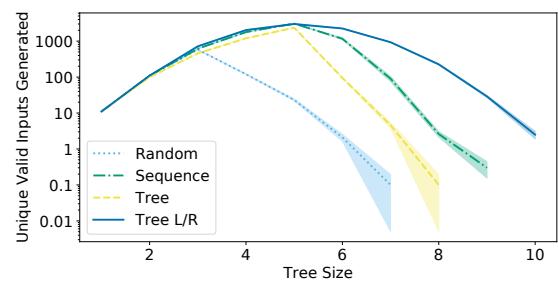


Figure 5: Distribution of unique valid tree sizes, by state abstraction. “Random” is a no-RL baseline.

- *Sequence*, the sliding window abstraction which retains choices from the sibling nodes, i.e. $A(\sigma) = c_{n-w+1}, \dots, c_{n-1}, c_n$.
 - *Tree L/R*, the abstraction function illustrated in Figure 2 and implemented in Figure 3.
 - *Tree*, which chooses k like Tree L/R but has $f_p = id$ for all choice points, and thus produces the same state for the left and right subtree of a node.

For example, taking $w = 4$ and the choices to be abstracted c_1, \dots, c_6 from Figure 2: *Sequence* will give `[1, False, False, True]`, *Tree* state will give `[2, True]`, and *Tree L/R* will give `[2, "R"]`.

We evaluate each of these abstraction techniques for generating BSTs with maximum depth 4 (i.e., 4 links), with $\epsilon = 0.25$ and rewards (Eq. 1) $R_{invalid} = -1$, $R_{valid} = 0$, and $R_{unique} = 20$. We set $w = 4$ for the abstraction function: since there are at least two elements in the state for each parent node, this means the learners cannot simply memorize the decisions for the full depth of the tree.

Results. Figures 4 and 5 show the results for our experiments. In each experiment we let each technique generate 100,000 trees. The results show the averages and standard errors over 10 trials. We compare to a baseline, *Random*, which just runs the generator from Figure 1. Figure 4 illustrates that no matter the state abstraction function chosen, *RLCheck* generates many more valid and unique valid inputs than the random baseline; *Tree L/R* generates 10× more unique valid inputs than random. Within the abstraction techniques, *Tree* generates the fewest unique valid inputs. *Sequence* appears to

be better able to distinguish whether it is generating a left or right child than *Tree*, probably because the *Tree* state is identical for the left and right child choice points.

Tree L/R generates the fewest valid inputs, but the most unique valid inputs, 36% more than *Sequence*. These unique valid inputs are also more complex those generated with other state abstractions. Figure 5 shows, for each technique, the average number of unique valid trees generated of each size. Note the log scale. The tree size is the number of nodes in the tree. We see that *Tree L/R* is consistently able to generate orders of magnitude more trees of sizes > 5 than the other techniques. Since we reward uniqueness, the *RLCheck* is encouraged to generate larger trees as it exhausts the space of smaller trees. These results suggest that *Tree L/R* has enough information to generate valid trees, and then combine these successes into more unique valid trees.

Overall, we see that even with a naïve state abstraction function, *RLCheck* generates nearly an order of magnitude more unique valid inputs than the random baseline. However, a well-constructed influence-style state abstraction yields more diverse valid inputs.

5 EVALUATION

In this section we evaluate how *RLCheck*, our MCC-based solution to the diversifying guidance problem, performs. In particular, we focus on the following research questions:

- RQ1 Does *RLCheck* quickly find many diverse valid inputs for real-world benchmarks compared to state-of-the-art?
- RQ2 Does *RLCheck* find valid inputs covering many different behaviors for real-world benchmarks?
- RQ3 Does adding coverage feedback improve the ability of *RLCheck* to generate diverse valid inputs for real-world benchmarks?

Implementation. To answer these research questions, we implemented Algorithm 1 in Java, and *RLCheck* on top of the open-source JQF [37] platform. JQF provides a mechanism for customizing input generation for QuickCheck-style property tests.

Baseline Techniques. We compare *RLCheck* to two different methods: (1) junit-quickcheck [26], or simply QuickCheck, the baseline generator-based testing technique which calls the generator with a randomized guide; and (2) Zest [38], also built on top of JQF, which uses an evolutionary algorithm based on coverage and validity feedback to “guide” input generators. Unlike *RLCheck* and QuickCheck, Zest is a *greybox* technique: it relies on program instrumentation to get code coverage from each test execution.

