PropCov: Effective Coverage for Property-Based Testing

ANONYMOUS AUTHOR(S)

Property-base testing (PBT), introduced by Haskell's Quickcheck, is becoming more popular with successful ports for other languages, such as Java's `junit-quickcheck`. With PBT, developers write a *property test* and a *data generator*. The *data generator* takes a source of non-determinism and uses it to output well-formed data. The *property test* exercises the System Under Test (SUT) using the random well-formed data from the generator to ensure a particular property always holds (*e.g.,* data serialized and deserialized should be equal to the original data). The PBT framework then performs many *trials*, each generating fresh data and executing the property test. A test failure shows a bug to developers, typically in edge-cases. A passing test gives some assurance on the quality of the SUT with regards to the property being tested. Unfortunately, well-known test coverage tools that are instrumental for understanding unit testing work poorly for PBT.

In this paper, we present PropCov, a tool for understanding coverage in PBT that also provides suggestions for coverage improvement. PropCov employs a novel combination of static analysis with PBT to approximate the maximum possible coverage, providing an effective measure of the PBT coverage and making suggestions to developers of where to improve existing tests. PropCov features an easily extensible architecture that can support new languages, build systems, and PBT frameworks. We evaluated PropCov using 25 Java projects using `junit-quickcheck` or `jqwik`, totaling 293 properties, and found that existing tools report missed coverage that is impossible to reach (86% of lines that JaCoCo reports as not covered), which leads developers to consider hundreds of extra lines of code (2910). Unlike existing coverage tools, PropCov results are accurate — only 7.5% of all properties contain unfeasible code, and PropCov only misses 4% of reachable code. Using PropCov's suggestions, we increased the coverage of 42 tests over 7 projects and found 4 new bugs in 4 projects.

ACM Reference Format:
Anonymous Author(s). 2026. PropCov: Effective Coverage for Property-Based Testing. 1, 1 (January 2026), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Property-based testing (PBT) allows developers to express sophisticated properties that should always hold true. For instance, Figure 1 shows a *round-trip property* implemented in Java with `junit-quickcheck` that applies to libraries that serialize and de-serialize data, and states that *any data serialized and then de-serialized must be the same* (Line 5). A PBT framework runs many *trials* of the same property (*e.g.,* 100 is a common default), each with fresh data from a *generator* that takes a source of randomness and creates concrete data — lists with random integers and random size (Line 9). A trial that finds a violation can be later replicated using the same generated data.

Unfortunately, a property test that does not find any violations gives developers a false sense of security in thinking that the System Under Test (SUT) implements the property correctly. In fact, the property test may be unable to find problems because either: (1) the generator lacks diversity (*e.g.,* only generating large numbers), or (2) the property is too narrow and actually does not cover the expected functionality (*e.g.,* a different property focused on list concatenation never covers empty lists). Developers often identify PBT's opaqueness as a barrier to adoption, comparing PBT poorly

ACM XXXX-XXXX/2026/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
01: interface Serial<T> { byte[] serialize(T t); T deserialize(byte[]); }
02:
03: @Property void roundtrip(@From(ListGen.class) List target)   {
04:     Serial<List<Integer>> s = new IntListSerial();
05:     assertEquals(target, s.deserialize(s.serialize(target)); }
06:
07: List ListGen.generate(SourceOfRandomness r, GenerationStatus s)           {
08:     List ret = new LinkedList();
09:     for (int i = 0 ; i < r.nextInt(100) ; i++) ret.add(r.nextInt(1_000_000));
10:     return ret;                                                           }
```

Fig. 1. Example of a test for property (`roundtrip`) and associated data generator (`ListGen.generate`). Figure 2 shows implementations of interface `Serial`. Argument `r` is a random-number generator, and `s` is an (unused) object that can keep state between calls to `generate`. The generator outputs a list of random size with random contents (Line 10). The test then serializes and deserializes each generated list, testing if the property holds by checking whether the result is the same as the original list (Line 5).

against (unrealistic) exhaustive tests that *"hit things I actually care about"* [3], and mentioning unknown *"holes in the testing coverage"* [4]. A recent study on developers using PBT [20, 21] also reports on developers struggling to trust PBT that does not find any issues, and resorting to: injecting bugs, manual analysis of the generated data, and ad-hoc analysis of coverage reports.

Making matters worse, well established techniques to measure and improve commonly used unit testing fail with PBT. One such technique is code coverage (*e.g.,* using JaCoCo): run a test suite with code coverage enabled and write new tests for uncovered code. Writing new property tests to reach particular lines is misguided, because a suite that covers line L with one test T_1 but misses the same line L with another test T_2 is still incomplete, as L may be critical to finding an error with T_2 even though T_1 already covers it.

In this paper, we present a technique that makes commonly used and well understood code coverage meaningful for PBT. Our **first novel insight** is to consider each individual property test in isolation. Unfortunately, when applied to a single test, existing code coverage techniques include a large amount of uncovered code that is actually impossible to reach. We show in Section 4.1.3 that the vast majority on such JaCoCo reports is impossible to be reached from the test (86% lines over 85% methods), leading developers to consider 2910 unfeasible lines over 696 methods. Our **second novel insight** is to restrict coverage reporting for each property test to only code that can be reached from that property test. Note that it is challenging to find which code can be reached from a particular test, especially in imperative languages that keep a large amount of mutable state and use dynamic function calls (*e.g.,* method dispatch based on the runtime type of the receiver object). We use existing state-of-the-art techniques to build a call-graph that approximates the maximum theoretical coverage for each property test, augment that call-graph with coverage information observed when running the property test, and present it as a report to the developer. The amount of uncovered code in our report provides the developer with a concrete measure of how complete each property test is, and where to focus efforts to improve it.

We implement our technique as an extensible framework that targets JVM languages (*e.g.,* Java, Scala, Kotlin, Groovy) and respective PBT frameworks (*e.g.,* `junit-quickcheck` [27], `jqwik` [1]). Our tool, named PropCov,[1] leverages state-of-the-art analysis framework OPAL [16, 17, 24] to deal with dynamic language features — which concrete methods can be reached via dynamic method calls, and targets of `invokedynamic` [45] (*e.g.,* lambda functions and stream operations in Java). PropCov drastically reduces the effort required to identify where to improve existing property tests

[1]Short for **Prop**erty **Cov**erage

(or their generators) via a novel heuristic technique (Section 3.4) that suggests possible program points to improve. We describe the design and implementation of PropCov for Java together with an extensive experimental evaluation using 293 properties over 25 projects. PropCov results are accurate — only 7.5% of all reports contain unfeasible code (Section 4.1.2), and PropCov only misses 4% of reachable code (Section 4.1.1). Following PropCov's recommendations, we improved the coverage of 42 properties and found 4 bugs, all on projects unfamiliar to us (Section 4.2.3). At the time of writing, 2 bugs were confirmed and fixed by the developers, and we submitted 9 pull requests, 5 merged by the developers (other bugs/PRs are still pending). PropCov also allows to perform **novel scientific studies** on PBT in general. Using PropCov and our corpus of projects, we measure the overall coverage of existing PBT as 55%/58% of lines/methods, as opposed to 10%/13% reported by JaCoCo (Section 4.2). We also measure the role of the number of trials in PBT (Section 4.2.2), and recommend a lower default, 10 instead of 100, together with using PropCov's reporting to understand when to increase it.

In summary, this paper makes the following contributions:

- A novel coverage mechanism specifically designed for PBT based on a static call-graph and dynamic code coverage, comparing their theoretical maximum coverage with the observed coverage during a test run.
- The modular design of PropCov, a novel tool that can be adapted to any language and PBT framework that targets JVM bytecode (*e.g.,* Java, Scala, Kotlin, Groovy).
- The implementation of PropCov for Java, together with an extensive evaluation using 293 properties over 25 Java projects. In the process, we improved 42 property tests, and we found 4 new bugs, all on codebases unfamiliar to us.
- A novel study on existing PBT for Java, not possible without PropCov, that measures PBT test coverage as 55%/58%, and suggests lowering the default number of trials to 10.
- A replication artifact that allows future researchers to reproduce the results in this paper, and to build on PropCov. We will publish PropCov as open-source and make the artifact freely available upon acceptance. Reviewers can access an early version of the artifact that can replicate all experiments described in this paper (Section 7).

2 Property-Based Testing and its limitations

To introduce property tests, let us consider the example property specified in Lines 3–5 of Figure 1 which uses `junit-quickcheck` [27]. Such a *round-trip property* is common for serialization libraries and states that serializing and deserializing any valid data (provided via argument `target` on Line 3) should result in the original data (Line 5).

