# Mutagen: Coverage-Guided, Property-Based Testing using Exhaustive Structure-Preserving Mutations

Anonymous Author(s)

## Abstract

Automatically synthesized random data generators are an appealing option when using property-based testing. Such generators can be obtained using a variety of techniques that extract static information from the codebase in order to produce random test cases. However, such techniques are not suitable for deriving generators producing random values satisfying complex invariants, which in turn complicates testing properties with sparse preconditions.

Coverage-guided, property-based testing (CGPT) is a recent technique that alleviates this limitation by enhancing automatically synthesized generators with structure-preserving mutations guided by execution traces. Albeit effective, the initial CGPT approach is limited by factors like large reliance on randomness and poor mutants scheduling, reducing its applicability and bug-finding capacity in existing software.

In this work we present Mutagen, a CGPT framework that overcomes these limitations by generating mutants in an *exhaustive* manner. This is coupled with heuristics that help to schedule mutants based on their novelty and to minimize mutating certain subexpressions leading to scalability issues of exhaustiveness. Our results indicate that our approach is capable of outperforming existing CGPT tools, as well as finding bugs in real-world scenarios.

## Keywords

random testing, mutations, heuristics

## 1 Introduction

Popularized by *QuickCheck* [10], Random Property-Based Testing (RPBT) is a popular technique for finding bugs [8, 10, 11, 23, 35]. This is achieved by using randomly generated inputs to instantiate executable properties encoding the expected behavior of the system under test. However, users of *QuickCheck* (and RPBT in general) are well aware that one of the biggest challenges while using RPBT is to provide suitable random data generators needed to instantiate the testing properties. In extreme cases, writing highly-tuned random generators capable of exercising every part of a complex system on a reasonable basis can take several thousand person-hours of trial and error [26]. To alleviate this, there exist several techniques to automatically synthesize random data generators by reifying the static information present in the codebase, e.g., data type definitions, application public interfaces (APIs), etc. [4, 13, 16, 27, 32, 33].

*ICST 2022, Apr 4–13, 2022,*
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

These techniques, however, are unable to synthesize random generators capable of producing data satisfying complex invariants not easily derivable from the codebase. Randomly generating valid programs is a common example of this problem, where developers are forced to put substantial efforts in writing specialized random generators by hand [34, 36, 42]. Moreover, automatically synthesized random generators offer particularly poor results when the testing properties are constrained by sparse preconditions. In this scenario, most of the generated tests are simply thrown away before they can be used to test the main components of the system under test.

Coverage-Guided, Property-Based Testing (CGPT) [26] is a technique that borrows ideas from the fuzzing community to generate highly-structured values while still using automatically derived generators. Instead of continuously generating random invalid test cases from scratch, CGPT works by keeping queues of *interesting* previously executed test cases that can be mutated using high-level, structure-preserving transformations to produce new ones. Intuitively, mutating an interesting test case in a small way (at the data constructor level) has a higher probability of producing a new interesting test case than generating a new one using a naïve generator. In addition, high-level, structure-preserving mutators are better suited for producing syntactically valid mutants. High-level mutators avoid wasting time with generic low-level mutators that work at the bit level and know very little about data structures, thus producing syntactically broken mutants most of the time. Such a grammar-aware mutation technique has shown to be quite useful when fuzzing systems accepting structurally complex inputs [19, 20, 41]. While most of the tools following this itechnique use existing grammars to tailor the mutators to the specific input structure used by the system under test, external grammars are not required in CGPT. The data types of the inputs to the testing properties already provide a good source of information about how data can be mutated, e.g., by changing one data constructor by another. Specialized mutators can be automatically derived directly from data types definitions — making strongly-typed programming languages an ideal match for CGPT.

Moreover, CGPT relies on execution traces to distinguish which executed test cases were interesting and are therefore worth mutating — a popular technique known as *coverage-guided fuzzing* and popularized by tools like *AFL* [30]. Mutated test cases are considered interesting for mutation only when they produce new execution traces — any other test case is simply thrown away.

Worth mentioning, CGPT is not meant to replace sophisticated manually-written random generators, but rather to provide an adequate testing solution for early stages of development.

In this work, we establish several aspects[1] of the original CGPT approach by Lampropoulos et al. that leave considerable room for improvement (see §2). In particular: (1) structure-preserving mutations, if not done carefully, can become superficial and rely

[1]Some of these were already identified by Lampropoulos et al. in their original work.

heavily on randomness to produce diverse mutants, (2) the queuing mechanism can cause delays if interesting values are enqueued frequently and there is no way to prioritize them, and (3) using an heuristic to assign a certain "mutation budget" to each interesting test case (often referred to as a *power schedule*) requires fine tuning and can be hard to generalize to different kinds of testing properties.

In this work, we introduce Mutagen, a CGPT framework that tackles the aforementioned limitations by applying mutations *exhaustively* (see §3). That is, given an interesting, previously executed test case, our tool precomputes and schedules every structure-preserving mutation that can be applied to it. This approach has two inherent advantages. On one hand, mutations do not rely on randomness to be generated. Instead, every subexpression of the input test case is mutated on the same basis, which guarantees that no interesting mutation is omitted due to randomness. On the other hand, scheduling mutations exhaustively eliminates the need for a power schedule to assign a given mutation budget to each input test case.

Our tool distinguishes two kinds of mutations, those that can be computed deterministically, yielding a single mutated value; and those that can be obtained non-deterministically. On one hand, deterministic mutations encode transformations that swap data constructors around, as well as return or rearrange subexpressions. Non-deterministic (random) mutations, on the other hand, are useful to represent transformations over large enumeration types, e.g., numbers, characters, etc. This mechanism let us selectively escape the scalability issues of exhaustiveness by randomly sampling a small random generator a reduced number of times. In this way, our tool avoids, for instance, mutating a given number into every other number in the 32 or 64 bits range.