Benchmarks. We compare the techniques on four real-world Java benchmarks used in the original evaluation of Zest [38]: Apache Ant, Apache Maven, Google Closure Compiler, and Mozilla Rhino. These benchmarks rely on two generators: Ant and Maven use an XML generator, whereas Closure and Rhino use a generator for JavaScript ASTs. The validity functions for each of these four benchmarks is distinct: Ant expects a valid build.xml configuration, Maven expects a valid pom.xml configuration, the Closure expects an ES6-compliant JavaScript program that can be optimized, and Rhino expects a JavaScript program that can be statically translated to Java bytecode. Overall, Ant has the strictest validity function and Rhino has the least strict validity function.

Design Choices. In our main evaluation, we simply use identity as the characteristic function ξ to which inputs get R_{unique} . Thus, *RLCheck* simply tries to maximize the number of unique valid inputs. This allows us to run *RLCheck* at full speed without instrumentation, and generate more inputs in a fixed time budget. In Section 5.3 we compare this choice to a greybox version of *RLCheck*, where $\xi(x)$ takes into account the branch coverage achieved by input x .

We instantiate our reward function (Eq. 1) with $R_{unique} = 20$, $R_{valid} = 0$ and $R_{invalid} = -1$. This incentivizes *RLCheck* to prioritize exploration of new unique valid inputs, while penalizing strategies that lead to producing invalid inputs. Additionally, we set $\varepsilon = 0.25$ in Algorithm 1, which allows *RLCheck* to explore at random with reasonably high probability.

We first modified the base generators provided by JQF for XML and JavaScript to transform choice points with infinite domains to finite domains. These are the generators we use for evaluation of Zest and QuickCheck. We then built guide-based generators with the same choice points as these base generators. For the guide-based generators, we built the state abstraction inline, like it is built in Figure 3. For each benchmark, the state abstraction function is similar to that in Figure 3 as it maintains abstractions of the parent choices. We set $w = 5$ for the state window size. The code for the generators is available as supplementary material for review.

Experiments. We sought to answer our research questions in a property-based testing context, where we expect to be able to run the test generator for a short amount of time. Thus, we chose 5 minutes as a timeout. To account for variability in the results, we ran 10 trials for each technique. The experiments in Section 5.1 and 5.2 were run on GCP Compute Engine using a single VM instance with 8vCPUs and 30 GB RAM. The experiments in Section 5.3 were run on a machine with 16GB RAM and an AMD Ryzen 7 1700 CPU.

5.1 Generating Diverse Valid Inputs

To answer RQ1, we need to measure whether *RLCheck* generates a higher number of unique, valid inputs compared to our baselines. On these large-scale benchmarks, where the test driver does non-trivial work, simple uniqueness, at the byte or string level, is not the most relevant measure of input diversity.

What we are interested in is inputs with diverse coverage. So, we measure inputs with different *traces*, a commonly-used metric for input coverage diversity in the fuzz testing literature [49] (sometimes these traces are called “paths”, but this is a misnomer). The trace of an input x is a set of pairs (b, c) where b is a branch and c is the number of times that branches is executed by x , bucketed to base-2 orders of magnitude. Let $\xi(x)$ give the trace of x . If x_1 takes the path A, B, A , then $\xi(x_1) = \{(A, 2), (B, 1)\}$. If x_2 takes the path A, A, A, B , then A is hit the same base-2 order-of-magnitude times, so $\xi(x_2) = \{(A, 2), (B, 1)\}$. We call valid inputs with different traces *diverse valid* inputs.

The results are shown in Figures 6 and 7. Figure 6 shows, at each time, the *percentage* of all generated inputs that are diverse valid inputs. For techniques that are only able to generate a fixed number of diverse valid inputs, this percentage would steadily decrease over time. In Figures 6c and 6d, we see an abrupt decrease at the beginning of fuzzing for Zest and QuickCheck, and for Closure we see a continuing decrease in the percentage over time for these