PBT frameworks execute the same property test many times, and we denote each execution as *a trial.* For each trial, the PBT framework runs the test with fresh random data in an effort to find bugs via input diversity, ideally testing corner-cases that are hard to anticipate by the developer. In our example the property test expects a well-formed list. PBT allows developers to write *generators* to bridge the gap between easy-to-generate random data and well-formed input data. Our example has such a generator (Lines 7 to 10) that takes a random number generator (argument `r`) and uses it to generate a well-formed list with random size and random contents (Line 9). When PBT frameworks find an input that violates any assertion in a test, they report the generated inputs together with the seed used to generate that particular input. Later, developers can use that information to trigger the same bug reliably, assuming test correctness depends only on the arguments passed to it (*i.e.,* tests are not flaky [34]).

However, **what happens when PBT cannot find any bugs?** There are three options that explain such behavior. **First**, the *System Under Test (SUT)* implements the property correctly. Of

```
11: class IntListSerial implements Serial<List<Integer>> {
12:   byte[] serialize(List l) {
13:     Serial<Integer> bS = new ByteSerializer(), sS = new ShortSerializer(),
14:                     cS = new CharSerializer(), iS = new IntSerializer();
15:     byte[] ret = new byte[0];
16:     for (Integer i : l)       {
17:       Serial<Integer> s;
19:       if (i<8) s=bB; else if (i<0xFF) s=cS; else if (i<0xFFFF) s=sS; else s=iS;
20:       ret += s.serialize(i);  } // end for
21:     return ret;             } // end method serialize
22:   ...
23:   void m1() { ... } ...  void m100() { ... }          } // end class IntListSerial
24: class ByteSerializer  implements Serial<Integer> { ... }
25: class ShortSerializer implements Serial<Integer> { ... }
26: class IntSerializer   implements Serial<Integer> { ... }
28: class CharSerializer  implements Serial<Integer> { byte[] serialize(Char i) { /* BUG */ } ... }
```

Fig. 2. Example Java program called by a property that leads to large useless reports, that include unfeasible methods `m1` through `m100`, when using typical code coverage tools. For Line 7 in Figure 1 to fail, Line 11 needs to output a number in a small range that fits in a `char` but not in a `short` or `byte`, which is very unlikely (chance of 0.002%) and thus leads to the test passing when it should fail. For compactness, we use the operator += on Line 20 to concatenate byte arrays and elided overrides for methods `serialize`/`deserialize` in classes `ByteSerializer`, `ShortSerializer`, `IntSerializer`, and `CharSerializer`.

course, testing cannot guarantee the absence of bugs [5], so developers must be careful when considering this option. **Second**, the generated input was not diverse enough to find existing bugs. For instance, one of the bugs we found for project `manu` [15] requires the size of the generated data needs to be just right, which the existing test/generator has such a small chance to generate that it never actually finds it. Unfortunately, as we report in Section 4.2.2, simply increasing the number of trials cannot find this bug. **Third**, the test is *too narrow* and cannot reach code that developers expect to cover. Unfortunately, each option is indistinguishable from the others. Developers often complain about PBT's opaqueness [3, 4, 20, 21].

2.1 Existing coverage tools and PBT

Figure 2 shows a possible implementation of a serialization library that supports lists, which can be tested by the property test in Figure 1. Consider that there is a bug in Line 28 that serializes small integers that fit in two bytes as a character. For example, negative integers lose the sign when serialized, and are then deserialized incorrectly as positive. Also, the implementation of the `IntListSerialized` is more complex than just the code shown. It uses many small methods to encapsulate and reuse behavior, exposing state safely to client classes, thus following good general practices of software engineering. The example abstracts such unrelated methods as `m1` through `m100`.

Note that the property test in Figure 1 *can* find this bug. However, each trial generating an integer in the range 0–2^{31} has a very low chance of generating an integer in the small range 8–256 in the 100 trials that PBT frameworks may use as default. In practice, the property test does not find this bug and the test always passes. This bug is adapted from a real bug in project `manu`, which we found when addressing a PropCov alert that not all serializer classes were reached, and addressed by ensuring the generator outputs data fitting each possible class.

Developers typically use code coverage tools, such as JaCoCo [26], to understand how tests exercise the SUT. Using such a tool on a suite of property tests may not reveal this particular bug,

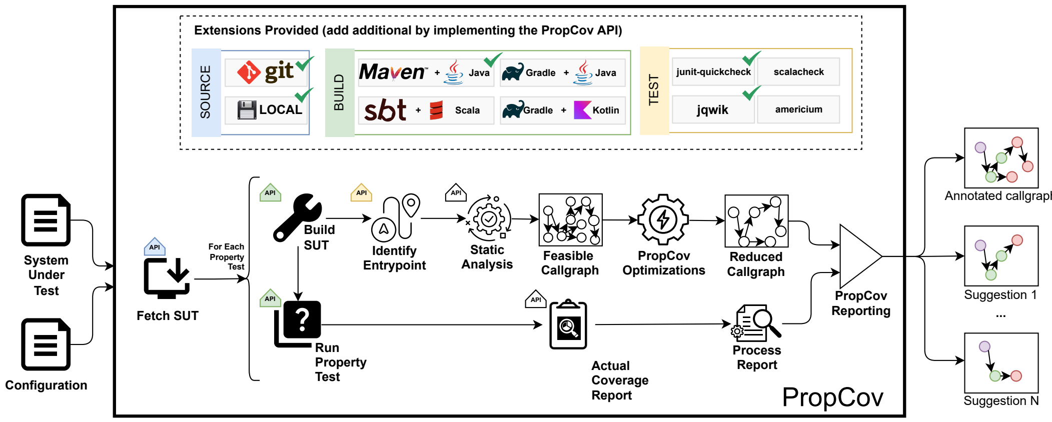


Fig. 3. Overview of PropCov's architecture. We note that PropCov is modular to enable supporting any build system, language, PBT framework, static analysis, and coverage tool that targets JVM bytecode. The green checkmarks denote what PropCov's prototype supports. PropCov currently supports Opal [16, 17, 24] and JaCoCo [26], as the static analysis and coverage tools, respectively (not shown).

as another property test may cover Line 28 but not reveal the bug (*e.g.*, ensuring characters with accents are serialized correctly). It thus makes sense to focus on each individual property test. Unfortunately, JaCoCo considers all methods in the class (and all methods in all other classes loaded during the test execution), including unrelated methods `m1` through `m100`. A developer worried about the example property test we are following never failing is thus left with a useless coverage report that contains a very large number of *unfeasible methods* — methods that cannot possibly be reached by the property. In Section 4.1.3, we show that most of the code in JaCoCo's reports (86% lines over 85% methods) is unfeasible. Even though the report does state that Line 28 and respective branch are not covered, we believe it is very hard for the developer to notice it among all the code listed in the coverage report.

We note that the code in Figures 1 and 2 is based on a real property written for the industrial project `mph-table` [28], and it is thus representative of how practitioners actually use PBT. We adapted the code to include a bug from project `manu` [15] to also show how easy it is to miss potential bugs without proper coverage report.

3 PropCov

PropCov is a novel approach for measuring coverage for PBT, which we developed with the main goal of solving the problems identified in Section 2. Another goal PropCov's design is to be easily adaptable to other languages and PBT frameworks, based on JVM bytecode. As such, PropCov follows a modular architecture based on *extensions* for each source code repository, build system, and PBT framework. Figure 3 shows an overview of PropCov's architecture.

First, PropCov takes as input the *System Under Test (SUT)* and a *configuration* that defines a collection of property tests that are part of a project. PropCov starts by using the first extension — *source* — to acquire the source code of the project specified. Currently, PropCov supports local projects and Git repositories. Then, PropCov uses the second extension — *build* — to turn the source code into JVM bytecode. Currently, PropCov supports Java projects built with Maven [6]. We believe it is straightforward to extend PropCov to new build systems and languages, such as Scala built with sbt, or Java/Kotlin built with Gradle. After building a project, PropCov uses the next extension — *test* — to interact with the particular PBT framework that the project uses. Our current prototype supports `junit-quickcheck` [27] and `jqwik` [1], and we believe it is straightforward to extend PropCov to more PBT frameworks such as scalacheck [38] and americium [37].

PropCov's next step is to approximate the theoretical maximum coverage of the property using static analysis, resulting in a feasible callgraph. PropCov's prototype currently supports Opal [16, 17, 24], and its modular design allows future users to add different analyses, such as Soot [47]. Note that Opal already supports many analyses, which we describe in Section 3.2.

Unfortunately, we found that the feasible callgraph still contains inaccuracies due to common code patterns, primarily due to exception propagation and use of iterators/iterable objects. PropCov optimizes the resulting callgraph to remove such inaccuracies, further reducing the size of the callgraph.

Finally, PropCov executes the property and uses another extension to obtain a coverage report. PropCov's current prototype supports the de-facto tool for Java coverage JaCoCo [26]. Then, PropCov combines the analysis results with the actual coverage observed, in the form of an annotated callgraph with suggestions of which paths seem to yield the best improvement if covered.