Additionally, the testing loop of Mutagen incorporates two novel heuristics that help finding bugs faster and more reliably (§4). In the first place, our tool uses first-in first-out (FIFO) scheduling with priority to enqueue interesting test cases for mutation. This scheduling algorithm is indexed by the novelty of each new test case with respect to the ones already seen. In this light, interesting test cases that discover new parts of the code earlier during execution are given a higher priority. Moreover, our scheduling allows the testing loop to jump back and forth between mutable test cases as soon as a new more interesting one is enqueued for mutation, eliminating potential delays when the mutation queues grow large.

The second heuristic adjusts the number of times our tool samples random mutants, i.e., those corresponding to large enumeration types as described above. For this, we keep track of how often we generate interesting test cases. If this frequency suddenly stalls, Mutagen resets the testing loop increasing the amount of times we sample such mutants. This automatically finds a suitable value for this external parameter on the fly.

To validate our ideas, we use two case studies (described in detail in §5): the Information-Flow Control (IFC) stack machine used by Lampropoulos et al. in their original work, and an existing WebAssembly engine of industrial strenght written in the strongly-typed programming language Haskell. In both case studies, we additionally compare the effect of the heuristics implemented on top of the base testing loop of Mutagen. In the IFC stack machine case study, we compare Mutagen against *FuzzChick*, the reference CGPT implementation developed in Coq [26]. Our results (§6) indicate that Mutagen outperforms *FuzzChick* both in terms of time to failure and failure rate. On the other hand, our WebAssembly case study shows the performance of our tool in a realistic scenario. There, Mutagen is capable of reliably finding 15 planted bugs in the validator and interpreter, as well as 3 previously unknown bugs that flew under the radar of the existing unit test suite of this engine. This case study also help us to compare the performance of our tool against a more traditional RPBT approach that does not rely on code instrumentation. All in all, our evaluation encompasses more than 600 hours of computing time and suggests that *testing mutants exhaustively together with our heuristics to escape scalability issues* can be an appealing technique for finding bugs reliably without sacrificing speed.

Finally, we discuss related work in §7 and conclude in §8.

## 2 Background

In this section, we briefly describe the motivation, ideas and limitations behind CGPT [26]. To illustrate this technique, let us focus on a simple property defined over binary trees. Such a data structure can be defined in Haskell using a custom algebraic data type with two data constructors for leaves and branches respectively:

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
```

The type parameter `a` indicates that trees can be instantiated using any other type as payload, so the value `Leaf True` has type `Tree Bool`, whereas the value `Branch (Leaf 1) 2 (Leaf 3)` has type `Tree Int`. Then, if we assume existence of a predicate (function) `balanced` of type `Tree a -> Bool` that asserts that a tree satisfies a certain notion of balancedness, we can write testing properties to assert that the operations defined over the `Tree` data type preserve this invariant. For instance, the following testing property can be used to find bugs in the implementation of a given `insert` function:

```
prop_insert_inv :: a -> Tree a -> Property
prop_insert_inv x t = balanced t ==> balanced (insert x t)
```

This property asserts that, given an element `x` and a balanced tree `t` as input, inserting `x` into `t` will produce a balanced tree as output. (The implementation details of `balanced` and `insert` are not relevant for the point being made here.) Moreover, the arrow operator (`==>`) indicates that the predicate `balanced t` is a precondition of this property, and so attempting to test it using an unbalanced tree as an input will result in the test case getting *discarded*.

The last missing piece is a random generator of trees. We can write a naïve one for trees of integers (simplified to make it more accessible[2]) as follows:

```
genTree(size) =
  if size == 0 then do { x <- genInt; return (Leaf x) }
  else oneof [ do { x <- genInt; return (Leaf x) }
             , do { l <- genTree(size-1);
                    x <- genInt;
                    r <- genTree(size-1);
                    return (Branch l x r) } ]
```

This definition follows a common type-directed generation approach, and it is a good example of what to expect from a random generator synthesized automatically from the codebase. In esence, this generator randomly picks one data constructor from its corresponding data type definition with uniform probability (i.e, either a `Leaf` or a `Branch`), and then calls itself for every recursive subexpression needed to produce a well-typed value, carefully reducing

[2]In practice, this code will most likely be implemented using Haskell's type-class mechanism, hiding the complexity behind the size limit and the payload generator.

the input size limit `size` by a unit on each step, ensuring termination by restricting itself to generating leaves when it reaches zero (case `size == 0`). Integers are generated simply by delegating the task to an external random generator (`genInt`) defined elsewhere.

Readers familiar with RBPT will recognize that using `genTree` to test `prop_insert_inv` with *QuickCheck* directly does not work well. In practice, *QuickCheck* gives up if most of the generated test cases get discarded due to failed preconditions. Sadly, most of the inputs produced by our naïve generator suffer from this problem, and the interesting part of the property (`balanced (insert x t)`) is tested very sporadically as a result.

The next subsection shows how GCPT helps overcoming this limitation without having to write specialized generators by hand.

## 2.1 Coverage-Guided Property-Based Testing

To alleviate the problem of testing properties with non-trivial preconditions while using automatically derived random generators, CGPT enhances the testing process with two key characteristics: (1) *target code instrumentation*, to capture execution information from each test case; and (2) *high-level, structure-preserving mutations*, to produce syntactically valid test cases by altering existing ones at the data constructor level.

Using code instrumentation in tandem with mutations is a well-known technique in the fuzzing community. Generic fuzzing tools like *AFL* [30], *HonggFuzz* [39], or *libFuzzer* [2019] as well as language-specific ones like Crowbar [12] or *Kelinci* [24] use execution traces to recognize interesting test cases, e.g, those that exercise previously undiscovered parts of the target code. Later, such tools use generic mutators to combine and produce new test cases from previously executed interesting ones. In contrast with traditional fuzzing approaches, CGPT can distinguish semantically valid test cases from invalid ones, i.e., those passing the testing property preconditions as opposed to those that are discarded early. This information is used to favour mutating valid test cases over discarded ones.

The testing loop in CGPT uses two queues to store valid and discarded previously executed test cases. Enqueued test cases are stored along with a given mutation budget, that controls how many times a given test case can be mutated before being finally thrown away. This budget is calculated using a heuristic derived from AFL's power schedule, i.e., more budget to test cases that lead to shorter executions, or that discover more parts of the code. On each iteration, the algorithm selects the next test case by mutating the first value on the queue of valid test cases. If that queue is empty, it then selects the first one from the queue of discarded test cases. If both queues are empty, CGPT generates a new random value from scratch. The algorithm then run this test case and evaluates whether it was interesting (i.e., it exercises a new path) based on its trace information. If the test case does in fact discover a new path, it is enqueued at the end of its corresponding queue, depending on whether it passed the property precondition or was discarded. This process alternates between generation and mutation until a bug is found or it reaches the test limit.

*Limitations of CGPT* Lampropoulos et al. demonstrated empirically that CGPT lies comfortably in the middle ground between using pure random testing with naïve automatically derived random generators and complex manually-written ones. However, the authors acknowledge that certain parts of its implementation have room for improvement, especially when it comes to the mutator's design. Moreover, we replicated the IFC stack machine case study bundled with *FuzzChick* and observed a surprising limitation: *after repeating each experiment 30 times, FuzzChick could only find 5 out of the 20 injected bugs with a 100% efficacy, the hardest one being found only around 13% of the time after an hour of testing*. Despite this, the authors show that CGPT outperforms normal random testing coupled with naïve random generators by comparing the mean-time-to-failure (MTTF) against normal random property-based testing. In turn, we believe that a more meticulous evaluation ought to take failure to find a counterexample as an important metric when comparing PBT tools, not just mean time to failure — when one is found.

All these observations led us to consider three main aspects of CGPT that can be improved upon:

- *Mutators distribution:* for the sake of simplicity, the mutators proposed by Lampropoulos et al. are derived to follow a top-down approach. This means that mutations can happen at the top-level or be recursively applied to an immediate subexpression of the input test case with approximately the same probability. This makes deep recursive mutations very unlikely, as their probability decreases multiplicatively with each recursive call. Hence, these simple mutators can only be effectively used to transform shallow data structures, excluding applications that might require producing deeper mutations. Ideally, *mutations should be able to happen on every subexpression of the input test case on a reasonable basis.*
- *Enqueuing mutation candidates:* the CGPT testing loop uses two single-ended queues for keeping valid and discarded interesting test cases. Whenever a new test case is found interesting, it is placed *at the end of its corresponding queue*. If this test case happens to have discovered a whole new portion of the target code, it will not be further mutated until the rest of the queue ahead of it gets processed. This can limit the effectiveness of the testing loop if the queues grow more often than they shrink, as interesting test cases can get "buried" at the end of a long queue and, in an extreme case, not get processed at all within the testing budget. Ideally, *we need a mechanism that prioritizes mutating test cases that discover new portions of the code right away.*
- *Power schedule:* it is not clear how the heuristic used to assign a given budget to each mutable test case in CGPT works in the context of high-level structure-preserving mutations. If it assigns too much budget to certain not-so-interesting test cases, some bugs might not be discovered on a timely basis. Conversely, assigning too little budget to interesting test cases might mean that some bugs cannot be discovered at all — randomly generating the same test case again later does not help, as it becomes uninteresting based on historic trace information.

To keep the comparison fair, Lampropoulos et al. replicated the same power schedule configuration used in AFL, which is tailored to work in tandem with generic bit-level mutators. It is unclear what the best heuristic configuration is when using a high-level mutation approach — something challenging to characterize in general given the expressivity of the data types used to drive the mutators.

The next section introduces Mutagen, our revised CGPT that aims to tackle the main limitations of CGPT to make it practical in real-world scenarios.

## 3 MUTAGEN: No Mutant Left Behind

In this section we describe the main ideas behind MUTAGEN, our CGPT tool written in Haskell. Notably, these ideas extend beyond Haskell and functional programming languages in general.

In contrast to CGPT and many other popular fuzzing tools, MUTAGEN works by mutating test cases on an exhaustive and precise manner, where (1) each subexpression of a mutable test case is associated with a set of structure-preserving mutations, and (2) each one of these mutations is scheduled *exactly once*. Our realization is that, by using an exhaustive mutation approach, we avoid needing a heuristic to assign a mutation budget to each mutated test case. Moreover, producing mutants exhaustively ensures that no interesting mutation is omitted due to randomness, as well a no mutation is evaluated more than once. This is inspired by exhaustive bounded testing tools like *SmallCheck* [38] or *Korat* [7], that produce test cases exhaustively — please refer to §7 for a detailed discussion.

The rest of this section describes our exhaustive mutation mechanism, as well as the adapted testing loop used in MUTAGEN.

### 3.1 Exhaustive Mutations

Mutations in MUTAGEN are defined as the set of mutants that can be obtained by transforming the input test case *at the top-level* (the root). We define mutations for a value of type `a` (written `Mutations a`) as *a function that takes a value of such a type and returns a list of mutants* (written `Mutant a`). In Haskell, we introduce the following type synonym:

```haskell
type Mutations a = a -> [Mutant a]
```

Where `Mutants` come in two flavours, pure and random:

```haskell
data Mutant a = PURE a | RAND (Gen a)
```

Pure mutants are used most of the time, and encode simple deterministic transformations over the outermost data constructor of the input — recursive mutations will be introduced soon. These transformations can: (1) return an immediate subexpression of the same type as the input; (2) swap the outermost data constructor with any other constructor of the same type, reusing existing subexpressions; and (3) rearrange and replace fields using existing ones of the same type. To illustrate this, Fig. 1 outlines a mutator for the `Tree` data type. Notably, this definition simply enumerates mutants that transform the outermost data constructor, hence no recursion is needed here. Moreover, notice how a default value (`def`) used to fill the subtrees when transforming a leaf into a branch. This value corresponds to the simplest expression we can construct for the mutant to be type-correct. In our example, the default value of a `Tree` is a leaf containing the smallest value of the tree payload type, e.g., in the case of trees with integers `def` is defined as `Leaf 0`.[3] Using a small default value, as oppossed to a randomly generated one (as done by the original CGPT), is also inspired by exhaustive bounded testing tools, and avoids introducing unnecessary randomness when "growing" mutated expressions.

Random mutants, on the other hand, serve as a way to *selectively avoid exhaustiveness* when mutating values of large enumeration types — or any other type the user might want to use random mutations with. Instead of mutating every numerical or character subexpression exhaustively, we sample a small number of values

[3]We use the type class system to abstract this complexity away in our implementation.

```haskell
mutate (Leaf x) =
  [ PURE (Branch def x def) ]
mutate (Branch l x r) =
  [ PURE l, PURE r
  , PURE (Leaf x)
  , PURE (Branch l x l), PURE (Branch r x r), PURE (Branch r x l) ]
```

**Figure 1: MUTAGEN mutator for the `Tree` data type.**

from a given random generator. This amount is a tunable parameter of MUTAGEN. Then, for instance, a mutator for integers becomes:

```haskell
mutate n = [ RAND genInt ]
```

*Mapping top-level mutations everywhere* So far we have defined mutations that transform only the root of the input. Now it is time to apply these mutations to every subexpression as well. To do so, we use two utility functions that can be automatically synthesized.

In first place, a function `Positions` traverses the input and builds a Rose tree of *mutable positions*, where positions are lists of indices encoding the path from the root to every mutable subexpression. For instance, the mutable positions of the value `Branch (Leaf 1) 2 (Leaf 3)` are as follows:

```
            ⎛    Branch    ⎞             []
            ⎜     / | \    ⎟            / | \
Positions   ⎜ Leaf  2  Leaf⎟   =    [0] [1] [2]
            ⎜   |       |  ⎟         |        |
            ⎝   1       3  ⎠       [0,0]    [2,0]
```

Later, we define a function `MutateInside` that takes a desired position within an input test case and mutates its corresponding subexpression, returning a list of mutants. This function simply traverses the desired position, calling itself recursively until it reaches the desired subexpression, where a mutation encoded by `mutate` can be applied. The rest of the `MutateInside`'s functionality simply unwraps and rewraps the intermediate subexpressions and is not particularly relevant for the point being made.

### 3.2 Testing loop

Having the exhaustive mutation mechanism in place, we can now introduce the base testing loop used by MUTAGEN. This is outlined in Algorithm 1. Like in CGPT, we use two queues, QValid and QDiscarded to keep valid and discarded mutable test cases, respectively.

In MUTAGEN, we precompute all the mutations of a given mutable test case before enqueueing them. These mutations are put together

**Algorithm 1:** MUTAGEN Testing Loop

```
Function Loop(P, N, R, gen):
    i ← 0
    TLog, QValid, QDiscarded ← ∅
    while i < N do
        x ← Pick(QValid, QDiscarded, gen)
        (result, trace) ← WithTrace(P(x))
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            if not Discarded(result) then
                batch ← CreateMutationBatch(x, R)
                Enqueue(QValid, batch)
            else if not Discarded(Parent(x)) then
                batch ← CreateMutationBatch(x, R)
                Enqueue(QDiscarded, batch)
        i ← i+1
    return Ok
```

**Algorithm 2:** Mutation Batch Initialization

```
Function CreateMutationBatch(x, R):
    batch ← ∅
    for pos in Flatten(Positions(x)) do
        for mutant in MutateInside(pos, x) do
            switch mutant do
                case PURE $\hat{x}$ do
                    Enqueue($\hat{x}$, batch)
                case RAND gen do
                    repeat R times
                        $\hat{x}$ ← Sample(gen)
                        Enqueue($\hat{x}$, batch)
    return batch
```

into lists that we call *mutation batches* — one for each mutated test case. To initialize a mutation batch (outlined in Algorithm 2), we first flatten all the mutable positions of the input test case in level order (recall that positions are stored as a Rose tree). Then, we iterate over all of them and retrieve all the mutants defined for each subexpression. For each one of these, there are two possible cases: (1) if it is a pure mutant carrying a concrete mutated value, we enqueue it into the mutation batch directly; otherwise (2) it is a random mutant that carries a random generator with it (e.g., corresponding to a numeric subexpression), in which case we sample and enqueue $R$ random values using this generator, where $R$ is a parameter set by the user. At the end, we simply return the accumulated batch.

Finally, the seed selection algorithm (Algorithm 3) picks the next test case using the same criteria as CGPT, prioritizing valid test cases over discarded ones, falling back to random generation when both queues are empty. Since mutations are precomputed, this function simply picks the next test case from the current batch, until it becomes empty and can switch to the next precomputed one in line.

Having selected the next test case, the testing loop proceeds to execute it, capturing both the result (valid, discarded, or failed) and its execution trace over the system under test. If the test case fails, it is reported as a bug. If not, the algorithm evaluates whether it was interesting (i.e., it exercises a new path) based on its trace information and the one from previously executed test cases (represented by TLog). If the test case discovers a new path, it is enqueued on its corresponding queue. This process alternates between generation and mutation until it finds a bug or reaches the test limit N.

**Algorithm 3:** Mutagen Seed Selection