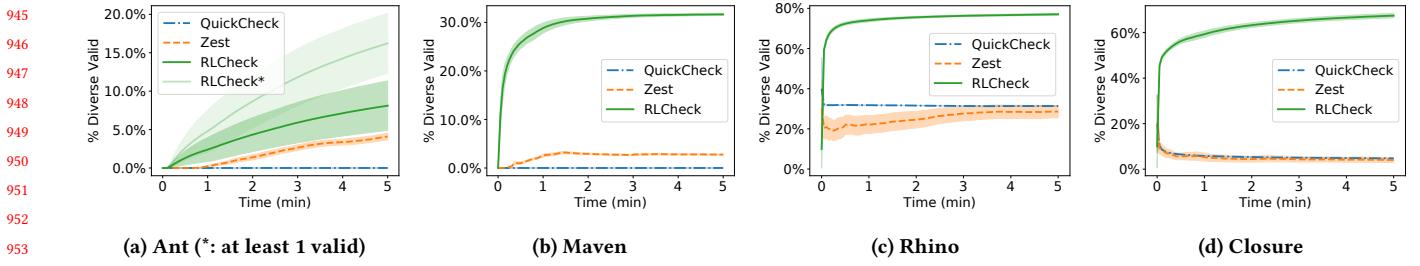


Figure 6: Percent of total generated inputs which are diverse valids (i.e. have different traces). Higher is better.

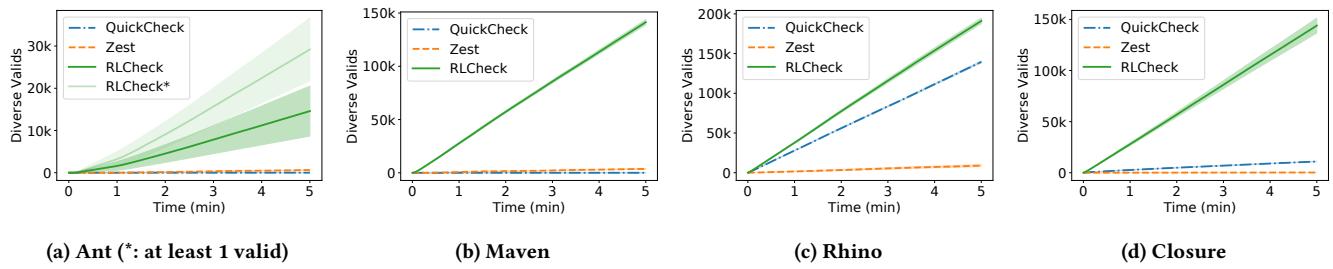


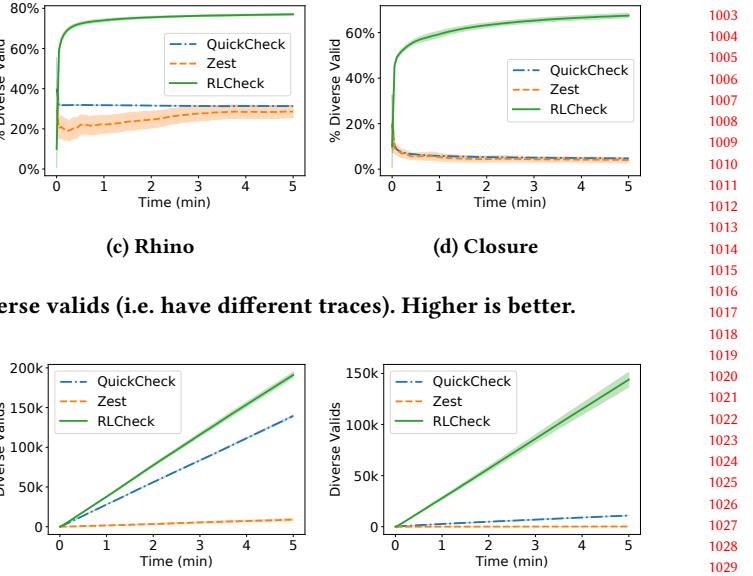
Figure 7: Number of diverse valid inputs (i.e. inputs with different traces) generated by each technique. Higher is better.

techniques. In Figures 6b, 6c, and 6d see that *RLCheck* quickly converges to a high percentage of diverse valid inputs being generated, and maintains this until timeout.

RLCheck also generates a large *quantity* of diverse valid inputs. Figure 7 shows the total number of diverse valid inputs generated by each technique: we see that *RLCheck* generates multiple order of magnitude more diverse valid inputs compared to our baselines. The exception is on Rhino (Figure 7), *RLCheck* only has a 1.4× increase over QuickCheck. Rhino's validity function is relatively easy to satisfy: most JavaScript ASTs are considered valid inputs for translation; therefore, speed is the main factor in generating valid inputs for this benchmark. Consequently, the blackbox techniques *RLCheck* and QuickCheck outperform the instrumentation-based Zest technique on the Rhino benchmark.