3.1 Writing Extensions

PropCov can be extended by writing new extensions, thus supporting new: repositories (*e.g.,* svn), build systems (*e.g.,* gradle, sbt), programming languages (*e.g.,* scala, kotlin), PBT frameworks (*e.g.,* scalacheck, americium), static analyses (*e.g.,* Soot), and coverage tools. The current prototype supports: git repositories and local sources, maven, Java, junit-quickcheck and jqwik, Opal, and JaCoCo. Note that we designed PropCov with projects that target JVM bytecode in mind.

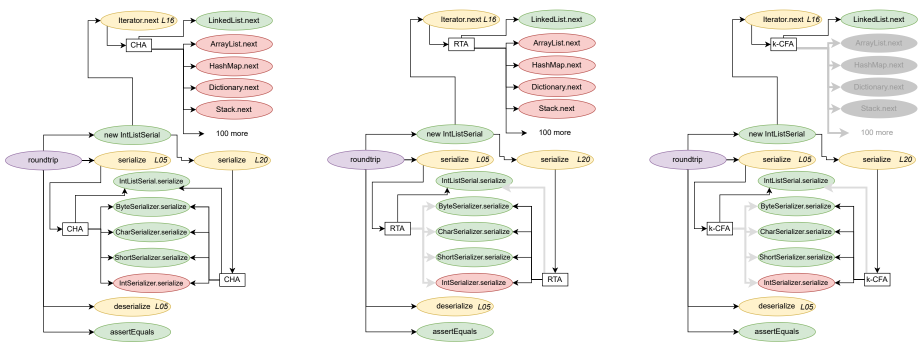
To write an extension, PropCov users need to extend a Java abstract class and provide the following functionality in separate methods: retrieving the **source**, **building** the retrived source and run a particular test, identify which methods are **tests**, perform **analysis** on the SUT to build a suitable callgraph given a property test as entry-point, and measure the **coverage** when executing a particular property tests.

We note that the *build* extension should use *coverage* tool when building the project. In our case, it means that we needed to modify some build scripts to include support for JaCoCo. Such effort was straightforward and mostly copy/pasting the JaCoCo configuration into each Maven build file.

3.2 Static Analysis

The first step in PropCov is to perform a static analysis on the SUT combined with the property, and custom generator if one exists. The goal of the analysis step is to generate a callgraph that represents the maximum feasible coverage possible for the property under analysis. Although most Java code (and JVM bytecode) is straightforward to analyze, there are some challenging language features, most notably: virtual method invocation, and lambdas. Figure 4 shows why virtual method invocation is challenging, inspired by the examples we have been following. Note that Line 5 in Figure 1 invokes the interface method `serialize`, with no body, defined in Line 1. Figure 2 shows 6 possible concrete implementations of method `serialize`: 2 explicitly for classes `IntListSerial` and `CharSerializer` (Lines 14 and 28), and 3 implicit implementations for each other class listed (Lines 25–27). The challenge for a static analysis building the callgraph is to **resolve the method**: choose which call is feasible from each call site. For instance, the only feasible concrete implementation of method `serialize` reachable from Line 5 is `IntListSerial`. However, Line 20 can reach all other concrete implementations *except* `IntListSerial`.

3.2.1 Supported Analyses. PropCov can be configured with many method resolution algorithms as implemented by the `OPAL` JVM bytecode analysis framework [16, 17, 24]: CHA, RTA, and k-CFA. **Class Hierarchy Analysis (CHA)** provides the simplest possible solution and considers



(a) CHA (b) RTA (c) k-CFA

Fig. 4. Possible callgraphs with different resolutions for virtual calls, using different static analysis algorithms, for the code shown in Figures 1 and 2. CHA (4a) considers as valid all possible concrete methods (including `IntListSerial` on Line 26). RTA (4b) removes unfeasible calls from the call on Line 5 and on Line 20, but cannot eliminate unfeasible calls to `Iterator.next`. k-CFA (4c) produces a minimal feasible callgraph. Nodes represent methods, and the colors mean: test nodes (purple), covered nodes (shades of green, according to coverage), uncovered nodes (red), and virtual method nodes (yellow).

all known concrete implementations as feasible. Figure 4a shows it always adds all known concrete implementations of each virtual method, including unfeasible methods such as reaching `IntListSerial.serialize` from Line 20. **Rapid Type Analysis (RTA)** tracks class instantiation and considers only concrete classes observed to create new objects. Figure 4b shows that RTA accurately resolves the virtual call on Line 5 because it can track the instantiation of the receiver on Line 4. However, RTA can behave as poorly as CHA when it does not have information about possible instantiations, as Figure 4b shows for the (implicit) call to `Iterator.next` on Line 16. **k-CFA** perform sophisticated points-to analysis with increasing context sensitivity. `OPAL` provides support for 0-CFA and 1-CFA, and these are the most accurate algorithms. Figure 4c shows that k-CFA, for this particular example, is able to resolve the call to `Iterator.next` on Line 16 to the only feasible implementation `LinkedList.next`, thus resulting in the highest accuracy.

3.2.2 Accurate support for custom generators. The ability for developers to write custom data generators is a powerful feature of PBT. Figure 1 shows such a generator on Lines 7–10, which generates inputs passed to the property test via the argument `target` on Line 3. Custom generators are not limited to primitive types and Java collections as shown in the example. In fact, custom generators can generate complex custom types defined in the SUT. Given that it is the PBT framework itself (*e.g.*, `junit-quickcheck`) that calls the generator, there is no direct connection between the two in the code that `OPAL` analyzes, which in turn leads to poor quality results for RTA and k-CFA.

For instance, capsyl [39] defines a property to check if two UUIDs are the same, together with a custom UUID generator. When analyzing such property test, which takes UUIDs as arguments, both RTA and k-CFA fail to connect those arguments with the PBT framework code that runs the custom generator to instantiate them. As a result, RTA and k-CFA assume the UUID arguments to be `null` and terminate the analysis with disappointing results at the first use of such arguments. Note that solving this problem is not as straightforward as simply adding the code of the PBT framework itself to `OPAL`'s analyses, as such code relies on reflection which is not supported by `OPAL` [16, 17, 24].

PropCov supports custom generators by configuring OPAL's RTA and k-CFA analyses to *"assume"* that certain types are already instantiated with unbounded values before the method defined as the entry point (*i.e.,* the property test) starts executing. This is equivalent to declaring such values as *symbolic* in symbolic execution [9, 10, 19]. PropCov adds all custom types passed as arguments (*e.g.,* `UUID`) to the set of instantiated types, and the custom types those types define (*e.g.,* `UUID`'s fields), recursively until reaching primitive types, or types defined outside the SUT (*e.g.,* libraries or the Java framework). For completeness, PropCov traverses the class hierarchy repeating the search for all fields declared in all parent classes and implemented interfaces.

3.3 Optimizations

In the extensive evaluation of PBT we describe in Section 4, we found common patterns that reduced the accuracy of the analyses PropCov uses. In this section, we describe optimizations that PropCov uses to reduce such inaccuracies.

3.3.1 External code. The static analysis stage considers all the code inside the packaged System Under Test (SUT), including library code that the SUT imports and uses. However, SUT developers do not write the libraries that the SUT uses and are not interested in the coverage of the libraries' code. As such, PropCov does not consider any external code.

Our design decision is pragmatic and allows PropCov to scale to large projects with many dependencies, and is similar to how JaCoCo also does not consider any code external to the SUT. However, discarding external code means that PropCov cannot reason accurately about rare cases where a library can call back into the SUT, which leads to PropCov missing reachable code. In practice, we found such cases to be rare (Section 4.1.1).

3.3.2 `invokedynamic` *and Java lambdas.* The JVM supports dynamic language features via `invokedynamic` [45]. For instance, Java uses `invokedynamic` to implement lambdas and stream operations. Java lambdas are typically defined inside the SUT but then passed to an external library. For instance, consider the following example using Java streams:

```
// Inside method m1, method m2 is part of the SUT
list.stream().forEach(i -> m2(i));
```

The main goal of the example above is to call method `m2` on each element in the provided list. Developers that write such code are not interested in understanding how the inner workings of method `forEach` eventually leads to calling the provided lambda, and PropCov discards the implementation of `forEach` when performing its analysis. However, we believe that developers are interested in the much simpler fact that method `m1` calls method `m2`. Unfortunately, PropCov's handling of external code also discards such a fact.

