Guided Random Testing Paper Evaluation

Aditya Akhileshwaran, Rich Chen, Edward Qin*
{aakhiles,rc2002,edwardcq}@cs.washington.edu
University of Washington
Seattle, WA, USA

ABSTRACT

Guided Random Testing (GRT) [5] is a set of techniques that enhances traditional random testing methods for automated test generation. The GRT techniques accomplish this by incorporating heuristics and feedback via a mix of static and dynamic analyses to improve efficiency and effectiveness. Though original authors Lei et al [5] found that their GRT ideas were effective, the underlying implementation was not released publicly. In this paper, we evaluate the strengths and limitations of GRT through a best attempt at a replication study of the results from the original paper, with the additional goal of releasing an open implementation of the techniques. To that end, we implement a subset of the GRT techniques on the pre-existing random testing generation tool Randoop [6]. Then, we conduct a replicated evaluation of the GRT techniques, following similar methods from the GRT paper. Our study includes a detailed examination of the implementation strategies, performance trade-offs, and case studies demonstrating its impact.

1 INTRODUCTION

Unit tests for object-oriented programming consist of a sequence of method calls involving object construction and mutation, culminating in an assertion about the correctness of the results. Writing unit tests can be tedious, time-consuming, and challenging, especially when aiming to obtain high levels of code coverage. This often discourages developers from creating an extensive testing suite, eventually increasing the risk of adding code with defects into production code. Such defects usually arise due to oversight in development or from not anticipating edge cases in the code.

Prior tools like Randoop [6] and EvoSuite [1] attempt to address the issue of automatic unit test generation. Guided Random Testing (GRT) [5] builds on Randoop and proposes a technique using static and dynamic information on program types, data, and dependencies in various stages of automated test generation. The GRT paper reports better performance than Randoop and EvoSuite in its evaluation, and also beat EvoSuite in the 2015 Search-Based Software Testing (SBST) competition [2]. This is surprising given EvoSuite's consistent leading performance, scoring first place at the SBST competition as recently as 2022 [7].

Much value for the research community can be derived from a systematic study of GRT. Unfortunately, the authors of the GRT paper were reluctant to share their implementation and experiments. As such, programmers are unable to use GRT-based tools in practice. Our implementation and evaluation of these techniques would be valuable to the research community, both with regards to providing verification for and deeper understanding of the GRT papers' results, and also providing an accessible, open-source implementation of GRT to end user programmers. Our study achieves this through a replication of the GRT paper's results: we provide an implementation of techniques and uncover deeper relationships between the techniques for defect detection, code coverage, and smutation score metrics.

2 RELATED WORK

Current techniques for test generation include random unit test generation like that of Randoop and GRT, and evolutionary search like EvoSuite.

2.1 Randoop

To our knowledge, Randoop [6] was one of the first test generation techniques to demonstrate the effectiveness of random test generation. Randoop centers around test generation for object-oriented programming languages, incorporating feedback from executed test inputs to better inform subsequent test inputs.

At its core, Randoop works with *method sequences* representing a sequence of assignment statements or method calls. Input arguments to method calls include primitive values and reference values. These values live in a pool of (or set of) possible input values, indexed on the type of the value. For example, there may be a pool of `int` values, a pool of `String` values, and a pool of `ClassA` values where `ClassA` is a reference type. For reference types, the pool also remembers the sequence required to construct the object.

Sequences of method calls are then extended by choosing a random method under test. Randomly chosen input sequences for values v_i for types T_i where $i \in [k]$ are selected. The method call $m(T_1, \ldots, T_k)$ is then appended at the end of the sequence. The resulting sequence is then checked for redundancy, legality, errors, and usefulness for new inputs. If the input has no violations of legality, a regression test can be constructed with asserts on object states. Otherwise, an error-revealing test can be produced to indicate a defect in the program. Usefulness of constructed objects for new inputs is also determined by configurable *filters*, for which default heuristics include adding inputs that do not have the same abstract value in the pool, are not null, and do not produce exceptions.

Consider the snippet of an example program in Figure 1. The program offers a `ListNoisifier` that creates a new list with added Gaussian noise to the elements, and an `ArrayListWrapper` that wraps

*All authors contributed equally to this research.

CSE 503 Final Project, Winter 2025, Seattle, WA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

class ListNoisifier {
  static final int MAX_SIZE = 1024;
  static final long DEFAULT_SEED = 42;

  static List<Double> createList(List<Double> list, Random r) {
    List<Double> result = new ArrayList<>(MAX_SIZE);
    for (int i = 0; i < Math.min(list.size(), MAX_SIZE); i++) {
      result.add(list.get(i) + r.nextGaussian());
    }
    return result;
  }

  static List<Double> createList(List<Double> list) {
    return createList(list, new Random(DEFAULT_SEED));
  }
}