On both metrics, the increase in Ant is less pronounced, and very variable. The variation in percentage for Ant is quite wide because it is hard to get a first valid input for *RLCheck* (and QuickCheck), and in some cases *RLCheck* did not get this within the five-minute time budget. For an understanding of the effect on the results, *RLCheck** shows the results for only those runs that find at least one valid input. The mean value for *RLCheck** is much higher, but the high standard errors remain due to the fact that these runs find the first valid input being at different times. For such extremely strict validity functions, *RLCheck* has difficulty finding a first valid input compared to coverage-guided techniques. This is a limitation of *RLCheck*: a good policy can only be found after some valid inputs have been discovered.

For completeness, we also ran longer experiments of 1 hour, to see if Zest or QuickCheck would catch up to *RLCheck*. In 1 hour, *RLCheck* generates between 5–15× more diverse valid inputs than Zest on all benchmarks and outperforms QuickCheck on all benchmarks. Furthermore, *RLCheck* continues to generate a higher



percentage of generated diverse valid inputs after one hour. In particular, the large improvements that are seen in Figures 6 are all maintained at roughly the same rate except for Rhino. In the case of Rhino, Zest improves its percentage of diverse valid inputs from 40% to 67% after one hour, while *RLCheck* continues to generate 78% diverse valid inputs throughout.

RQ1: *RLCheck* quickly converges to generating a high percentage of diverse valid inputs, and on most benchmarks generates orders of magnitude more diverse valid inputs than our baselines.

5.2 Covering Different Valid Behaviors

Section 5.1 shows that *RLCheck* generates many more diverse valid inputs than the baselines, i.e. solves the diversifying guidance problem more effectively. A natural question is whether the valid inputs generated by each method cover the same set of input behaviors (RQ2). For this, we can compare the cumulative branch coverage achieved by the valid inputs generated by each technique.

Figure 8 shows the coverage achieved by all valid inputs for each benchmark until timeout. The results are much more mixed than the results in Section 5.1. On the Closure benchmark (Fig. 8d), QuickCheck and *RLCheck* achieve the same amount of branch coverage by valid inputs. On Rhino (Fig. 8c) QuickCheck dominates slightly. On Maven (Fig. 8b), *RLCheck* takes an early lead in coverage but Zest's coverage-guided algorithm surpasses it at timeout.

On Ant (Figure 8a), *RLCheck* appears to perform poorly, but this is mostly an artifact of *RLCheck*'s bad luck in finding a first valid input. Again, for comparison's sake, *RLCheck** shows the results for only those runs that generate valid inputs: we see that *RLCheck*'s branch coverage is slightly above Zest's on these runs.

1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081

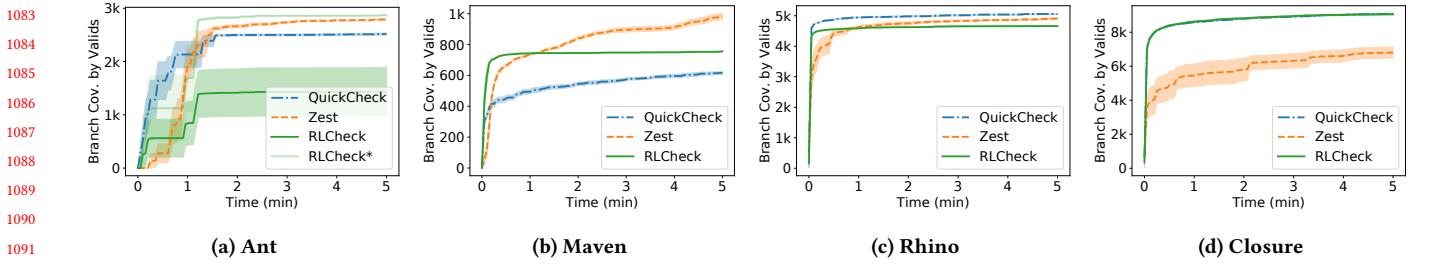


Figure 8: Number of branches covered by valid inputs generated by each technique. Higher is better

The overall clearest trend from Figure 8 is that *RLCheck*'s branch coverage seems to quickly peak and then flatten compared to the other techniques. This suggests that our MCC-based algorithm, while it is exploring diverse valid inputs, may still be tuned too much towards exploiting the knowledge from the first set of valid inputs it generates. We discuss in Section 6 some possible avenues to explore in terms of the RL algorithm.

RQ2: No method achieves the highest branch coverage on all benchmarks. *RLCheck*'s plateauing branch coverage suggests that it may be learning to generate diverse inputs with similar features rather than discovering new behavior.