The prevalence of Java lambdas requires PropCov to handle them carefully. Instead of just discarding all the information, PropCov leverages `OPAL` to recover the fact that the code in the SUT (*e.g.,* `m1` in our example) calls the lambda also in the SUT, even if indirectly via third party code. `OPAL` rewrites `invokedynamic` invocations to resemble static invocations that create a lambda object and then invoke it [40]. PropCov lists all paths that contain such rewritten lambda invocations, and simplifies them by adding edges from the original method (`m1`) to each edge departing the rewritten static invocation (`m2`). Even though it now looks like that the lambda can be called directly from the code that defines it, we believe it captures the original intention of the developer and is much simpler than the whole path through external code unfamiliar (and even unknown) to the SUT developer.

3.3.3 Exceptions. In our experience, most exceptions thrown while performing PBT mean that the test found a bug and should terminate. For instance, consider the code on the left-hand side of

```
29: void write(List objs) throws Exception
30:   if (limit > 0 && objs.size() > limit) {
31:       Log.getLog().logError(...);
32:       throw new Exception(...);        }
33:   // rest of method
```

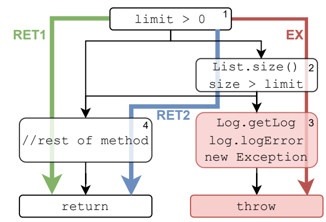


Fig. 5. Possible paths through code that throws exceptions.

Figure 5, adapted from mph-table [28]. In this example, throwing the exception on Line 32 means that this method was called on an invalid list of objects. Successful PBT executions *should never* cover Lines 31–32. However, PropCov considers such lines as reachable and always reports them as uncovered, which leads to a catch-22 report: these lines are not covered, and covering them requires a bug which then terminates PBT execution before reaching full coverage. We experienced such nonsensical reports as annoying that eroded our own trust in PropCov's reports, and designed an optimization to remove such code from reports.

The right-hand side of Figure 5 shows how PropCov skips code that throws exceptions. First, PropCov builds the control-flow graph (CFG) of basic blocks (BBs) for each method. Then, it lists all possible paths through the method, which end in either `return` or `throw`. Our example has 4 BBs and 3 paths: RET1, RET2, and EX. PropCov then lists all BBs only on the paths that end in `throw`: BB3 in our example. Finally, PropCov removes all methods/lines from the listed BBs from the results. As a result, PropCov reports method `write` as fully covered by a property test that covers Lines 30 (including all subexpressions in the && expression) and 33.

3.3.4 Iterable/Iterator. While evaluating PropCov, we noticed that projects with custom implementations of iterable and iterators caused RTA to lose precision when *any* iterator or iterable was used in the project, as shown in Figure 4b. As such, PropCov's iterable/iterator optimization simply removes all calls to iterable and iterator.

3.4 Combining analysis and coverage

PropCov combines the results of the static analysis with the actual coverage gathered by JaCoCo when executing each property test. To provide recommendations to the developer about where to focus to improve a particular property test, PropCov computes a *score* per node in the call graph. The intuition behind the scoring is inspired from our own use of an early version of PropCov, when we noticed that the most interesting nodes have two properties: (1) they are close to the entry-point (*i.e.,* the property test itself), and (2) they can reach a large sub graph. Property 1 means that developers can connect the generator and the test readily to a particular line or method not being covered. Understanding how to connect the entry point to code far away from it (*i.e.,* needing a long sequence of many method calls to reach) is challenging, and we found that shallow uncovered nodes already lead to improvements in code coverage. We believe such a challenge can be best approached with automated techniques (*e.g.,* symbolic execution [9, 10, 19] or fuzzing [42, 50]), and we leave the integration of PropCov with such techniques as exciting future work, out of the scope of this paper. Property 2 maximizes the impact of reaching an uncovered line, as it "controls" a large portion of the code that was not reached before.

PropCov uses the following algorithm to compute the score of each node in the call graph, which can be a method or a line of code. Considering leaf nodes first, the score is either 0 for covered nodes, or 1 for nodes not covered. First, PropCov sets the score of all covered nodes to 0, and all uncovered nodes to 1. Then, PropCov traverses the graph in Depth First Search (DFS) order, starting from the deepest nodes and working its way back to the entry point. Leaf nodes retain their score. Branch nodes (parents) sum their original score with the score of each child — thus following Property 2 above — multiplied by 50% — thus implementing a decay function to follow Property 1 above. Once the algorithm finishes, PropCov lists all the paths in the graph from the entry-point to each uncovered leaf node, and computes the score of such path by simply summing the score of all methods on that path. Finally, PropCov suggests up to N paths to the developer. We found that N=3 works well in practice and used it for our evaluation, which we describe in Section 4.

Finally, PropCov presents a report as a colored method call graph similar to Figure 4, and visually highlights the N paths suggested, so that the developer can the follow each path from the property test to an uncovered red node (not shown in Figure 4).

3.5 Limitations

The current implementation of PropCov works only for projects that use Maven [6] as the build system. This is an engineering limitation, and we leave adding other build systems, such as Gradle [22], as future work.

Due to the pruning approach described in Section 3.3.1, PropCov misses method calls from overridden methods in parent classes outside of the SUT. For instance, the program below:

```
class Outside                { void a();        void b() { a(); ... } } // Outside the SUT
class Inside extends Outside { void a() { ... }                      } //  Inside the SUT
```

Class `Inside`, part of the SUT, extends class `Outside`, outside of the SUT. Both classes implement method `a`, called by method `Outside.b`. In this case, a property test that calls `Outside.b` can reach method `Inside.a`. Unfortunately, PropCov removes all code outside the SUT, thus missing this case and (incorrectly) considering method `Inside.a` as unreachable.

Some PBT frameworks, such as `junit-quickcheck`, allow developers to annotate methods that should run before/after each property test/class. PropCov does not support such annotations and misses all fields initialized this way, resulting in analyses RTA and k-CFA considering such fields as `null` and dropping all method calls that use them as the receiver object.

Some property tests only exercise other parts of the tests and no portion of the SUT. For instance, project `wires` [13] uses a property test to ensure that a custom generator outputs diverse values. PropCov does not consider the coverage of tests, and returns a trivially false result in such cases (*e.g.*, no code found).

4 Experimental Evaluation

In this section we measure the accuracy of PropCov in terms of **Missed Reachable (MR)** code — identified as *not reachable* but that is actually reachable — and **Unfeasible Reachable (UR)** code — identified as *reachable* that is impossible to reach. We answer the following Research Questions (RQs), grouped into: measuring PropCov's accuracy (RQs 1–5), and using PropCov to conduct a novel study on the use of PBT (RQs 6–8):

RQ1 What is PropCov's Missed Reachable (MR) rate?
RQ2 What is PropCov's Unfeasible Reachable (UR) rate?
RQ3 What is JaCoCo's Unfeasible Reachable (UR) rate?

Table 1. Projects used in this paper. This table lists the 25 projects we consider, totaling 293 properties, and describes each project in terms of: ★ stars, ⑂ forks, 👤 contributors, ↺ number of commits, **L**ines **O**f **C**ode, and number of **P**ropertie**S**.

Project	**Description**	★	⑂	👤	↺	**LOC**	**Props**
3d-bin	Geometric computation	392	108	13	1018	19110	3
beam-zstd	Compression	0	0	1	9	296	1
browserstack	Website testing	13	50	25	387	5047	1
capsyl	Code visualization	3	0	1	133	1778	1
convex	Internet of value	83	27	19	4106	49129	3
dyn-shield	Reinforcement learning	5	1	1	4	775	1
funcj	Functional data structures	99	10	4	739	30052	55
HprofCrawler	Heap dump tool	2	0	1	46	3412	11
im-collections	Immutable collections	0	0	2	44	1808	38
ipresource	IP resources	15	13	13	319	4388	1
JESA	Event sourced aggregates	9	4	3	39	691	1
jflex	Lexer	574	111	19	2109	617162	45
jLens	Lenses for Java fields	0	0	1	3	298	9
libosu	Game API	1	1	1	63	4687	4
logish	Logical programming	0	0	3	50	4057	2
manu	Time series storage	3	0	1	140	5869	13
mph-table	Hash table	95	19	7	55	5447	8
NexappCore	REST API collection	3	3	5	95	1495	1
occurrent	Event Sourcing Library	149	17	8	1403	26181	1
paillier4j	Cryptography	3	0	2	19	423	6
rdapd	Network API	8	3	8	586	7694	1
rpki-commons	Resource certification	13	10	15	1269	20652	6
sinch-sms	SMS/MMS API	3	5	12	542	27036	20
stoa	Array utilities	0	0	2	7	6331	38
wires	Circuit simulator	1	1	1	345	4969	23

RQ4 What is the trade-off in graph size versus accuracy between each type of analysis that PropCov supports?
RQ5 How do PropCov's optimizations impact the results?
RQ6 What is the existing coverage of PBT?
RQ7 Does simply increasing the trials increase coverage? What is the time trade-off?
RQ8 Can we use PropCov to improve the quality of existing PBT?