class ArrayListWrapper {
  static void ensureCapacity(ArrayList<?> list, int size) {
    foo();
    list.ensureCapacity(size);
  }
  ...
}
...
```

Figure 1: Example methods that Randoop or GRT could generate tests for

the `ensureCapacity` library function specific to `ArrayList` implementations. When running Randoop, method sequences consisting of `List` construction and method calls may produce new `List` values. These are added into the input pool under the corresponding type. In each iteration, Randoop selects a random method to perform sequence extensions on. For example, if `ListNoisifier.createList(List<Double>)` is selected, then a corresponding `List<Double>` value and its method sequence is randomly selected from the input pool. The new sequence returned is the result of the `ListNoisifier.createList(List<Double>)` called and appended as a new statement at the end of the sequence. The resulting sequence can be added as a new input, and since there are no errors, a unit test can also be generated based on asserts of the object states.

This simple approach has been shown to be very effective at fault detection, finding many previously-unknown errors. However random test generation may suffer from the inability to capture more complex logic.

2.2 Guided Random Testing

Guided Random Testing (GRT) [5] proposes six techniques to help guide random test generation. The techniques use static analysis and heuristics at run-time to better inform feedback-directed test generation. In particular, it aims to improve defect detection, code coverage, and mutation score metrics by utilizing information about program types, data, and dependencies during the test generation process. The authors evaluated GRT on 32 real-world projects and reported that it outperformed major peer techniques in terms of code coverage and mutation score. The approach involves extracting knowledge from the system under test (SUT) through static analysis and combining it with dynamic feedback, such as exact type information and test coverage data, to guide the generation of effective test cases.

STATIC TECHNIQUES:

- **Constant Mining:** Add constants found in the source code (at the class or global level) to the test input pool.

DYNAMIC TECHNIQUES:

- **Dynamic Typing (Elephant Brain):** Use run-time types for generated values in the pool for more diverse inputs.
- **Min-Cost-First (Orienteering):** Favor method sequences that cost less to execute.
- **Min-Coverage-First (Bloodhound):** Favor methods with lower coverage criterion first.

HYBRID (STATIC + DYNAMIC) TECHNIQUES:

- **Input Fuzzing (Impurity):** Fuzz primitive values with a Gaussian distribution and objects with calls to impure (affecting state) methods at run-time.
- **Input Construction (Detective):** If the input type of a tested method does not exist in the pool, then construct or search for methods generating the input type.

Figure 2: Classification of GRT Techniques

We summarize and rename the main techniques in Figure 2. An exact copy of the diagram showcasing the static analysis and run-time effects of each GRT techniques from the original paper [5] is also given in Figure 3.

Note that DYNAMIC techniques refer to heuristics done at test generation run-time, rather than running the program to generate tests for. Additionally, inputs refer to parameters to methods in the method sequence for test generation. The input to Randoop or GRT is the SUT, but the input to the sequences are primitive and reference values generated from the test generation algorithm.

As described in Figure 3, GRT utilizes insights gathered from the SUT to guide each step of the test generation. The process begins with a lightweight static analysis to extract constant values from the target classes. These values serve as "seeds" for constructing complex object states throughout the testing process. Furthermore, a static purity analysis classifies methods under test (MUTs) as either pure or impure, enabling more efficient generation of novel object states. A third static analysis examines dependencies between method parameter types (test inputs) and return types (outputs) to determine which object types are critical for testing MUTs. Since precise types may only be identified during execution, GRT also incorporates dynamic analysis to capture type dependencies at run-time.

To showcase each of the 6 techniques presented in GRT, we can re-examine the program snippet in Figure 1. Building on top of Randoop, **Constant Mining** may statically mine the `MAX_SIZE` and `DEFAULT_SEED` constants in the program and add them to the input pool. **Dynamic Typing** will dynamically determine that sequences involving calls to `createList()` will return an `ArrayList` and can thus add entries in the pool of `ArrayList` values. Inputs from this pool can then be passed into methods like `ensureCapacity()` that expect `ArrayLists`. **Min-Cost-First** may increase the probability of selecting cheaper method sequences to construct like a single call to

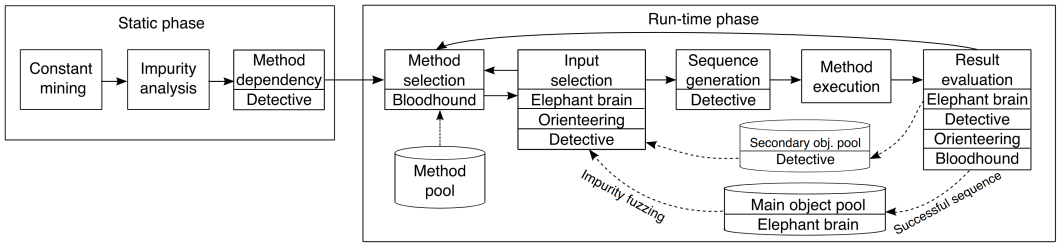

Figure 3: Direct copy of GRT Paper's Diagram of 6 techniques [5]

`new ArrayList<>()` over longer sequences that involve multiple, repeated calls to `ListNoisifier.createList()`. **Min-Coverage-First** may likewise target methods with less code coverage with higher probability. **Input Fuzzing** may apply Gaussian noise to primitive values like the `size` parameter for `ensureCapacity`, and may apply impure functions like `add()` or `remove()` to `List` variables passed in to any of the three methods in the snippet. Finally, **Input Construction** may first search for a constructor of an `ArrayList` or `List` if the iteration of random test generation selects one of the three methods in the snippet before having ever computed and added a method sequence for type `List`.

2.3 EvoSuite