```
Function Pick(QValid, QDiscarded, gen):
    if not Empty(QValid) then
        batch ← Deque(QValid)
        if Empty(batch) then Pick(QValid, QDiscarded, gen)
        else
            PushFront(QValid, Rest(batch))
            return First(batch)
    if not Empty(QDiscarded) then
        batch ← Deque(QDiscarded)
        if Empty(batch) then Pick(QValid, QDiscarded, gen)
        else
            PushFront(QDiscarded, Rest(batch))
            return First(batch)
    else return Sample(gen)
```

Another difference with CGPT's testing loop is the criterion for enqueuing discarded tests. We found that, especially for large data types, the queue of discarded candidates tends to grow disproportionately large during testing, making them hardly usable and consuming large amounts of memory. To improve this, we resort to mutate discarded tests cases only when we have some evidence that they are "almost valid." For this, each mutated test case remembers whether its parent (the original test case they derive from after being mutated) was valid. Then, we enqueue discarded test cases only if they meet this condition. As a result, we fill the discarded queue with lesser but much more interesting mutable test cases.

The next section describes two heuristics we added to Mutagen's testing loop based on the limitations we found in CGPT.

## 4 Mutagen Heuristics

In this section we introduce two heuristics implemented on top of the base testing loop of our tool.

### 4.1 Priority FIFO Scheduling

This heuristic tackles the issue of enqueuing new interesting mutation candidates at the end of (possibly) long queues of not-so-interesting previously executed ones.

Execution traces in Mutagen represent the specific *path* in the code taken by the program, as opposed to just the (unordered) set of edges traversed in the control-flow graph (CFG) used by default by most coverage-guided fuzzers. Using this criterion lets us gather precise information from each new execution. In particular, we are interested in the *depth* where each new execution branches from already seen ones. Our assumption is that test case executions that branch at shallower depths from the ones already executed are more likely to discover new portions of the code under test, and hence we want to prioritize them.

In this light, every time we insert a new execution path into the internal trace log (TLog), we calculate the number of *new nodes* that were executed, as well as the *branching depth* where they got inserted. The former is used to distinguish interesting test cases (whether or not new nodes were inserted), whereas the latter is used by this heuristic to schedule mutation candidates. Fig. 2 illustrates this idea, inserting two execution traces (one after another) into a trace log that initially contains a single execution path. The second insertion (with trace $1 \to 2 \to 6 \to 7$) branches at a shallower depth than the first one (2 vs. 3), and so its corresponding test case is given a higher priority.

Using this mechanism, we can modify Mutagen's base testing loop by replacing each mutation queue with a priority queue indexed by the branching depth of each new execution. These changes are illustrated in Algorithm 4. Statements in red indicate important

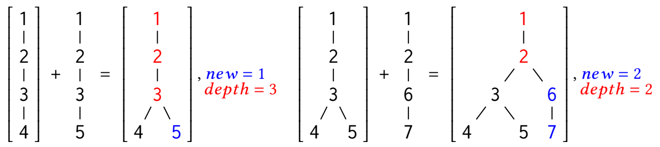

**Figure 2: Inserting two new execution traces into the internal trace log (represented using brackets).**

**Algorithm 4:** Priority FIFO Heuristic

```
Function Loop(P, N, R, gen):
    ...
    x ← Pick(QValid, QDiscarded, gen)
    (result, trace) ← WithTrace(P(x))
    ...
    if Interesting(TLog, trace) then
        if not Discarded(result) then
            batch ← Mutate(x, R)
            prio ← BranchDepth(TLog, trace)
            PushFront(QValid, prio, batch)
        ...
Function Pick(QValid, QDiscarded, gen):
    if not Empty(QValid) then
        (batch, prio) ← DequeMin(QValid)
        if Empty(batch) then  Pick(QValid, QDiscarded, gen)
        else
            PushFront(QValid, prio, Rest(batch))
            return First(batch)
    if not Empty(QDiscarded) then
        /* Analogous to the case above */
    ...
```

changes to the base algorithm, whereas ellipses denote parts of the code that are not relevant for the point being made.

To pick the next test case, we retrieve the one with the highest priority (the smallest branching depth). Then, when we find a new interesting test case, it gets enqueued at the *beginning* of the queue at its corresponding priority. This allows the testing loop to jump immediately onto mutating new interesting test cases as soon as they are found (even at the same priority), and to jump back to previous test cases as soon as their mutants become less novel.

### 4.2 Detecting Trace Space Saturation And Tuning Random Mutations Parameter

As introduced in §3, our tool is parameterized by the number of random mutations to be generated over each mutable subexpression defined using a random mutant, e.g., numeric values. But, how many random mutations should we use? A single one? A few tenths? A few hundred? Answering this question precisely is not an easy task, and this second heuristic aims to tackle this issue.

We found that, the smaller the number of random mutations we set, the easier it is for the trace log that records executions to start getting saturated, i.e., when interesting test cases stop getting discovered or are discovered very sporadically. We realized that we can use this information to automatically optimize the number of random mutations used by our tool. This idea is described in Algorithm 5. The process is simple: (1) we start the testing loop with the number of random mutations set to one, (2) each time we find that a test is not interesting (i.e. boring), we increment a counter, (3) if we have not produced any interesting test case after a certain threshold (1000 tests seems to be a reasonable value in practice), we increment the number of random mutations and the threshold by twice the current amount. Additionally, each time this happens we also reset the trace log, so interesting test cases found on a previous

**Algorithm 5:** Trace Saturation Heuristic