We found existing Java projects that use PBT on Github, searching for repositories that include Java sources that match the string `junit.quickcheck`. We found a total of 348 projects. The vast majority of these projects, 219, contain code used by students in academic classes on testing, which we discarded as the scope of this paper is not on how developers learn to use PBT. We also discarded 23 projects that include the source or modify `junit-quickcheck` itself, or use parametric fuzzing via JQF [31, 41]. We further discarded 36 projects that are just forks of other projects already included. The current prototype of PropCov requires Maven [6], so we discarded 30 that use other build systems (*e.g.*, Gradle [22]). We discarded 7 that do not use `junit-quickcheck` and 9 that do not build, or have failing property tests. To show that PropCov can be used with other PBT frameworks, we added support for `jqwik` [1] and included one project that uses it: occurrent. In total, we considered the 25 shown in Table 1.

We ran all experiments on a machine equipped with 4 sockets, each holding an Intel(R) Xeon(R) Gold 5318H CPU (18 physical cores) and 194GB of RAM, running Ubuntu 22.04.3 LTS. We limited each experiment to a single socket.

Table 2. Missed Reachable (MR) and Unfeasible reachable (UR) statistics for PropCov and JaCoCo. The Lines and Methods columns report average counts across all properties for each project, with relative percentage shown in parenthesis. The final row reports an average over all the properties and projects, rather than an average of the rows above. By definition, JaCoCo's MR is always zero and is therefore omitted. PropCov's UR is further classified as limitations of either the underlying analysis (A) or PropCov itself (P), with the UR (A + P) column reporting per project counts whose sum equals the total number of properties with limitations.

Project	PropCov					JaCoCo			
	Coverage		MR		UR	Coverage		UR	
	Lines	Method	Lines	Method	(A + P)	Lines	Method	Lines	Method
3d-bin	317 (75%)	101 (81%)	1551 (12%)	381 (20%)	0+3	1868 (14%)	482 (25%)	11365 (87%)	1454 (77%)
beam-zstd	24 (77%)	5 (100%)	0 (0%)	0 (0%)	0+0	24 (51%)	5 (50%)	16 (34%)	5 (50%)
browserstack	25 (68%)	6 (86%)	0 (0%)	0 (0%)	0+0	25 (1%)	6 (1%)	2180 (98%)	433 (98%)
capsyl	1 (100%)	1 (100%)	3 (1%)	1 (1%)	0+0	4 (1%)	2 (1%)	314 (99%)	165 (99%)
convex	467 (14%)	178 (15%)	369 (1%)	121 (3%)	0+0	836 (3%)	299 (6%)	21285 (86%)	3807 (79%)
dyn-shield	7 (2%)	2 (5%)	8 (2%)	3 (4%)	0+0	15 (4%)	5 (7%)	38 (9%)	24 (34%)
funcj	210 (92%)	150 (74%)	60 (1%)	41 (1%)	1+0	270 (3%)	191 (5%)	8520 (97%)	4025 (96%)
hprofcrawler	83 (84%)	28 (93%)	48 (3%)	6 (2%)	0+0	132 (9%)	34 (10%)	1398 (91%)	293 (90%)
im-collections	429 (94%)	131 (89%)	80 (14%)	56 (26%)	0+3	509 (87%)	187 (86%)	115 (20%)	80 (37%)
ipresource	71 (61%)	28 (85%)	92 (9%)	30 (11%)	0+0	163 (16%)	58 (21%)	833 (80%)	217 (77%)
JESA	20 (91%)	7 (100%)	0 (0%)	0 (0%)	0+0	20 (22%)	7 (25%)	71 (76%)	21 (75%)
jflex	352 (85%)	67 (94%)	68 (1%)	18 (3%)	0+0	420 (7%)	85 (12%)	5673 (93%)	630 (89%)
jlens	45 (96%)	19 (79%)	3 (5%)	13 (35%)	0+0	48 (75%)	32 (86%)	17 (27%)	8 (22%)
libosu	76 (95%)	31 (86%)	16 (1%)	6 (1%)	0+0	92 (5%)	37 (9%)	1699 (95%)	395 (93%)
logish	86 (84%)	21 (81%)	28 (2%)	9 (2%)	0+0	114 (6%)	30 (5%)	1689 (93%)	556 (94%)
manu	416 (88%)	69 (97%)	234 (14%)	30 (13%)	0+0	652 (38%)	99 (44%)	1237 (72%)	156 (69%)
mph-table	78 (53%)	34 (87%)	11 (1%)	8 (2%)	0+0	89 (5%)	42 (9%)	1692 (92%)	412 (91%)
nexappcore	0 (0%)	0 (0%)	8 (4%)	3 (3%)	0+1	8 (4%)	3 (3%)	199 (96%)	89 (97%)
occurrent	17 (50%)	4 (100%)	0 (0%)	0 (0%)	0+0	17 (11%)	4 (6%)	116 (77%)	60 (94%)
paillier4j	40 (93%)	24 (100%)	39 (37%)	6 (15%)	0+0	79 (76%)	30 (73%)	34 (33%)	13 (32%)
rdapd	1 (33%)	1 (100%)	102 (5%)	16 (2%)	0+0	103 (5%)	17 (3%)	1935 (95%)	659 (97%)
rpki-commons	1290 (75%)	365 (96%)	109 (2%)	45 (3%)	0+0	1399 (26%)	410 (27%)	3658 (68%)	1124 (74%)
sinch-sms	452 (90%)	176 (99%)	159 (2%)	85 (4%)	0+0	611 (8%)	261 (11%)	6850 (93%)	2118 (92%)
stoa	847 (82%)	190 (95%)	325 (17%)	113 (24%)	8+0	1175 (60%)	303 (63%)	909 (47%)	279 (58%)
wires	31 (72%)	11 (55%)	12 (1%)	7 (2%)	0+6	43 (5%)	18 (4%)	901 (94%)	383 (94%)
	215 (55%)	66 (58%)	133 (4%)	40 (5%)	9+13	349 (10%)	106 (13%)	2910 (86%)	696 (85%)

Unless otherwise stated, the results use PropCov's RTA support and all optimization techniques enabled (Section 3.3). We use RTA as default because, as this section shows, RTA provides a good compromise between accuracy and time.

4.1 PropCov's Accuracy

4.1.1 PropCov's MR. The first experiment measures how many lines PropCov considered not reachable that are indeed reachable: Missed Reachable (MR). The **ground-truth** is the coverage reports produced by JaCoCo. We consider as MR lines that JaCoCo observes to be reached during PBT execution but PropCov does not include in its reports. Such lines are cases where PropCov's analysis under-approximates the SUT. Table 2 shows the result.

Answer to RQ1: PropCov misses only 4% lines of code that are actually reachable (5% of all the methods). Developers can trust PropCov's results to miss little, if any, reachable code.

4.1.2 PropCov's UR. In this experiment, we measure how PropCov over-approximates the SUT by counting how many properties have reachable (but not reached) nodes that are in fact unfeasible: Unfeasible Reachable (UR). We ran this experiment using 1-CFA instead of RTA to ensure maximum accuracy. Unfortunately, this experiment does not have a factual source of **ground-truth**. Instead, we manually analyzed all the reports with unreached nodes that PropCov produced, for a total of 165 properties. For each unreached node, we executed the SUT inside the debugger to understand

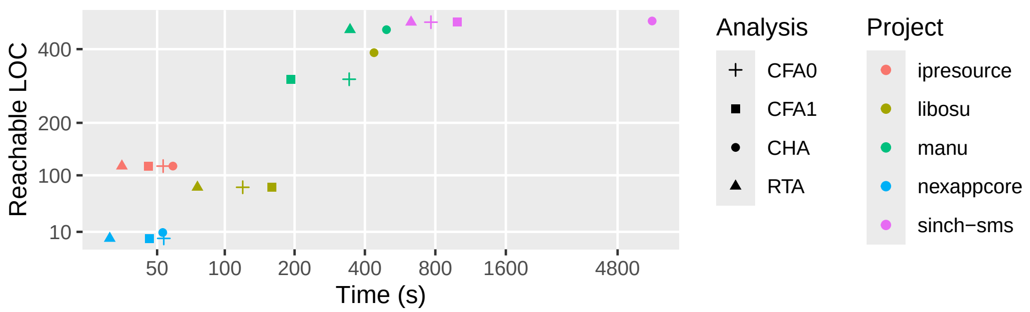


Fig. 6. Size of generated graph versus time taken to generate the graph per analysis. All missing projects fall within one of the 5 representative projects shown.

how the node could be reached. For nodes we believe are not feasible, we further classify them as a limitation of: the underlying analysis (*e.g.*, a value that is always `null` but the analysis does not track nullness), or PropCov itself (*e.g.*, an field inherited from outside of the SUT that the SUT itself reads/writes, as explained in Section 3.5). Table 2 shows the results.