EvoSuite [1] uses evolutionary search to generate whole test suites from scratch. The search is based around evolution: a population of candidates are evolved via mutations and crossover. Reproduction rates are higher for candidates with a higher fitness level. In the context of test generation, EvoSuite represents whole test suites as candidates and coverage criterions as fitness levels. Test suites are composed of test cases created by sequences of method calls. Mutations to test suites occur by adding or mutating test cases. Mutations to test cases occur by adding or deleting statements and parameters in source code. Crossover between two test suites involves the exchange of randomly chosen test cases across each test suite.

As mentioned in Section 1, EvoSuite has been a leading technique in test generation. However, it can be computationally expensive to run the search.

3 METHODOLOGY

We replicate a subset of the techniques proposed in the GRT paper [5]. The techniques primarily augment test input quality and method selection in Randoop.

There has been an ongoing process to integrate the techniques as optional parameters in Randoop [6]. In fact, two techniques (Min-Cost-First and Min-Coverage-First) have already been implemented and merged into the Randoop, and at the time of implementation, there were branches with changes for the remaining techniques.

Our approach was to first target and verify the unimplemented phases. In particular, we implement and evaluate **Dynamic Typing**, **Min-Cost-First**, **Min-Coverage-First**, and **Input Construction**. We *do not* implement **Input Fuzzing** and **Constant Mining** techniques due to scope and unclear interpretation of the original GRT paper. However, for each technique, we provide insights on the implementation and assumptions we make.

We then evaluate the GRT implementation to verify the results as reported in the original paper. We focus on defect detection, code coverage, and mutation score on subsets of the GRT techniques. More of our evaluation methodology is detailed in Section 5.

4 IMPLEMENTATION

The implementation of GRT is built on the Randoop codebase. Each technique can be enabled via configurable flags on the Randoop base command. The changes described build on the existing Randoop implementation [6] where there has been an ongoing process to implement the techniques of the GRT paper. For each technique, we describe assumptions made and if why we were not able to implement if it is an excluded technique.

4.1 STATIC: Constant Mining

Constants can be mined from the source code at the global and local levels. The weighting of which random input to then select during run-time is weighted by inverse document frequency as described in Section III.A for the original GRT paper [5].

One implementation we considered interpreted p_{const} as the probability of using any constant at the extracted level. From our understanding of the GRT paper, this should instead refer only to the probability that a local, class-level constant is used. However, it is still difficult to tell from the wording in the original GRT paper how input selection between constants from the global pool or class-level pool at test generation run-time works. It is unclear *which pool* should be selected from, and with what probability. Additionally, it is unclear how the probability of selecting some constant from the class-level pool can be represented as a single static constant instead of a probability relative to the other constants in the pool. Presumably, a change in the size of the pool should affect the probability of selecting a constant.

Consequently, we felt that this change could not be verified in the scope of this project, and may miscontrue the effect that **Constant mining** has on test generation performance. We thus EXCLUDE this technique from our evaluations. Future work will require deliberation and investigation into the original intent of the implementation.

4.2 DYNAMIC: Dynamic Typing

Run-time types can be determined at test generation run-time. Method sequences can then be added to the input pool corresponding to the run-time type, rather than the compile type. The method sequence is augmented using type casts.

We believe that this technique is straightforward and correctly implements the technique as intended by the original authors in Section III.C of the original paper [5].

4.3 DYNAMIC: Min-Cost-First

Cheaper method sequences can be favored to generate more tests given a constrained time budget. In particular, the execution cost can be computed as a factor of execution time and the number of method calls in the sequence.

The implementation considered closely adapts this weighting. The function in Section III.E in the original paper [5] describes how cost is inversely proportional to execution time and to the square root of the number of method calls.

A couple key assumptions are made in the implementation. It is assumed that sequence execution times do not change over runs since the input does not change. As such, the summation of different execution times and method calls can be replaced with a multiplication by the number of times teh input is selected by the random test generation algorithm. To avoid divide by zero errors, input sequences with zero execution time are assumed to take 1 nanosecond. These are reasonable assumptions made for ease of implementation and performance.

4.4 DYNAMIC: Min-Coverage-First

Methods with lower branch coverage can be targeted with higher probability during random test generation. This increases the probability of covering more branches, thus increasing the probability of generating a test that finds a defect, given that a defect is randomly present in some branches of the program.

The implementation closely adapts the weighting to select methods to test with higher probability based on the uncovered branch ratio and the ratio of the number of times the method was used to create a new regression test. The equation for the weighting is given by the equation in Section III.F in the original paper [5]. The intuition for the algorithm centers around a multi-armed bandit strategy that favors methods with low code coverage and downgrades weights logarithmically when a method is successfully tested.