5.3 Greybox Information

Given that *RLCheck* is able to attain its objective as defined by the diversifying guidance problem—generating large numbers of unique valid inputs—(Section 5.1), but does not achieve the highest branch coverage over all benchmarks (Section 5.2), a natural question is to ask whether choosing a different ξ , one that is coverage-aware, could help increase the diversity of behaviors discovered. This is what we seek to answer in RQ3.

For this experiment, we re-ran *RLCheck* both blackbox, i.e. with $\xi_{bb} = id$, and with greybox information, using $\xi_{gb}(x) = \text{"the set of all branches covered by the input } x\text{"}$. Thus, Greybox *RLCheck* is rewarded when it discovers a valid input that covers a distinct set of branches compared to all generated inputs. Note that this does not reward the guide more for generating an input which covers a wholly-uncovered branch, compared to an input that covers a new combination of already-seen branches. Again, we ran each method for 10 trials, timing out at 5 minutes.

Figures 9 shows the number of diverse valid inputs (valid inputs with distinct traces) generated the the blackbox and greybox versions of *RLCheck*, and Figure 10 shows the branch coverage by valid inputs for these two versions. We see universally across all benchmarks and both metrics that Blackbox *RLCheck* outperforms Greybox *RLCheck*. This suggests that the slowdown incurred by instrumentation is not worth the increased information *RLCheck* gets in the greybox setting. The difference is less striking for branch coverage than number of diverse valid inputs generated, because fewer inputs are required to get the same cumulative branch coverage.

We see much lower variation in Ant in this experiment because on all 10 runs, Blackbox *RLCheck* was able to generate at least one valid input for Ant. We chose random seeds at random in both experiments, so this is simply a quirk of experimentation.

RQ3: Adding greybox feedback to *RLCheck* in terms of the characteristic function ξ causes a large slowdown, but no huge gains in number of valid inputs or coverage achieved. Overall, *RLCheck* performs best as a black-box technique.

6 DISCUSSION

Performance. Tabular methods such as ours do not scale well for large choice or state spaces. Let S and C denote state and choice space sizes, respectively. The Monte Carlo control algorithm requires $O(SC)$ space overall, and requires $O(C)$ time for evaluating the policy function π . This is because all the algorithmic decision-making is backed by a large Q -table with $S \times C$ entries. Because of these constraints we had to restrict our state and choice spaces for *RLCheck*. For example, in our JavaScript implementation, when selecting integer values, we restricted our choice space to the range $\{0, 1, \dots, 10\}$ rather than a larger range like $\{0, 1, \dots, 100\}$. This was necessary to generate inputs in a reasonable amount of time with this generator. Function approximation methods, such as replacing the Q -table with a neural network, may be necessary for dealing with larger, more complex, state and choice spaces.

Increasing Exploration. In Section 5.2 we observed that the branch coverage achieved by *RLCheck*-generated valid inputs tends to quickly plateau, even for benchmarks where the other methods could achieve higher branch coverage (Figs 8b, 8c). This suggests that even with a high ϵ , MCC may still be too exploitative for the diversifying guidance problem. One approach to increase exploration would be to allow the learners to “forget” old episodes so choices made early in the testing session that are not necessary to input validity do not persist throughout the session. Curiosity-based approaches, which strongly encourage exploration and avoid revisiting states [40], may also be applicable.

Fine-tuning. In our experiments we heuristically chose the ϵ and k values, and then kept them fixed across benchmarks. We noted the importance of a large ϵ and modest k value for both generating unique valid inputs and doing so quickly. We also chose the reward function heuristically and in our design process we noticed how that this choice significantly affected performance, and particularly the distribution of invalid, valid, and unique valid inputs generated. It may be valuable to fine tune these hyperparameters and reward functions for different benchmarks.

Bootstrapping. In Section 5.1 we saw that *RLCheck* had difficulty generating a first valid input for very strict validity functions (Ant).