We can see that most projects (19) have zero UR. Unreached nodes in these projects *can* be reached, but the current test and/or generator are not able to do so. 1-CFA struggled with *funcj* — it connected an application of a lambda to an (incorrect) `toString` method — and *stoa* — 8 tests rely on Java reflection, which is not supported. PropCov's limitations lead to UR on tests that target only the generator — *3d-bin* and *wires* — and tests that rely on fields inherited from classes outside the SUT — *im-collections*, *nexappcore*, and *stoa*.

Answer to RQ2: When using PropCov, we found UR on 22 properties (7.5% rate) over 6 projects. These break down to 9 properties (3%) due to limitations of the underlying analysis, and 13 properties (4%) due to limitations of PropCov itself. Developers can trust that code reported by PropCov is indeed reachable by the property being tested.

4.1.3 JaCoCo's UR. We used PropCov's results to measure how much unfeasible code JaCoCo's results show as reachable. We use as **ground-truth** the results of PropCov that we manually validated (Section 4.1.2) and count all the lines and methods that JaCoCo reports as not covered but that PropCov reports as unreachable from the original property. Table 2 shows the results.

Answer to RQ3: The vast majority of code in JaCoCo's reports (86% lines over 85% methods) are UR, which causes developers to consider 2910 lines over 696 methods that cannot be possibly reached from any property. These results show that JaCoCo is not suitable for PBT, as we discuss in Section 2.1.

4.1.4 PropCov's different analyses. PropCov uses `OPAL`'s analyses to compute the reachability from each property test. In this section, we show the impact of considering each analysis — CHA, RTA, 0-CFA, and 1-CFA. We measure the impact of each analyses in terms of how much more code it includes, and time to complete. We manually clustered similar results, and Figure 6 shows a representative project of each 5 clusters:

1. k-CFA smallest. k-CFA removes more unfeasible nodes, resulting in a smaller graph produced faster than all other analyses: convex, funcj, logish, **manu**, rdapd, and stoa.

2. RTA smallest. RTA's simplicity works to its advantage, resulting in smaller graphs produced faster than all other analyses: im-collections, jflex, and **sinch-sms**.

Table 3. Nodes added by λ functions and removed by the **I**terator and **E**xception pruning strategies from the original graph — $\varnothing$. We omit: CHA, as it is very inaccurate, 1-CFA, as it is similar to 0-CFA, and 11 projects (43 properties) without change. This table reports how many properties do not change (**0**), all properties of that project (**n**), and the average node change per project.

Project	RTA						0-CFA					
	0/n	$\varnothing$	λ	λ, I	λ, E	λ, I, E	**0/n**	$\varnothing$	λ	λ, I	λ, E	λ, I, E
convex	0/3	1228	+10	−38	−24	−62	0/3	973	+1	−15	−23	−38
dyn-shield	0/1	11	+32	—	—	—	0/1	10	+33	—	—	—
funcj	23/55	110	+94	—	—	—	32/55	99	+88	—	—	—
hprofcrawler	11/11	26	+4	—	—	—	11/11	26	+4	—	—	—
im-collections	26/38	158	+22	−25	—	−25	29/38	142	+27	−23	—	−23
jflex	30/45	79	—	−6	—	−6	23/45	38	—	−6	—	−6
jlens	2/9	17	+11	—	—	—	2/9	17	+7	—	—	—
libosu	2/4	23	+13	—	—	—	2/4	13	+18	—	—	—
logish	0/2	68	+4	−7	-1	−9	0/2	17	+4	—	—	—
manu	9/13	102	—	−15	−12	−31	9/13	60	—	−6	−2	−8
rdapd	0/1	1	+4	—	—	—	1/1	1	—	—	—	—
sinch-sms	12/20	197	—	−16	−7	−19	14/20	197	—	−16	−7	−19
stoa	8/38	52	+246	−9	—	−9	9/38	43	+145	−5	−2	−7

3. RTA and k-CFA tied for size but RTA is faster. RTA achieves similar graph sizes to k-CFA, and CHA clearly worse on both dimensions: HprofCrawler, **libosu**, mph-table, and rpki-commons.

4. All have same size and RTA is fastest. The underlying project uses virtual methods with only one concrete implementation, and the small difference in time is due to inefficiencies in the implementation of each analysis: beam-zstd, browserstack, capsyl, dyn-shield, **ipresource**, JESA, jLens, libosu, paillier4j, and wires.

5. k-CFA and RTA fail. Limitations of PropCov (Section 3.5) cause k-CFA and RTA to not find any SUT code. CHA simply lists *all* possible concrete implementations, and is thus able to find *some* SUT code: 3d-bin, **nexappcore**, and occurrent.

We also computed the MR rate for each analysis using the same methodology as Section 4.1.1. 20 projects have the same MR rate for RTA and k-CFA. Surprisingly, k-CFA did not minimize the MR rate for any project, which means that the analysis, or how PropCov configures it (Section 3.2.2), under-approximates the SUT and discards nodes that can be reached. For the remainder 7 projects, RTA has the smallest MR rate. Compared to Table 2, 3 projects have slightly lower MR — 3d-bin (5%), convex (1%), and jflex (3%) — and the remaining 4 have a significant difference — im-collections (31%), rdapd (14%), rpki (23%), and stoa (61%).

Answer to RQ4: RTA is always the fastest in 15 projects: for 11 projects, all analyses result in the same graph size, and for 4 project RTA and k-CFA generate similar results. RTA generates the smallest graph for 5 projects, and k-CFA for other 5 projects. RTA also always generates graphs with the lowest MR. Our results show that RTA is the preferred analysis as it provides the fastest best results for most projects (20). CHA never produces small or fast results.

4.1.5 Impact of PropCov's optimizations. Section 3.3 describes PropCov's optimizations for common language features: Java lambdas, iterators/iterable, and exceptions. We measured the impact of these optimizations on all projects by rerunning PropCov with different configurations: no optimizations ($\varnothing$), Java lambda (λ), iterators/iterable (I) and exceptions (E) independently but together with λ, and all optimizations. Table 3 shows the results for RTA and 0-CFA. We omit the results for 1-CFA, which are similar to 0-CFA, and CHA, which are much worse than RTA.

Each optimization needs to reach code with their target feature, and no optimization triggered for 125/156 properties for RTA/k-CFA. Most optimizations have more impact on RTA than 0-CFA, possibly due to 0-CFA's higher accuracy [16, 17, 24]. Note that combining optimizations is not linear, as two optimizations can remove the same code (*e.g.,* a custom iterator catching exceptions). Finally, we also computed the MR for each configuration using the same methodology as Section 4.1.1. The λ optimization included missing reached code, the **I** and **E** optimizations removed reached code on projects that use custom iterators/iterable or use exception handling for non-terminal errors, respectively. As a result, in average, λ reduced the MR by 45.6 LOC and 9.5 methods, **I** increased it by 40.2 LOC and 9.2 methods, and **E** increased it by 28.6 LOC and 7.6 methods.

Answer to RQ5: PropCov's optimizations change the results of 117/108 (for RTA/0-CFA) properties by adding 440/327 feasible nodes reached through lambdas, removing 161/101 unfeasible nodes due to the analyses over-approximation of iterators/iterable, and ignoring exception handling nodes. On average PropCov's optimizations find 33.9/25.2 more feasible nodes, and remove 12.4/7.8 unfeasible nodes. The optimizations have no benefit for 13 projects, because such projects do not use any target feature. The more accurate analyses k-CFA benefit less from the optimizations as they are more accurate to start with. The λ optimization always increases accuracy, reducing the MR by 45.6 LOC in average. The **I** and **E** optimizations drop uninteresting code that is reached, resulting in a modest MR increase of 40.2 and 28.6 LOC, respectively.

4.2 Study of existing PBT

4.2.1 Coverage of existing PBT. PropCov provides a novel insight into the coverage of existing PBT. We start by comparing its coverage results per property against JaCoCo by running each tool on each property. Table 2 shows the results.

Answer to RQ6: PropCov reports that, for existing PBT, 55% lines and 58% methods are covered. JaCoCo reports 10% lines and 13% methods. PropCov's results show that there is room to improve PBT and that JaCoCo is not suitable for PBT as it reports inaccurate coverage. PropCov is the first tool that reports accurate PBT coverage to developers.

4.2.2 Impact of the number of trials. Given the room for improvement described in Section 4.2.1, developers can simply increase the number of trials in PBT. We used PropCov to gain novel insight into how trials impact PBT coverage by varying the number of trials, and measuring the coverage and time per property. We ran each property for each number of trials 2 times, and clustered the results according to maximum line coverage, increase in duration (over 2x when compared to 1 trial). Table 4 shows the results.