A key assumption for the implementation is that the original GRT paper's reference to a "successfully invoked" method is one that *creates* a new sequence kept in a regression test, whereas the implementation interprets this as the number of times the methods *appears* in any regression test.

4.5 HYBRID: Input Fuzzing

The representation of object states can be diversified by fuzzing or modifying the input objects. This can be useful for reasoning around edge cases.

The implementation adapts primitive value fuzzing as described in Section III.B of the original paper [5], applying Gaussian noise to numeric values and random Insert, Delete, and Replace operations on String objects. At test generation time, inputs sequences are fuzzed. However, fuzzing on reference type objects is out of scope. This requires a purity analysis to determine impure methods that can modify object state. While the original GRT paper used an existing purity analysis called ReIm and ReImInfer [3], we were unable to build the project. Other purity analysis methods exist, but to our knowledge, there is no easily-adoptable implementation.

Consequently, we feel that only fuzzing primitive objects does not demonstrate the full effect of Input Fuzzing. Thus, we EXCLUDE this technique from our evaluations. Future work will require a purity analysis implementation, which is an ongoing research direction in the field. This can then inform which methods may mutate reference type objects, enabling input fuzzing on such objects.

4.6 HYBRID: Input Construction

A demand-driven approach to construct missing input objects at test generation run-time can prevent the test generation loop from failing to generate a method sequence. To augment this, a static analysis on return types of MUTs can help build a recursive dependency structure on type dependencies outside the SUT.

The implementation modifies the behavior when an input type does not exist for a randomly selected method. For these types, a recursive dependency search is done. For each type, constructors and methods returning the object are identified, with input types added to the queue. The constructors for such types are recursively iterated over until successful sequences to construct the original input types can be found. The intermediate types are also saved in the input pool.

Notably, the GRT paper describes in Section III.D how input types that were not originally defined by the user (like static library objects) are maintained in a *secondary* object pool. For the scope of this project, we work with the existing implementation to store such inputs in the larger pool. The assumption is that this does not affect correctness, but may potentially affect performance. An investigation into the performance improvement from using a secondary pool (such that constructed inputs do not affect method sequence lookup during test generation) is scoped for a future work.

5 EVALUATION

To evaluate our implementation of GRT, we use the same benchmarks from the original GRT paper. As described in Section 4, we study 4 of the original 6 techniques that we felt confident in the implementation.

Our primary research questions are:

- Q1: How accurate are the defect detection, code coverage, mutation score metrics as reported by the original GRT paper?
- Q2: How much does each technique contribute to an increase in code coverage?

This project is scoped to verify the evaluation of the *effectiveness* of 4 of the GRT techniques (Q1). Defect detection, code coverage, and mutation score have been found to be closely correlated with a test suite's ability to find faults. However, the reader should note that these metrics do not perfectly represent test quality, but we use them for our evaluation since the GRT paper used these metrics for their evaluation. In particular, the key metrics of test generation are as follows: the test suite should be 1) *effective*, capable of detecting faults, 2) *readable*, with clarity and small test size, and 3)

maintainable, with a co-evolution between code and tests. On the other hand, the GRT paper[5] mostly only evaluates how *effective* the generated tests are.

In conjunction with the measurement of GRT techniques on test quality metrics, we compare how the techniques compare (Q2) over these metrics with a series of ablation studies. In particular, we compare how GRT (over all 4 techniques) compares with each combination of GRT with one of the techniques removed. This approach can help identify the effect of the removed technique on the generator's performance.

We aim to replicate or counter the general trends (and not exact results) shown in the GRT evaluation. For the experiment setup, we use the same hyperparameters as GRT: coverage guidance weighting and time intervals $p = 0.99$, $\alpha = 0.9$, and $t = 50$ seconds. For the benchmarks, we measure the same Defects4J framework [4], as well as a subset of the 32 real-world programs and their respective versions.

As mentioned before, we review the general trends from the GRT paper [5], and not exact results. Exact results are infeasible to replicate because of differences in hardware and the impact of random noise. The original GRT paper usesm a computer cluster where each node runs Ubuntu 12.04 LTS with Linux kernel 3.5.0, on a 16-core 1.4 GHz AMD 64-bit CPU with 48 GB of RAM. Our hardware setup uses a 128-core 2 GHz AMD EPYC 7702P CPU with 512 GB of memory running Ubuntu 24.04.

5.1 GRT Defect Detection

We repeat the Defect Detection experiments from the original paper. We use the same Defects4J framework [4] from the original GRT paper[5], which enables controlled testing to determine how well test generation tools can capture defects. The framework contains a database that (at the time) consisted of 5 open source projects with 357 faults. Each fault has an incorrect and correct version of the program. The evaluation script runs the test generation tool on the correct version of the program before running the tests on the incorrect version, in which case the outcome is that the generated tests failed to catch the defect, successfully detected the defect, or failed even on the correct version of the program. We aim to replicate a similar methodology from the GRT paper [5] showing the number of defects caught by GRT, Randoop, and EvoSuite in 120, 300, and 600 second global time budgets. Unlike in the original paper where each tool is run 10 times to generate 10 separate test suites with the defects found aggregated as a union of the defects found by each test suite, we instead run the tool a single time due to compute constraints.