1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220

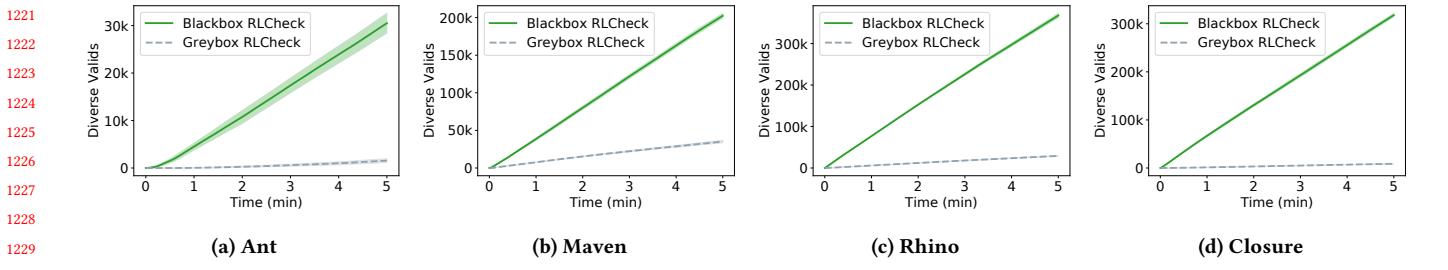


Figure 9: Number of diverse valid inputs generated by each technique. Higher is better.

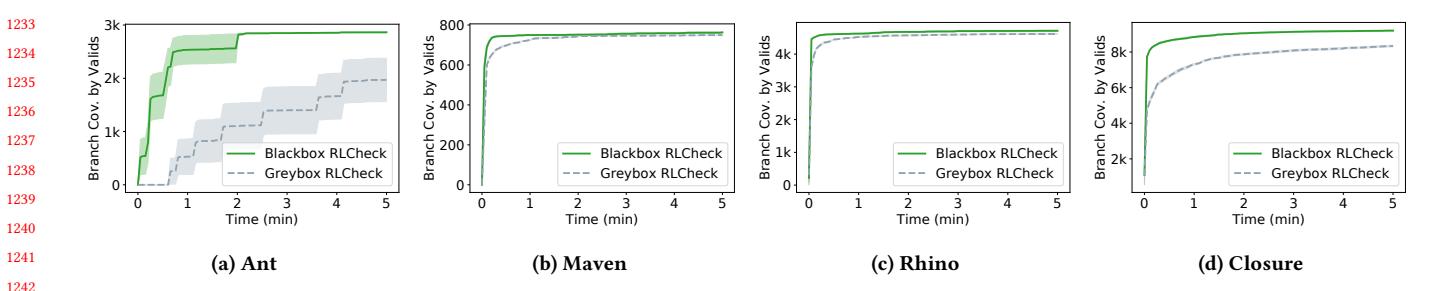


Figure 10: Number of branches covered by valid inputs generated by each technique. Higher is better.

This limitation could be overcome by allowing *RLCheck* to be *bootstrapped*, i.e. given a sequence of choices that produces a valid input at the beginning of testing. This choice sequence could be user-provided, as long as there exists a relatively short sequence of choices resulting in a valid input.

7 RELATED WORK

The problem of automatically generating test inputs that satisfy some criteria has been studied for over four decades. Symbolic execution [17, 27] as well as its dynamic [14] and concolic [23, 43] variants attempt to generate test inputs that reach program points of interest by collecting and solving symbolic constraints on inputs. Despite numerous advances in improving the precision and performance of these techniques [10, 11, 21, 28, 44], the path explosion problem [15] remains a key challenge that limits scalability to large complex constraints.

Fuzz testing [35] is a popular method to generate byte-sequence test inputs. The key idea is to generate a huge quantity of test inputs at random, without incurring much cost for each individual input. Input validity requirements can be addressed either via user-provided input format specifications [5] or by mutating existing valid inputs [6]. Coverage-guided fuzzing, popularized by tools such as AFL [48], improves the effectiveness of mutation-based fuzz testing by instrumenting the program under test and incorporating feedback in the form of code coverage achieved by each test execution; the feedback is used to perform an evolutionary search for test inputs that cover various behaviors in the test program.

Search-based software testing [24, 25, 33, 47] generates inputs which optimize some objective function by using optimization techniques such as hill-climbing or simulated annealing. These techniques work well when the objective function is a smooth function of the input characteristics.

QuickCheck [16] introduced the idea of formulating tests as properties $\forall x : P(x) \Rightarrow Q(x)$, which could be validated to some degree by randomly generating many instances of x that satisfy $P(x)$. Of course, the main challenge is in ensuring that $P(x)$ is satisfied often enough. Some researchers have attempted to write generators that produce diverse valid inputs by construction, such as for testing compilers [34, 46], but solutions turn out to be highly complex domain-specific implementations. For some domains, effective generators can also be automatically synthesized [31, 36]. Targeted property-based testing [32] biases hand-written input generators for numeric utility values. Domain-specific languages such as Luck [29] enable strong coupling of generators and validity predicates. In contrast, we address the problem of biasing arbitrary input generators towards producing inputs that satisfy arbitrary validity functions, without any prior domain knowledge.