We found 85 tests reaching full coverage: 60 with 1 trial, 22 with 10, 2 with 100, and 1 with 1000. Inspecting the tests with 1 trial reveals straight line code with three possible explanations. **First**, these tests are very simple and exercise the SUT in a trivial way. For instance, calling a SUT method that returns an input lambda wrapped in another lambda, which then the test uses more sophisticated logic to ensure the wrapped lambda behaves as expected. **Second**, the tests sample a large input space. For instance, a SUT method that turns longs into byte arrays has straight line code, but it makes sense to write a property test to sample the space of inputs and ensure that it behaves as expected. Finally, **third**, the tests involve either loops executed at least once or Java streams. In both cases, at least one iteration results in full line coverage. PropCov can be adapted to use better notions of coverage (*e.g.,* path coverage, either static from JaCoCo or dynamic as fuzzers use [42, 50]), and we leave such extension as exciting future work.

For all tests, we did not see an increase in time taken to run each property until 100 trials. Increasing the trials to 500 or more resulted in longer testing times for 36 tests, and only 12 tests

Table 4. Impact of varying the number of trials per line coverage and time, clustered per number of trials to reach maximum coverage. This table does not include 43 tests because they require more than 10 minutes to run 10 or more trials, or because they fail at low numbers of trials (*e.g.,* ensuring a generator has at least a 1% chance of generating a particular value).

Cluster	n	**Coverage per trials**					**Time (s) per trials**				
		1	10	100	500	1000	1	10	100	500	1000
100% at 1; Same Time	46	100%	100%	100%	100%	100%	12.66	12.75	13.07	14.31	26.5
100% at 1; Time Increases	14	100%	100%	100%	100%	100%	11.27	11.37	11.49	12.5	24.21
Max at 1; Same Time	50	87%	87%	86%	87%	85%	13.76	13.87	14.3	15.2	24.46
Max at 1; Time Increases	3	82%	79%	82%	79%	79%	12.1	12.07	12.32	13.83	34.36
100% at 10; Same Time	22	50%	100%	100%	100%	100%	10.17	10.18	10.43	10.91	14.91
Max at 10; Same Time	48	67%	89%	86%	86%	87%	17.09	17.1	18.05	18.78	29.44
Max at 10; Time Increases	13	58%	89%	85%	85%	87%	12.65	12.99	14.07	43.69	157.26
100% at 100; Same Time	2	74%	95%	100%	100%	100%	15.64	15.16	16.48	15.98	16.5
Max at 100; Same Time	22	58%	75%	84%	76%	82%	11.59	11.65	12.21	17.88	53.02
Max at 100; Time Increases	8	66%	74%	85%	84%	84%	8.95	9.2	9.64	17.71	79.48
Max at 500; Same Time	15	60%	81%	82%	89%	85%	11.61	11.69	12.49	15.68	29.03
Max at 500; Time Increases	3	66%	76%	77%	80%	79%	7.77	7.86	8.07	13.64	50.44
100% at 1000; Same Time	1	92%	92%	92%	92%	100%	24.35	23.99	24.74	23.51	24.75
Max at 1000; Same Time	1	43%	57%	57%	57%	62%	9.76	9.73	9.83	9.88	26.12
Max at 1000; Time Increases	1	62%	79%	77%	88%	91%	7.54	7.45	7.56	7.78	20.72

Table 5. Projects improved by following PropCov's suggestions. This table reports coverage as measured by PropCov.

Project	**#**	**Methods**				**LOCs**			
	Props	Total	Before	After	Improv	Total	Before	After	Improv
funcj	1	16	15	16	100%	17	16	17	100%
jflex	7	46	35	46	100%	363	193	297	61.18%
manu	3	40	32	38	75%	261	184	210	33.77%
mph-table	1	27	21	24	50%	120	52	83	45.59%
sinch-sms	4	102	98	102	100%	297	272	284	48%
stoa	10	245	178	193	22.39%	1085	696	776	20.57%
wires	16	20	13	17	57.14%	43	29	37	57.14%
Total	42			Average:	72.08%			Average	52.32%

more coverage. 1000 trials results in even worse results, causing 63 tests to double their original time with only 3 increasing their coverage.

Answer to RQ7: Most tests (219) reached their 100% or maximum coverage with 10 trials. 100 trials further increased the coverage on only 29 tests, without increasing the time required to run the test. 500 or more trials further increased the coverage for only 12 tests, with significant slowdowns for 36 tests with 500 trials, and 63 trials for 1000 trials. Increasing the number of trials alone typically does not lead to increased coverage. We believe that 10 is a good default number of trials for most properties (instead of 100 for `junit-quickcheck`), and that developers should use PropCov to understand if each particular property should use more trials.

4.2.3 Using PropCov with existing PBT. When conducting the manual analysis in Section 4.1.2, we also attempted to improve the quality of existing PBT by changing the generator or the test to reach previously unreached code. As non-specialists in any of the projects, we followed PropCov's recommendations via its novel scoring heuristic (Section 3.4). Table 5 shows the results.

In total, we improved the coverage of 42 tests over 7 projects, and found 4 previously unknown bugs. We opened 9 pull requests, with 5 merged at the time of writing. We also opened 4 issues for the bugs, with 2 confirmed and fixed by the developers. The remaining 4 pull requests and 2 bugs remain open.

Once again, PropCov provides novel insight into PBT: high coverage does not necessarily mean absence of bugs. For instance, funcj started with 92% method and 74% line coverage. Despite such values, improving PBT still revealed a bug, which shows PBT's ability to find subtle bugs in corner-cases even for a modest improvement in overall coverage. Using PropCov, we improved method/line coverage by an average of 72.08%/52.32%, reaching 100% method coverage for 12 properties over 3 projects, and 100% line coverage for 1 test. We found the following bugs:

convex An existing test did not cover the data-type `blob`. We improved the generator, and we found an error for blobs larger than 4KB. This issue was reported, acknowledged, and fixed.

funcj The optional `Either` must short-circuit: `Either.left(x).left(y))` must result in `Either.left(x)`. We added a case to the test that should short-circuit as described, and found that it resulted in `Either.left(y)`. The revealing input requires a particular deletion on empty integer maps, clearly a corner-case that the developers missed. This issue was reported, acknowledged, and fixed.

logish We found `null` checks that always failed when inserting/removing items from a hash map, and improved the tests to also insert/remove `null` items. We found two bugs, one for insertion and one for deletion. These issues are still pending.

manu On a test to roundtrip encoding and decoding IDs, the code to encode small and medium numbers was not covered. We changed the test to generate diverse number sizes, and found one bug which attempts to encode medium-sized IDs with not enough bits, resulting in a different ID being decoded. This issue is still pending.

Answer to RQ8: Developers can use PropCov to understand which property tests require improvement, and then follow PropCov's guidance to improve those tests. As non-specialists, we used PropCov to improve 42 tests over 7 projects, improving method/line coverage by 72.08%/52.32% and finding 4 previously unknown bugs in the process. The original developers confirmed our findings by merging 4 pull-requests and fixing 2 bugs. We also reached 100% method coverage for 12 properties over 4 projects, and 100% line coverage on 1 property.

5 Related Work

Property-based testing (PBT) was first introduced for Haskell with Quickcheck [12]. The authors note how PBT is particularly suited for functional languages, given their lack of mutable state. Since then, Quickcheck has been ported to many other functional languages such as Clojure, Scala, F#, OCaml [14, 25, 38, 46]. In an effort to understand its foundational principles, PBT has been implemented in proof assistants [44] with some success in generating properties automatically [32, 43]. Despite inherent limitations, PBT has also been ported to a number of popular imperative languages such as C, Java, Javascript, and Python [23, 27, 35, 48]. PropCov, on the other hand, targets helping developers in writing better Java properties, and assumes that the original property is correct with regards to mutable state. Tools for checkpoint-rollback [7] can deal with the mutable state, resetting it automatically before each new run. Formal research on parametric properties [8] targets functions that take many different types, common in functional languages. In contrast, PropCov targets virtual calls that may resolve to a number of concrete methods being called depending on the type of the receiver.

Based on a user-study of developers using PBT [20], Tyche [21] provides different visualizations to help developers understand PBT results, including how coverage increases with each fresh trial/input. PropCov provides observed/possible coverage per test, which allow developers to gauge if a particular test already covers the intended functionality, and, if not, PropCov provides actionable suggestions on how to improve such a test.

PBT simply re-executes the same test a fixed number of times, each with different input. Random-testing, fuzz testing, or *fuzzing* [36], also re-executes the same code repeatedly with different random inputs. However, fuzzing is typically guided by code coverage, and employs many techniques to maximize coverage by modifying the random input in particular ways [18, 29, 50]. Fuzzing campaigns are much longer in length than PBT, ranging from hours to days [30], and use unstructured input, unlike PBT. Similarly to PropCov, Fuzz-Inspector [11] approximates the maximum possible coverage via the whole-program call-graph. FuzzInspector targets fuzzing harnesses, and PropCov targets PBT. FuzzInspector considers the whole application instead of a single test, resulting in a coarse approximation of the program behavior that suffers from over- and under-counting [33] as seen in FuzzInspector's database entry for fuzzing `jflex` [2], which reports 38.58% method coverage with 61% MR (156/254). PropCov provides high-quality feedback that guides developers towards code locations that kept PBT from making progress, and reports 94% method coverage with 1% MR for `jflex`.