5.1.1 Defect Detection on 4 GRT Techniques. We perform an ablation study over the techniques to evaluate the contribution of individual techniques or components to the overall performance of a model or system. By systematically removing or modifying specific elements, we can observe the effect of each component on the combined system's performance. For instance, if a particular technique is removed and there is a noticeable decrease in the system's performance, it indicates that the removed element plays a significant role in achieving high rates of defect detection. Conversely, if removing a component has little to no effect, it suggests that the component may not be as crucial.

	Isolated	Randoop			GRT Combined			EvoSuite		
Program	Defects	120s	300s	600s	120s	300s	600s	120s	300s	600s
JFreeChart	26	11	11	12	3	5	7	10	11	12
Apa. Math	106	17	21	23	11	13	17	35	35	39
Joda Time	27	1	1	4	1	1	1	7	9	9
Apa. Lang	65	9	9	10	2	6	6	22	23	26
Total	224	38	42	46	17	25	31	74	78	86

Figure 4: Defect Detection in Defects4j Benchmarks

	Isolated	No Dynamic Typing			No Input Construction			No Min-Cost-First			No Min-Coverage-First		
Program	Defects	120s	300s	600s	120s	300s	600s	120s	300s	600s	120s	300s	600s
JFreeChart	26	4	5	5	10	10	13	4	6	6	6	6	6
Apa. Math	106	14	17	25	16	21	33	10	14	16	9	14	17
Joda Time	27	1	1	2	0	0	1	0	0	1	0	0	1
Apa. Lang	65	4	4	5	4	4	8	3	3	5	4	7	4
Total	224	23	27	37	30	35	55	17	23	28	19	27	28

Figure 5: Defect Detection in Defects4J Benchmarks with Ablations Over Defects4J

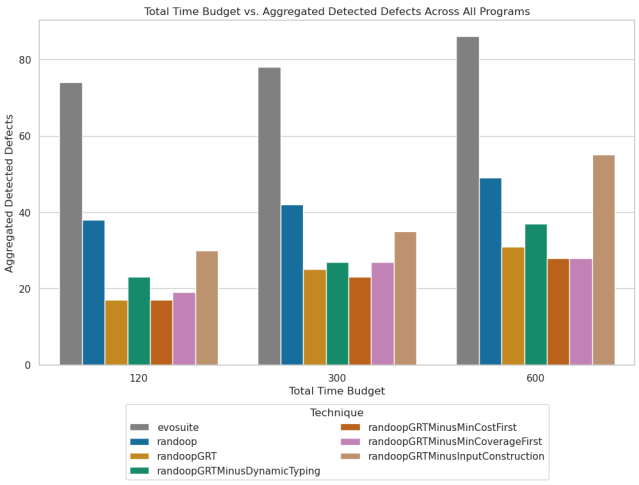

Figure 6: Total defects detected over the four subject programs (JFreeChart, Apache Commons Math, Joda-Time, and Apache Commons Lang) with ablations over the four GRT techniques.

The results are given in Figures 4 and 5. Figure 4 shows the defect detection performance between Randoop, EvoSuite, and Randoop extended with all four GRT techniques (listed as GRT Combined). Figure 5 shows the results of the ablation study where each of the four techniques was disabled from the system (and thus Randoop with GRT extensions was run with the remaining 3 techniques at a time).

Figure 6 visualizes the combined results from Figures 4 and 5 by plotting the total detected defects over all four subject programs for each of the seven total generator configurations.

A surprising result is quickly revealed - extending Randoop with the GRT techniques appears to hurt defect detection performance, as demonstrated by the lower number of defects detected by Randoop with GRT techniques as compared to base Randoop in Figure 4 and the corresponding visualization in Figure 6. Upon closer examination, this is because one of the techniques, **Input Construction**, performs very poorly.

Input Construction has a strong negative correlation with defect detection efficacy. The ablation that removes Input Construction in Figure 5 shows that the total number of defects detected increases at a 600 second global time budget while the other ablations and the four combined techniques indicate fewer defects detected compared to base Randoop. This relative performance across the ablations is present over all three (120, 300, and 600 total second time budget for test generation) of the time budgets.

Our hypothesis for this behavior lies in the expensive cost of Input Construction. Of our four techniques, this is arguably the most expensive, having both static and dynamic steps in a two-stage process. Due to the limited time allotted to the generators to produce tests, it is possible that the overhead and costs from this technique do not justify any improvements in defect detection. This idea of incurred overhead is supported by the general trend that each of the four generators that include Input Construction (e.g. all GRT techniques combined, removed Dynamic Typing ablation, removed Min-Cost First ablation, and removed Min-Coverage First ablation) slowly close the gap with base Randoop's performance as the time budget is increased.

An initial run shows that GRT without Input Construction can generate 8155 method sequences from 9782 steps (iterations of method sequence extension operation), while GRT with Input Construction can generate 7429 method sequences from 9319 steps generated on a 200 second global time budget on A4J. While these results do not indicate an overwhelming distinction and is only run on a small example, they hint at a possibility of evidence for the hypothesis.

Regardless, further experimentation is needed to confirm or refute this hypothesis. The first possibility is to run more experiments on extended time budgets over the 600-second time budget, and evaluate if Input Construction becomes useful at high time budgets. Another possibility is that, compared to the computing cluster mentioned in the original GRT paper[5] (of which we do not have any details), the hardware used in these experiments is weaker or under more load. Thus, running the experiments on a computer cluster with nodes dedicated to running the test generation job may be useful in verifying if Input Construction is more useful with stronger compute capabilities.

5.1.2 Defect Detection on 3 GRT Techniques. The next evaluation for defect detection was to remove Input Construction and consider the three remaining techniques. With Input Construction disabled for all of these experiments, we follow a similar methodology to the prior experiment. Specifically, we ran another ablation study was on the three remaining techniques, choosing one at a time to remove and studying the remaining efficacy of the two others, as well as comparing their results to running the system with all three techniques combined. Finally, these results were compared to the previously acquired results from EvoSuite and base Randoop.