Recently, techniques such as Zest [38], Crowbar [19], and FuzzChick [30] have combined ideas from coverage-guided mutation-based fuzzing with generative techniques such as QuickCheck. Although code coverage guidance helps discover new program behaviors, it comes at the cost of program instrumentation, which significantly reduces the number of test inputs that can be executed per unit time. In contrast, we address the problem of generating valid test inputs when considering the program as a black box.

There has been increasing interest in using machine learning to improve fuzz testing; Saavedra et al. [42] provide a survey. Böttlinger et al. [12] propose a deep reinforcement learning approach to fuzz testing. This work uses the reinforcement learning agent to, given a subsequence of the input file as state, perform a mutation action on that subsequence. Instead of learning to mutate serialized input strings directly, *RLCheck* employs reinforcement learning on generators for highly structured inputs.

1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357

1359 8 CONCLUSION

1360 In this paper we investigated a reinforcement learning approach to
 1361 guiding input generators to generate more valid inputs for property-
 1362 based testing. We began by formalizing the problem of generating
 1363 many unique valid inputs for property-based testing as the diversify-
 1364 ing guidance problem. We proposed *RLCheck*, where generators
 1365 follow a Monte Carlo Control (MCC)-based guide to generate in-
 1366 puts. We found that *RLCheck* has great performance in terms of
 1367 generating many diverse valid inputs on real-world benchmarks.
 1368 However, MCC seems to be prone to overfitting to a certain space of
 1369 valid inputs. We believe more exploration-oriented RL approaches
 1370 could be better suited to provide the guidance in *RLCheck*.
 1371

1372 REFERENCES

- [1] 2019. Eris: Porting of QuickCheck to PHP. <https://github.com/giorgiosironi/eris>. Accessed January 28, 2019.
- [2] 2019. FsCheck: Random testing for .NET. <https://hypothesis.works/>. Accessed January 28, 2019.
- [3] 2019. Hypothesis for Python. <https://hypothesis.works/>. Accessed January 28, 2019.
- [4] 2019. JSVerify: Property-based testing for JavaScript. <https://github.com/jsverify/jsverify>. Accessed January 28, 2019.
- [5] 2019. PeachFuzzer. <https://www.peach.tech>. Accessed August 21, 2019.
- [6] 2019. Radamsa: a general-purpose fuzzer. <https://gitlab.com/akihe/radamsa>. Accessed August 21, 2019.
- [7] 2019. ScalaCheck: Property-based testing for Scala. <https://www.scalacheck.org/>. Accessed January 28, 2019.
- [8] 2019. test.check: QuickCheck for Clojure. <https://github.com/clojure/test.check>. Accessed January 28, 2019.
- [9] Cláudio Amaral, Mário Florido, and Vítor Santos Costa. 2014. PrologCheck—property-based testing in Prolog. In *International Symposium on Functional and Logic Programming*. Springer, 1–17.
- [10] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 367–381.
- [11] 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 (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094.
- [12] Konstantin Böttiger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. *CoRR* abs/1801.04589 (2018). arXiv:1801.04589 <http://arxiv.org/abs/1801.04589>
- [13] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 123–133. <https://doi.org/10.1145/566172.566191>
- [14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*.
- [15] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
- [16] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.
- [17] Lori A. Clarke. 1976. A program testing system. In *Proc. of the 1976 annual conference*. 488–491.
- [18] David Coppit and Jixin Lian. 2005. Yagg: An Easy-to-use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 356–359. <https://doi.org/10.1145/1101908.1101969>
- [19] Stephen Dolan. 2017. Property fuzzing for OCaml. <https://github.com/stedolan/crowbar>. Accessed Jul 23, 2019.
- [20] Roy Emek, Itai Jaeger, Yehuda Naveh, Gadi Bergman, Guy Aloni, Yoav Katz, Monica Farkash, Igor Dozoretz, and Alex Goldin. 2002. X-Gen: A random test-case generator for systems and SoCs. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*. IEEE, 145–150.
- [21] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDTA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 225–234. <https://doi.org/10.1145/1806799.1806835>
- [22] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*.
- [23] 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 (PLDI '05)*.
- [24] Mark Harman. 2007. The current state and future of search based software engineering. In *2007 Future of Software Engineering*. IEEE Computer Society, 342–357.
- [25] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.
- [26] Paul Holser. 2014. junit-quickcheck: Property-based testing, JUnit-style. <https://pholser.github.io/junit-quickcheck>. Accessed August 21, 2019.
- [27] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19 (July 1976), 385–394. Issue 7.
- [28] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Acm Sigplan Notices*, Vol. 47. ACM, 193–204.
- [29] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hrițu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 114–129. <https://doi.org/10.1145/3009837.3009868>
- [30] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. [n.d.]. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 2, OOPSLA ([n. d.]).
- [31] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proc. ACM Program. Lang.* 2, POPL, Article 45 (Dec. 2017), 30 pages. <https://doi.org/10.1145/3158133>
- [32] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 46–56. <https://doi.org/10.1145/3092703.3092711>
- [33] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, Washington, DC, USA, 153–163. <https://doi.org/10.1109/ICSTW.2011.100>
- [34] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP (2017). <http://doi.acm.org/10.1145/3110259>
- [35] Barton P. Miller, Louis Frederiks, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. <https://doi.org/10.1145/96267.96279>
- [36] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching Processes for QuickCheck Generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 1–13. <https://doi.org/10.1145/3242744.3242747>
- [37] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 398–401. <https://doi.org/10.1145/3293882.3339002>
- [38] 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 (ISSTA '19)*. <https://doi.org/10.1145/3293882.3330576>
- [39] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr Integration of Types and Function Specifications with Property-based Testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang (Erlang '11)*. ACM, New York, NY, USA, 39–50. <https://doi.org/10.1145/2034654.2034663>
- [40] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. 2017. Curiosity-driven Exploration by Self-supervised Prediction. In *ICML*.
- [41] Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2017. A Solver-aided Language for Test Input Generation. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 91 (Oct. 2017), 24 pages. <https://doi.org/10.1145/3133915>
- [42] Gary J. Saavedra, Kathryn N. Rodhouse, Daniel M. Dunlavy, and W. Philip Kegelmeyer. 2019. A Review of Machine Learning Applications in Fuzzing. *CoRR* abs/1906.11133 (2019). arXiv:1906.11133 <http://arxiv.org/abs/1906.11133>
- [43] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*.
- [44] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 842–853.
- [45] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press. <http://www.incompleteideas.net/book/ebook/node53.html>
- [46] 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 (PLDI '11)*.