Another main difference between fuzzing and PBT is that fuzzing does not guarantee a valid input for each execution, which is somewhat useful as it allows fuzzers to detect bugs when processing invalid input, but severely limits the applicability of fuzzing on programs with complex input because all inputs can be expected to be invalid. Some fuzzers get around this limitation by using a sophisticated generator of valid inputs. For instance, Csmith [49] generates random C programs guaranteed to be valid (*i.e.,* syntactically correct and without undefined behavior) to test C compilers. More generally, *parametric fuzzing* [31, 41, 42] uses generators to obtain valid input that is then passed to the Software Under Test (SUT). Each random bit provided is now a parameter that the generator uses, such that different unstructured bits control different parts of the structured output. Given the similarity between parametric fuzzing and PBT, we believe there is exciting future work in adapting PropCov's insights to guide parametric fuzzing.

6 Conclusion

This paper presents a novel mechanism for measuring the coverage of PBT in a meaningful manner on a per-test basis, especially suited for imperative languages such as Java, that combines a static analysis, to compute an approximation of the maximum theoretical coverage of each property test, with coverage-based dynamic analysis. Based on such a novel mechanism, this paper presents the design and implementation of PropCov, an extensible tool to guide developers using PBT, together with an extensive evaluation using 293 properties over 25 Java projects using PBT frameworks `junit-quickcheck` and `jqwik`. PropCov results are accurate — only 7.5% of all properties contain unfeasible code, and PropCov only misses 4% of reachable code. The proposed technique drastically reduces the effort required to identify where to improve existing property tests (or their generators) via a novel heuristic technique that scores program paths according to how close they were to be reached during execution. Based on such insight, PropCov recommends possible program points to improve. Following PropCov's recommendations, we improved the coverage of 46 properties and found 4 bugs, all on projects unfamiliar to us. At the time of writing, 2 bugs were confirmed and fixed by the developers, and we submitted 9 pull requests, 5 merged by the developers (other bugs/PRs are still pending).

7 Data Availability

An anonymous artifact is available as a VM image for x86_64 at: https://doi.org/10.5281/zenodo.17095525. The artifact contains scripts to run all experiments described in the paper, together with the data we used. We ask that the reviewers use the most recent version of the artifact to ensure the most up-to-date contents. We will update the artifact post-submission to include any feedback from the review process. We are committed to replicable science, and assure the reviewers that the main author has a track-record of making research artifacts available with published papers.

References

[1] 2017. jqwik: Property-Based Testing in Java. https://jqwik.net/. Accessed: 2025-03-13.

[2] 2024. FuzzInspector results: jflex. https://storage.googleapis.com/oss-fuzz-introspector/jflex/inspector-report/20251123/fuzz_report.html#functions_cov_hit_0. Accessed: 2025-11-30.

[3] 2024. Haskell Discourse — The sad state of property-based testing libraries. https://discourse.haskell.org/t/the-sad-state-of-property-based-testing-libraries/9880. Accessed: 2025-03-13.

[4] 2024. SuperCollider: Property-Based Testing –– Call for Opinions, Comments. https://scsynth.org/t/property-based-testing-call-for-opinions-comments/9087. Accessed: 2025-03-13.

[5] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, USA.

[6] Apache Foundation. 2002. Apache Maven is a software project management and comprehension tool. https://maven.apache.org/. Accessed: 2025-03-13.

[7] Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:31. https://doi.org/10.4230/LIPIcs.ECOOP.2018.17

[8] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. 2010. Testing Polymorphic Properties. In *Proceedings of the 2010 European Symposium on Programming (ESOP)*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–144.

[9] 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages. https://doi.org/10.1145/1455518.1455522

[11] Oliver Chang, Navid Emamdoost, Adam Korczynski, , and David Korczynski. 2016. Introducing fuzz introspector, an openssf tool to improve fuzzing coverage. https://openssf.org/blog/2022/06/09/introducing-fuzz-introspector-an-openssf-tool-to-improve-fuzzing-coverage/. Accessed: 2023-07-05.

[12] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[13] Laurent Claisse. 2017. A Java port of the "Simulator for digital circuits" from SICP (Structure and Interpreation of Computer Programs). https://github.com/vandekeiser/wires. Accessed: 2025-03-13.

[14] Simon Cruanes. 2023. qcheck - QuickCheck inspired property-based testing for OCaml. https://github.com/c-cube/qcheck. Accessed: 2023-07-05.

[15] Colin Dellow. 2017. manu: A time series storage format for integers and floats, using efficient delta encodings from FastPFOR. https://github.com/cldellow/manu. Accessed: 2025-03-13.

[16] Michael Eichberg and Ben Hermann. 2014. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614630

[17] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini. 2018. Lattice based modularization of static analyses. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 113–118. https://doi.org/10.1145/3236454.3236509

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[20] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[21] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology, UIST 2024, Pittsburgh, PA, USA, October 13-16, 2024*, Lining Yao, Mayank Goel, Alexandra Ion, and Pedro Lopes (Eds.). ACM, 10:1–10:16. https://doi.org/10.1145/3654777.3676407

[22] Gradle Inc. 2008. Gradle Build Tool. https://gradle.org/. Accessed: 2025-03-13.

[23] Oleg Grenrus. 2021. JSVerify - property based testing for JavaScript. https://jsverify.github.io/. Accessed: 2023-07-05.

[24] Dominik Helm, Tobias Roth, Sven Keidel, Michael Reif, and Mira Mezini. 2024. Unimocg: Modular Call-Graph Algorithms for Consistent Handling of Language Features. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/3650212.3652109

[25] Rich Hickey and Reid Draper. 2023. test.check - QuickCheck for Clojure. https://github.com/clojure/test.check. Accessed: 2023-07-05.

[26] Marc R. Hoffmann. 2009. EclEmma - JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/. Accessed: 2023-07-05.

[27] Paul Holser. 2023. Property-based testing, JUnit-style. https://github.com/pholser/junit-quickcheck. Accessed: 2023-07-05.

[28] Indeed Corp. 2018. Indeed MPH: Fast and Compact Immutable Key-Value Stores. https://engineering.indeedblog.com/blog/2018/02/indeed-mph/. Accessed: 2023-07-05.

[29] LLVM Compiler Infrastructure. 2016. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer. Accessed: 2023-07-05.

[30] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[31] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 438–450. https://doi.org/10.1145/3510003.3510628

[32] 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

[33] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing.

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[35] David MacIver. 2023. Hypothesis - Test faster, fix more. https://hypothesis.works/. Accessed: 2023-07-05.

[36] Barton P. Miller, Lars Fredriksen, 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

[37] Gerard Murphy. 2022. Americium: Generation of test case data for Scala and Java, in the spirit of QuickCheck. https://github.com/sageserpent-open/americium. Accessed: 2025-03-13.

[38] Rickard Nilsson. 2023. ScalaCheck: Property-based testing for Scala. https://scalacheck.org/. Accessed: 2023-07-05.

[39] Jesper Olsson. 2017. capsyl: Visualizes Java objects' encapsulations. https://github.com/jesperolsson-se/capsyl. Accessed: 2025-03-13.

[40] Opal Project. 2021. OPAL Documentation - Facilitating the Development of Static Analyses. https://github.com/opalj/opal/blob/488ec8d4962aeb7cfa9ba2e4af70bfe60c7d50f2/src/site/DeveloperTools.md?plain=1#L72. Accessed: 2025-03-13.

[41] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[42] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[43] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing Correctly with Inductive Relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 966–980. https://doi.org/10.1145/3519939.3523707

[44] Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343.

[45] John Rose, Ola Bini, William R Cook, Rémi Forax, Samuele Pedroni, and Jochen Theodorou. 2011. JSR 292: Supporting Dynamically Typed Languages on the JavaTM Platform. https://jcp.org/ja/jsr/detail?id=292. Accessed: 2025-03-13.

[46] Kurt Schelfthout. 2023. FsCheck: Random Testing for .NET . https://github.com/fscheck/FsCheck. Accessed: 2023-07-05.

[47] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[48] Scott Vokes. 2019. theft: property-based testing for C. https://github.com/silentbicycle/theft. Accessed: 2023-07-05.

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (jun 2011), 283–294. https://doi.org/10.1145/1993316.1993532

[50] Michał Zalewski. 2014. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2023-07-05.