The results of this evaluation are presented in Figure 7. Here, GRT Combined refers to the combination of the three techniques of Dynamic Typing, Min-Cost First, and Min-Coverage First. Then, the three ablations are ablations on this combination, each removing one of the three techniques and running Randoop extended by the remaining two techniques. Figure 8 plots the total defects each of the four systems detected across the four subject programs similarly

	Isolated	GRT Combined			No Dynamic Typing			No Min-Cost-First			No Min-Coverage-First		
Program	Defects	120s	300s	600s	120s	300s	600s	120s	300s	600s	120s	300s	600s
JFreeChart	26	10	10	13	11	12	12	7	7	8	8	9	10
Apa. Math	106	16	21	34	2	5	7	4	4	5	3	5	6
Joda Time	27	1	1	1	1	1	2	0	1	1	0	1	2
Apa. Lang	65	4	4	8	2	5	7	4	4	5	3	5	6
Total	224	31	36	55	16	23	28	15	16	19	14	20	24

Figure 7: Defect Detection in Defects4J Benchmarks with Ablations Over Defects4J (Without Input Construction)

to the previous evaluation. Figure 8 also reproduces the values of EvoSuite, base Randoop, and the experiments prior to removing Input Construction as points of comparison with the second set of ablation experiments.

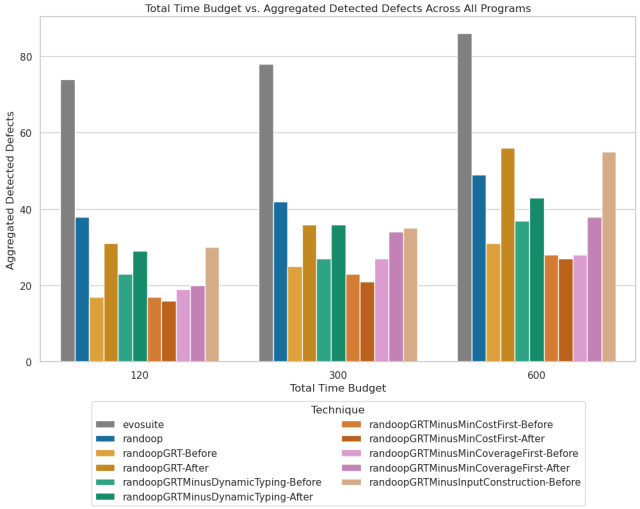

Figure 8: Total defects detected over the four subject programs (JFreeChart, Apache Commons Math, Joda-Time, and Apache Commons Lang) before and after removing Input Construction from the ablation study. The Before suffix refers to experiments run before removing Input Construction; the After suffix refers to experiments run after removing Input Construction. Note that EvoSuite and base Randoop do not have before and after - this is because disabling Input Construction does not affect EvoSuite or base Randoop, and so these values were included only as a reference.

As expected, after removing Input Construction, we found that all of the generators that previously included Input Construction have generally improved defect detection performance. Most importantly, the GRT extended version of Randoop shows better performance than base Randoop.

In the ablations after removing Input Construction, we observe that Min-Cost First is the most significant technique for defect detection performance improvement because the ablation test where it is removed has the greatest impact on performance. Dynamic Typing and Min-Coverage First are both less, but still significant because the ablations missing these techniques show slightly better performance over base Randoop.

At the lowest time budget of 120 seconds, we find that even without Input Construction, the GRT techniques still perform worse than base Randoop. However, this is no longer the case as the time budget is increased. We posit that this is due to overhead

in running the additional GRT techniques that make all of the techniques ineffective compared to just running base Randoop under constrained time budgets. The key takeaway is that the GRT techniques are useful only if the user has sufficient computational resources to run the additional analyses for each technique.

There are also some interesting trends that are difficult to isolate. For instance, removing Dynamic Typing at lower time budgets (120 and 300 seconds) results in similar or equivalent performance to the combined techniques, while the gap widens at the higher time budget of 600 seconds. Additionally, removing Min-Coverage-First at a time budget of 300 seconds has relatively low impact, but the impact is higher at both 120 and 600 seconds. One possibility for these trends is the varying degrees of interdependence between the GRT techniques. As the original GRT paper mentions in a diagram [5], almost all of the techniques have interactions with other techniques. It may be that Dynamic Typing is a more expensive technique that is not worth running on lower time budgets, hence the lower performance of the remove Min-Coverage First ablation (which contains Dynamic Typing) at a 120 second time budget. At the same time, it's possible that Min-Coverage First is also just a fairly important technique, and so its removal would also hurt performance at a 600 second time budget.

One major evaluation to be done in future work is to examine every possible combination of the GRT techniques (beyond just doing ablations). This would be useful in identifying the connections between particular combinations of techniques and identifying which pairs, triplets, or other combinations of the techniques are most important compared to the others. From the results we observe here, it appears that the techniques are not easily analyzed independently and in a vacuum - some of the techniques might only be effective if combined with other techniques, while others might be effective regardless. We were unable to perform experiments on this due to computation limitations - the number of combinations is exponential in the number of techniques, and so this would have increased the computational costs exponentially.

5.1.3 Defect Discovery. One other evaluation from the original paper was new defect detection on open source projects in the GRT paper. We find that this is infeasible to replicate. This is because, while Randoop and GRT can be used to generate error-revealing tests, Randoop and thus GRT are deterministic in the default case, using random seeds to achieve its random results. Therefore, we cannot hope to find the exact version of the repository and attempt to replicate detection of "new" bugs, especially when the random seeds are not given. However, it may be useful to run the GRT implementation for new defects on current open source programs.

5.2 GRT Coverage Scores

5.2.1 Setup. We also repeat the Instruction coverage and Branch coverage distributions as per the regression tests by showcasing the effectiveness of each of the GRT techniques, Randoop, and EvoSuite. In terms of code coverage, we attempted to run each subject program provided in the original GRT Paper[5] for 2 s/class, 10 s/class and 30 s/class in terms of seconds per class. In terms of global time per subject program, we ran it for 0, 10, 50, 100 and 300 seconds over the whole program.

Software (version)	NCLOC	# Class	# Insn.	# Bran.	# Mut.
A4J (1.0b)	3,602	45	9,773	544	936