- 1497 [47] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case
1498 selection. In *Proceedings of the 2007 international symposium on Software testing
1499 and analysis*. ACM, 140–150.
- 1500
- 1501
- 1502
- 1503
- 1504
- 1505
- 1506
- 1507
- 1508
- 1509
- 1510
- 1511
- 1512
- 1513
- 1514
- 1515
- 1516
- 1517
- 1518
- 1519
- 1520
- 1521
- 1522
- 1523
- 1524
- 1525
- 1526
- 1527
- 1528
- 1529
- 1530
- 1531
- 1532
- 1533
- 1534
- 1535
- 1536
- 1537
- 1538
- 1539
- 1540
- 1541
- 1542
- 1543
- 1544
- 1545
- 1546
- 1547
- 1548
- 1549
- 1550
- 1551
- 1552
- 1553
- 1554
- [48] Michał Zalewski. 2014. American Fuzzy Lop. <http://lcamtuf.coredump.cx/afl>. Accessed January 11, 2019.
- [49] Michał Zalewski. 2014. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed Aug 2019.
- 1555
- 1556
- 1557
- 1558
- 1559
- 1560
- 1561
- 1562
- 1563
- 1564
- 1565
- 1566
- 1567
- 1568
- 1569
- 1570
- 1571
- 1572
- 1573
- 1574
- 1575
- 1576
- 1577
- 1578
- 1579
- 1580
- 1581
- 1582
- 1583
- 1584
- 1585
- 1586
- 1587
- 1588
- 1589
- 1590
- 1591
- 1592
- 1593
- 1594
- 1595
- 1596
- 1597
- 1598
- 1599
- 1600
- 1601
- 1602
- 1603
- 1604
- 1605
- 1606
- 1607
- 1608
- 1609
- 1610
- 1611
- 1612
- 1613
- 1614
- 1615
- 1616
- 1617
- 1618
- 1619
- 1620
- 1621
- 1622
- 1623
- 1624
- 1625
- 1626
- 1627
- 1628
- 1629
- 1630
- 1631
- 1632
- 1633
- 1634