Apache BCEL (5.2)	23,631	338	65,179	5,133	7,209
Apache C. Codec (1.9)	5,803	76	24,960	1,835	2,747
Apache C. Collection (4.0)	25,840	326	78,974	5,179	10,461
Apache C. Compress (1.8)	17,462	181	57,083	4,634	7,605
Apache C. Lang (3.0)	18,997	141	47,773	7,179	9,057
Apache C. Math (3.2)	81,792	845	288,250	18,576	41,023
Apache C. Primitive (1.0)	9,836	231	18,462	1,446	3,290
Apache Commons Cli (1.2)	1,978	20	3,588	490	512
Apache Shiro-core (1.2.3)	13,818	217	27,964	3,291	3,770
ASM (5.0.1)	24,193	176	65,146	7,475	9,765
ClassViewer (5.0.b)	1,485	23	5,266	470	609
Depasrcsys (10/2008)	204	6	652	88	103
Easymock (3.2)	4,372	79	9,449	915	1,382
Fixsuite (R48)	2,665	36	6,520	374	804
Guava (16.0.1)	66,566	1,546	136,261	11,247	20,709
Hamcrest-core (1.3)	1,253	40	2,199	155	314
Jcommander (1.36)	2,154	34	5,688	640	686
Java Simp. Arg. Parser (2.1)	4,888	69	8,623	714	969
Java View Control (1.1)	4,617	24	15,650	2,064	2,084
Javassist (3.19)	34,574	367	87,381	8,830	n.a.
Javax Mail (1.5.1)	28,271	284	79,599	9,523	11,070
Jaxen (1.1.6)	20,345	175	20,352	3,323	4,338
Jdom (1.0)	8,362	70	20,970	3,196	4,116
Joda Time (2.1 03/2014)	27,638	208	62,627	6,172	9,838
Nekomud (R16)	363	8	809	44	63
Pmd-dcd (5.2.2)	1,608	20	2,902	305	384
SAT4J Core (2.3.5)	17,397	213	41,840	3,815	6,140
Slf4j-api (1.7.12)	1,504	18	2,581	271	265
Tiny Sql (2.26)	7,672	31	20,850	2,237	2,755
Total	460,763	5,911	1,186,321	110,485	159,944

Figure 9: Direct copy of GRT Paper's Software Metrics Table [5]

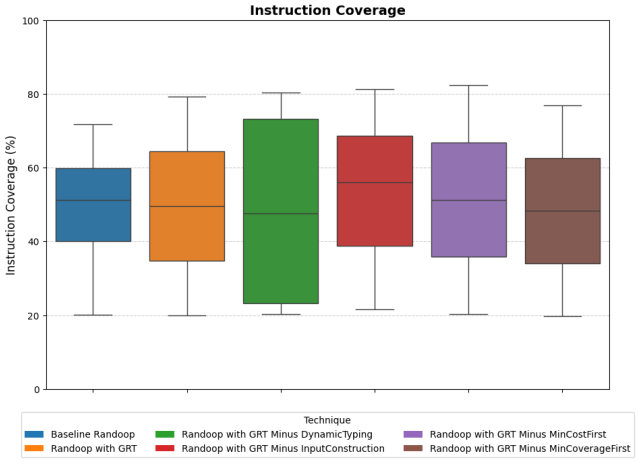

Figure 10: Instruction coverage for each component on 12 benchmarks (300 s).

Unlike the actual experiments performed in the paper, we were only able to reproduce 12 out of the 32 provided subject programs in Figure 9. This is discussed later[1] .For each time budget, generator, subject program configuration, the experiment was run once.

5.2.2 Metrics for Code Coverage. To give a better understanding of the metrics described above, code coverage measures percentage of program elements executed by tests. Specifically, instruction coverage is the percentage of bytecode instructions executed, and

[1]We have been getting build error issues as well as JDK issues when attempting to write scripts for the mutation score. As we clarify each issue, we will be able to run coverage tests on all the benchmarks.

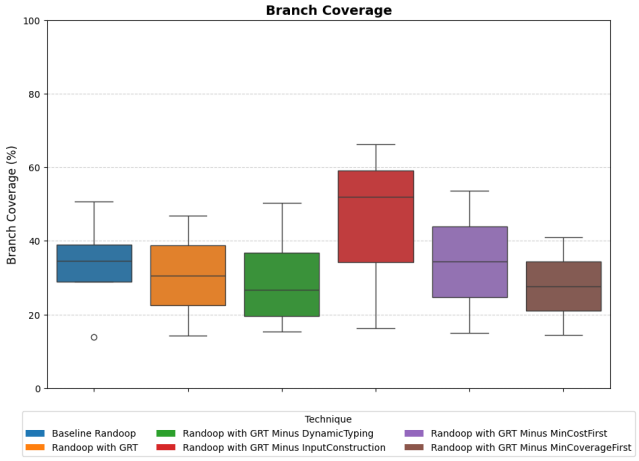

Figure 11: Branch coverage for each component on 12 benchmarks (300 s).

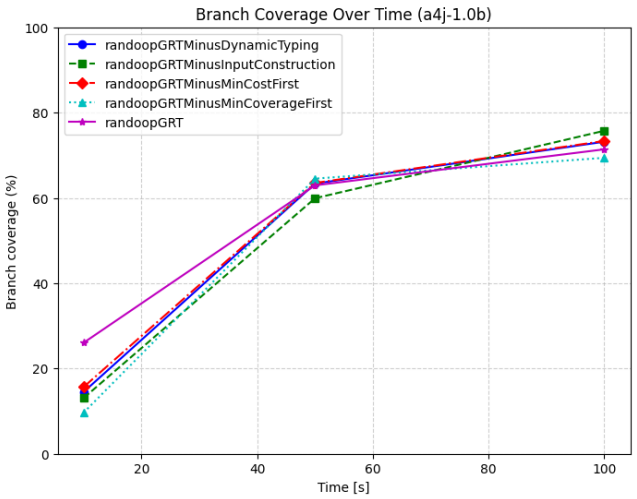

Figure 12: Branch coverage over time for the subject program a4j-1.0b

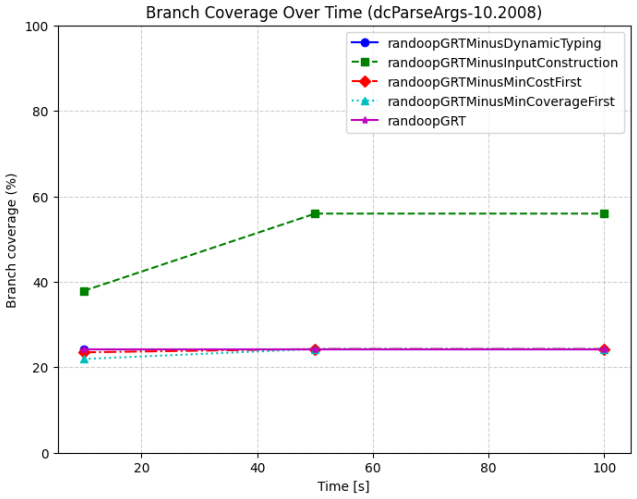

Figure 13: Branch coverage over time for the subject program dcParseArgs-10.2008

branch coverage is percentage of branches (if/else, loops) taken across each class in the subject program. After running on each generator, generated tests are compiled and executed against the subject program. All the tests that are compilable are then evaluated by JaCoCo (Java Code Coverage tool) which tracks which instructions and branches are executed. The coverage percentage is then evaluted by using the formula (executed elements / total elements) × 100%.

5.2.3 Coverage Analysis. From Figures 10 and 11, we observe several important trends when removing individual components from the GRT system in order to test each GRT techniques contribution to the ablation test score. They illustrate the impact of each component's removal on instruction and branch coverage across the benchmark suite.

Notably, in Figure 10, all GRT variants consistently match the baseline Randoop implementation, which could contradict the original paper's central claim that GRT's enhancements improve upon traditional random testing. The differences between the various GRT configurations, however, are less pronounced than might be expected,

The most striking observation is the significant drop in coverage found that removing the Input Construction component actually resulted in slightly improved coverage metrics. Figure 10 shows that we achieve higher instruction and branch coverage compared to the complete GRT implementation. This strongly corroborates our findings in Section 5.1.1, where Input Construction may be counterintuitive. While demand-driven input construction aims to prevent steps from failing due to missing input types, the cost to constructing these inputs may outweigh the costs of skipping the method sequence generation.

While the ablation results don't highlight dramatic differences between configurations, similar to as discussed in Section 5.1.2, the techniques still seem to consistently demonstrate GRT's improvement over baseline random testing. This reinforces the value of the techniques discussed in GRT, even if the contribution of individual components is more subtle than previously understood.

5.2.4 Individual Program analysis. We run GRT with each of its six components disabled individually, and one where all are enabled with 10s, 50s and 100s as the global time budget. We observe a coverage improvement for each component and for full GRT as time increases, but it seems to be dependent on other factors, too.

The three Figures 12, 13, and 14 present branch coverage over time for different variants of Guided Random Testing (GRT) on three subject programs: a4j-1.0b, dcParseArgs-10.2008 and easymock-3.2.

For a4j-1.0b, Figure 12 shows that all test generators show similar patterns, starting with relatively low coverage (10-25%) and improving significantly, reaching approximately 60-65% at the 60-second mark and 70-75% by 100 seconds. The full GRT implementation (randoopGRT) on the four techniques begins with higher coverage (around 25%) but is eventually matched or slightly surpassed by the ablation variants randoopGRTMinusInputConstruction, which has

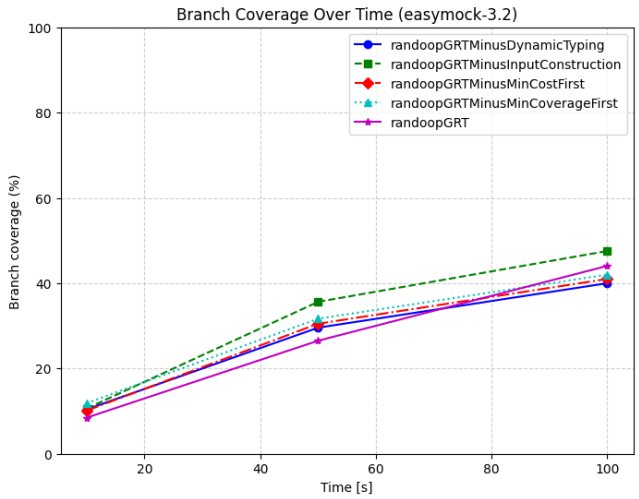

Figure 14: Branch coverage over time for the subject program easymock-3.2

marginally higher branch coverage at 100 seconds. All approaches demonstrate diminishing returns as testing time increases.

For easymock-3.2, depicted in Figure 14, the results show similar results to a4j-1.0b, with the only change being the differing values with the best branch coverage score of almost 50% by the end.

In stark contrast, the dcParseArgs-10.2008 in Figure 13 results reveal that most approaches, including the full randoopGRT, achieve only about 22-25% branch coverage throughout the entire testing period with minimal improvement over time. Notably, the randoopGRTMinusInputConstruction variant significantly outperforms all others for this program, reaching approximately 55-57% coverage. This dramatic difference between these subject programs once again highlights the negative effect of InputConstruction on branch coverage.

In general, a few key ideas that could be reasoned about is the number of classes each program have. a4j-1.0b and easymock-3.2 have much more classes than dcParseArgs-10.2008, which means that dedicating more time in the subject program will show much better coverage results as each generator genuinely has more time to cover more branches in the program. Moreover, for smaller programs, Input construction can play a larger negative role by being extremely expensive and hampering the coverage values.

The results align with our general findings so far that the Input Construction technique contributes negatively to the Coverage metrics, despite the original paper claiming that the combination of all techniques is generally stronger than its individual components. Clearly, for each of the tested subject programs, removing the Input Construction component actually improves performance, suggesting this component might be counterproductive in certain contexts.

6 LIMITATIONS AND THREATS TO VALIDITY

As referenced previously in Section 5, our study focuses on evaluating the effectiveness of GRT at test generation. While we are specifically testing for three useful metrics—code coverage, mutation score, and defect detection—we acknowledge that these do not fully capture test quality. However, we primarily rely on these metrics due to the replication nature of our study, as we aim to validate the claims made in the GRT paper, which also uses these metrics in its evaluation. Additionally, because we do not have the original implementation, we must make assumptions about the original implementation from the wording in the GRT paper. As such, our replication risks incorrect implementation.

To determine causality, we replicate the GRT paper's methodology by running each experiment 10 times. However, we note that the variations between random seeds can introduce unexplained variance, and that arbitrary time constraints may not necessarily fully represent the full test quality. For example, for a specific seed, while GRT may generate more tests in less time, running Randoop to completion could result in higher code coverage, mutation scores, or defect detection.

In terms of generalization, the original GRT paper evaluates numerous real-world open source programs, including Defects4J, which contain real defects. However, test quality is not solely determined by metrics like coverage and defect detection. Factors like *readability* and *maintainability* could also influence the test generation that Randoop and GRT adopt.

Additionally, our experiments are conducted on different hardware than in the original study, and we do not account for the impact of hardware on the runtime or *effectiveness* of test generation.

7 CONCLUSION

Our replication study of the Guided Random Testing (GRT) paper revealed several important insights that contribute to the testing research community. We learned that while GRT techniques can enhance test generation effectiveness, their benefits are not as universal as originally claimed. The performance of individual techniques varies significantly across different programs and time budgets, with some techniques like Input Construction potentially degrading performance in certain contexts.

Based on our observations, we recommend that the research community:

(1) Prioritize releasing implementations alongside research papers to enable proper validation and extension of techniques
(2) Approach composite techniques like GRT with more nuance, recognizing that not all components will benefit all programs equally
(3) Consider the computational overhead of advanced testing techniques when evaluating their practical utility
(4) Conduct every possible combination, such as all 32 subject programs, when performing ablation studies to to better understand the contribution of each technique
(5) Report detailed experimental configurations to ensure reproducibility of results

Without our observations, researchers might continue to assume that combining multiple testing techniques always yields better results, potentially wasting computational resources on ineffective techniques. Our work demonstrates the value of critical replication studies and highlights the importance of carefully analyzing the

trade-offs between technique complexity and performance gains in automated test generation.

REFERENCES

[1] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[2] Gordon Fraser and Andrea Arcuri. 2015. EvoSuite at the SBST 2015 Tool Competition. In *8th International Workshop on Search-Based Software Testing (SBST'15) at ICSE'15*. To appear.

[3] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA 2012, Object-Oriented Programming Systems, Languages, and Applications*. Tucson, AZ, USA, 879–896.

[4] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[5] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 212–223. https://doi.org/10.1109/ASE.2015.49

[6] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84. https://doi.org/10.1109/ICSE.2007.37

[7] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. 2022. EvoSuite at the SBST 2022 Tool Competition. In *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. 33–34. https://doi.org/10.1145/3526072.3527526

APPENDIX A: CODEBASES

(1) Our fork of Randoop lives on Github at https://github.com/edward-qin/randoop-grt.
(2) Our fork of Defects4j lives on Github at https://github.com/edward-qin/defects4j-grt.
(3) We are using an existing (private) repository for coverage and mutation score at https://github.com/randoop/grt-testing/tree/aditya-scripts-branch