```
Function Loop(P, N, gen):
    boring ← 0; reset ← 1000; R ← 1
    ...
    while i < N do
        if boring > reset then
            TLog ← ∅
            reset ← reset * 2
            R ← R * 2
        ...
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            boring ← 0
            ...
        else  boring ← boring + 1
        ...
```

iteration can be found and enqueued for mutation again — this time with a higher effort dedicated to producing random mutations.

Then, if the execution of the system under test depends heavily on the values stored at randomly mutable subexpressions, starting with a single random mutation will quickly saturate the trace space, and this heuristic will continuously increase the random mutations parameter until that stops happening.

## 5 Case studies

We evaluated the performance of Mutagen using two case studies. The first one is a simple abstract stack machine that enforces the hyperproperty *noninterference* [15] using runtime checks. Its implementation was proven correct by Azevedo de Amorim et al. [2014] in Coq, and subsequently degraded by systematically introducing 20 bugs on its enforcing mechanism. Lampropoulos et al. used this case study to compare *FuzzChick* against RPBT using naïve- and hand-tuned random generators. Here, we replicate their results and compare them against our tool. Worth mentioning, we reimplemented this case study from scratch in Haskell in order to run Mutagen on its test suite.

The second case study aims to evaluate Mutagen in a realistic scenario, and targets *haskell-wasm* [37], an existing WebAssembly engine of industrial strength. Sadly, comparing Mutagen with *FuzzChick* on this case study has been out of the scope of this work, since porting *haskell-wasm* into Coq, and later adapting our test suite to work with *FuzzChick* requires a subtantial effort. Instead, we compare Mutagen against a traditional RPBT approach, evaluating its effectiveness versus the relative overhead of the code instrumentation used by our tool.

### 5.1 IFC Stack Machine

This abstract stack machine consists of four main elements: a program counter, a stack, and data- and instruction memories. Moreover, every runtime value is labeled with a security level, i.e., L (for "low" or public) or H (for "high" or secret). Labels form a trivial 2-lattice where information can either stay at the same level, or public information "can flow" to secret one but not the opposite (represented using the familiar $\sqsubseteq$ binary operator). Security labels are propagated throughout the execution of the program whenever the machine executes an instruction. These instructions are defined as:

```
data Instr = Nop | Push Int | Call Int | Ret | Add | Load | Store | Halt
```

Control flow is achieved using the `Call` and `Ret` instructions, which let the program jump back and forth within the instruction memory using specially labeled values in the stack representing memory addresses. The argument in the `Push` instruction represents a value to be inserted in the stack, whereas the argument of the `Call` instruction encodes the number of elements of the stack to be treated as arguments. Then, programs are simply modeled as sequences of instructions. Finally, machine states are modeled using a 4-tuple *(pc, stk, m, i)* representing a particular configuration of the program counter, the stack, and the data- and instruction memories, respectively. To preserve space, we encourage the reader to refer to the work of Hritcu et al. [2013, 2016] and Lampropoulos et al. for more details about the implementation and semantics of this case study.

*Single-Step Noninterference (SSNI)* This abstract machine is designed to enforce noninterference, which is based on the notion of *indistinguishability*. Intuitively, two machine states are indistinguishable if they only differ on secret data. Using this notion, the variant of noninterference we are interested in is called *single-step noninterference* [21] (SSNI). Given two indistinguishable machine states, SSNI asserts that running a single instruction on both machines brings them to resulting states that are also indistinguishable.

The tricky part about testing this property is to satisfy its sparse precondition: we need to generate two valid indistinguishable machine states in order to even proceed to execute the next instruction. Lampropoulos et al. demonstrated that generating two independent machine states using *QuickCheck* has virtually no chance of producing valid indistinguishable ones. However, using the mutation mechanism, we can obtain a pair of valid indistinguishable machine states by generating a single valid machine state (something still hard but much easier than before), and then producing a similar mutated copy. This way, we have a higher chance of producing two almost identical states that pass the sparse precondition.

*IFC Rules* The IFC rules enforced by this machine are encoded in a table indexed by its different instructions. This table contains: the dynamic check that the machine needs to perform to enforce the IFC policy, along with the corresponding security label of the program counter and the instruction result after the instruction is executed. For instance, to execute the `Store` instruction (which stores a value in a memory pointer) the machine checks that both the labels of the program counter and the pointer together can flow to the label of the destination memory cell. If this condition is not met, the machine then halts as indication of a violation in the IFC policy. After this check, the machine overwrites the value at the destination cell and updates its label with the maximum sensibility of the involved labels. In the rule table, this looks as follows:

| **Instruction** | **Precondition Check** | **Final PC Label** | **Final Result Label** |
|---|---|---|---|
| `Store` | $l_{pc} \vee l_p \sqsubseteq l_v$ | $l_{pc}$ | $l_{v'} \vee l_{pc} \vee l_p$ |

Where $l_{pc}, l_p, l_v$, and $l_{v'}$ represent the labels of: the program counter, the memory pointer, and the old and new values stored in that memory cell. The symbol $\vee$ simply denotes the join of two labels, i.e., the *maximum* of their sensibilities. Later, bugs are systematically introduced in the IFC enforcing mechanism by removing or weakening the checks stored in this rule table.

## 5.2 WebAssembly Engine

WebAssembly [18] is an assembly-like language designed for executing low-level code in the web. WebAssembly programs are first validated and later executed in a sandboxed environment. The language is relatively simple, in essence (1) it contains only four numerical types, representing both integers and IEEE754 floating-point numbers of either 32 or 64 bits; (2) values of these types are manipulated by functions written using sequences of stack instructions; (3) functions are organized in modules and must be explicitly imported and exported; (4) memory blocks can be imported, exported and grown dynamically; among others. WebAssembly semantics are fully specified, and programs must be consistently interpreted across engines — despite some subtle details that we will address soon. For this, the WebAssembly standard provides a reference implementation with all the functionality expected from a compliant engine.

Our tool is an attractive match for testing WebAssembly engines: most of the programs that can be represented using WebAssembly's AST are invalid, and automatically derived random generators cannot satisfy the invariants required to produce interesting test cases.

In this work, we are interested in using Mutagen to test the two most complex subsystems of *haskell-wasm*: the *validator* and the *interpreter* — both being previously tested using a unit test suite.

We avoided spending countless hours writing an extensive property-based specification to mimic the WebAssembly specification. Instead, we used the reference implementation to find discrepancies (that could potentially lead to bugs) via differential testing [31]. In this manner, our testing properties assert that any result produced by *haskell-wasm* matches that of the reference implementation.

Notably, this engine had several latent bugs that were not caught by the existing unit tests and that we discovered by Mutagen while developing our test suite. Moreover, Mutagen exposed two other discrepancies between *haskell-wasm* and the reference implementation. These discrepancies trigger parts of the specification that are either not yet supported by the reference implementation (multi-value blocks), or that produce a well-known non-deterministic undefined behavior (NaN reinterpretation) [36]. These findings are briefly outlined in Table 1. All of them were reported and later confirmed by the authors of *haskell-wasm*. Having sorted these issues

| **Id** | **Subsystem** | **Category** | **Description** |
|---|---|---|---|
| 1 | Validator | Bug | Invalid memory alignment validation |
| 2 | Validator | Discrepancy | Validator accepts blocks returning multiple values |
| 3 | Interpreter | Bug | Instance function invoker silently ignores arity mismatch |
| 4 | Interpreter | Bug | Allowed out-of-bounds memory access |
| 5 | Interpreter | Discrepancy | NaN reinterpretation does not follow reference |

**Table 1: Issues found by Mutagen in *haskell-wasm*.**

| **Id** | **Subsystem** | **Description** |
|---|---|---|
| 1 | Validator | Wrong if-then-else type validation on else branch |
| 2 | Validator | Wrong stack type validation |
| 3 | Validator | Removed function type mismatch assertion |
| 4 | Validator | Removed max memory instances assertion |
| 5 | Validator | Removed function index out-of-range assertion |
| 6 | Validator | Wrong type validation on `i64.eqz` instruction |
| 7 | Validator | Wrong type validation on `i32` binary operations |
| 8 | Validator | Removed memory index out-of-range assertion |
| 9 | Validator | Wrong type validation on `i64` constants |
| 10 | Validator | Removed alignment validation on `i32.load` instruction |
| 11 | Interpreter | Wrong interpretation of `i32.sub` instruction |
| 12 | Interpreter | Wrong interpretation of `i32.lt_u` instruction |
| 13 | Interpreter | Wrong interpretation of `i32.shr_u` instruction |
| 14 | Interpreter | Wrong local variable initialization |
| 15 | Interpreter | Wrong memory address casting on `i32.load8_s` instruction |

**Table 2: Bugs injected into *haskell-wasm*.**

out, we injected 10 new bugs in the validator as well as 5 new bugs in the interpreter of this engine (see Table 2). These bugs were mechanically injected by either (1) removing an existing integrity check (to weaken the WebAssembly type-system/validator); or by (2) simulating a copy-and-paste induced bug,[4] replacing the implementation of an instruction with a compatible one (e.g., `i32.add` by `i32.sub`).

*Testing the WebAssembly Validator* To keep things simple, we assert that, whenever a randomly generated (or mutated) WebAssembly module is valid according to *haskell-wasm*, then the reference implementation agrees upon it. In Haskell, we write the property:

```
prop_validator m = isValidHaskellWasm m ==> isValidRefImpl m
```

The precondition (`isValidHaskellWasm m`) runs the input WebAssembly module `m` against *haskell-wasm*'s validator, whereas the postcondition (`isValidRefImpl m`) serializes `m`, runs it against the reference implementation and checks that no errors are produced.

We note that, although we only focus on finding false negatives, a comprehensive test suite should also test for false positives, i.e., when a module is valid and *haskell-wasm* rejects it.

*Testing the WebAssembly Interpreter* Testing the WebAssembly interpreter is substantially more complicated than testing the validator, since it requires actually running and comparing the output of test case programs. To achieve this, the generated test cases need to comply a with a common interface that can be invoked both by *haskell-wasm* and the reference implementation. To simplify things, we write a function `mk_module` to build a stub module which initializes one memory block and exports a single WebAssembly function. Such function is parameterized by the definition of the single function, along with its name and type signature. Then, we can use `mk_module` to define a testing property parameterized by a function type, along with its definition and invocation arguments:

```
prop_interpreter ty fun args =
  do let m = mk_module ty "foo" fun
     resHs   <- invokeHaskellWasm m "foo" args
     resSpec <- invokeRefImpl     m "foo" args
     return (equivalent resHs resSpec)
```

This property instantiates a module stub `m` using the input function (`fun`) and its type signature (`ty`). Then, it invokes the function `foo` of the module `m` both on *haskell-wasm* and the reference interpreter with the provided arguments (`args`). Finally, the property asserts whether their results are equivalent.[5] Interestingly, equivalence in does not imply equality. Non-deterministic operations in WebAssembly like NaN reinterpretations can produce different equivalent results (as exposed by the discrepancy #5 in Table 1), and our equivalence relation needs to take that into account.

Using this testing property directly might not sound like a great idea, as randomly generated lists of input arguments will be very unlikely to match the type signature of randomly generated functions. However, it lets us test what happens when programs are not properly invoked, and it quickly discovered the previously unknown bug #3 in *haskell-wasm* mentioned above. Having fixed this issue, we define a more useful specialized version of `prop_interpreter` that fixes the type of the generated function to take two arguments (of type `I32` and `F32`) and return an `I32` as a result:

```
prop_interpreter_i32 fun i f = prop_interpreter
  (FuncType { params = [I32, F32], result = [I32] }) fun [VI32 i, VF32 f]
```

[4]This kind of bugs was inspired by the real bug #1 we found prior this step.
[5]In our implementation, we additionally set a short timeout to discard potentially diverging programs with infinite loops.

This property lets us generate functions with a fixed type and invoke them with the exact number and type of arguments required. We use this property when finding all the injected bugs into the *haskell-wasm* interpreter in the next section.

## 6 Evaluation

We performed all experiments in a dedicated workstation with 32GB of RAM and an Intel Core i7-8700 CPU running at 3.20GHz. We repeated each experiment 30 times except for the ones testing the WebAssembly interpreter, which were run 10 times due to time constraints. From there, we followed the approach taken by Lampropoulos et al. and collected the mean-time-to-failure (MTTF) of each bug, i.e., how quickly a bug can be found in wall clock time. We used a one-hour timeout to stop the execution of both tools if they have not yet found a counterexample. In addition, we collected the failure rate (FR) observed for each bug, i.e. the proportion of times each tool finds each bug within the one-hour testing budget.

In both case studies, we show how the two Mutagen heuristics described in §4 affect the testing performance by individually disabling them. We call these variants *no FIFO* and *no reset*, respectively. In the case of *no reset*, the amount of random mutations is no longer controlled by this heuristic, so we arbitrarily fixed it to 25 random mutations throughout the execution of testing loop.

### 6.1 IFC Stack Machine

The results of this case study are shown in Fig. 3. The injected bugs are ordered by the failure rate achieved by *FuzzChick* in decreasing order. Moreover, notice the logarithmic scale used on the MTTF.

As introduced earlier, *FuzzChick* can only find 5 out of the 20 bugs injected with a 100% success rate within the one hour testing budget. In turn, our tool manages to find every bug on all runs, and taking less than a minute in the worst absolute case. While these results are somewhat pessimistic towards *FuzzChick* if compared to the ones presented originally by Lampropoulos et al., we remark that our experiments encompass 30 independent runs instead of the 5 used in the original study. Moreover, since the MTTF simply aggregates all runs, regardless of if they found a bug or timed out, this metric is sensitive to the timeout used on each experiment, and will inflate the results as soon as the failure rate goes below 1. For this reason, comparing the failure rate between tools gives a better estimation of their reliability, where Mutagen shows a clear improvement.

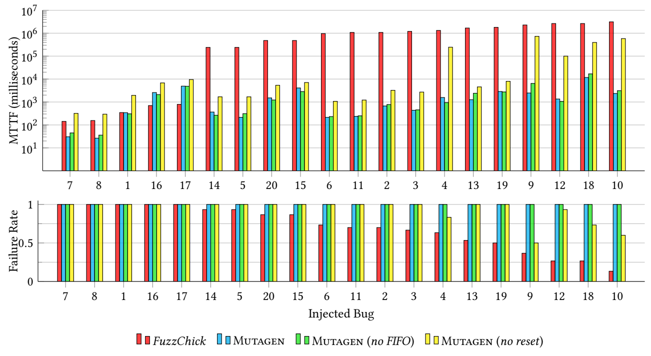

**Figure 3: Comparison between *FuzzChick* and Mutagen across 20 different bugs for the IFC stack machine.**

After carefully analyzing the results obtained using *FuzzChick*, we observed a peculiarity. This tool either finds bugs quickly (after a few thousands tests) or does not find them at all within the one-hour budget. With this in mind, it is fair to think that *FuzzChick* might be a better choice if we set a shorter timeout and run it several times in a row. Thus, to be an improvement over *FuzzChick*, our tool must find bugs not only more reliably, but also fast!

Table 3 shows a comparison between the mean number of tests required by both tools to find the first failure *when we only consider successful runs.* Additionally, as advised by Arcuri and Briand [2], we computed the non-parametric Vargha-Delaney $A_{12}$ measure [40]. Intuitively, this measure encodes the probability of MUTAGEN to yield better results than *FuzzChick*. The *Estimate* of this measure simply denotes the statistical difference between the compared distributions in a categorical manner. Estimates in red and blue indicate results favoring *FuzzChick* and MUTAGEN, respectively. As it can be observed, MUTAGEN is likely to find bugs faster than *FuzzChick* in 14 of the 20 bugs injected into the abstract machine, being substantially slower in only 2 cases.

| Bug | *FuzzChick* | MUTAGEN | $A_{12}$ measure | |
|---|---|---|---|---|
| | | | Value | Estimate |
| 1 | 13974.9 | 3632.7 | 0.89 | large |
| 2 | 23962.9 | 7122.3 | 0.90 | large |
| 3 | 19678.5 | 4633.9 | 0.89 | large |
| 4 | 20398.4 | 16831.2 | 0.72 | medium |
| 5 | 17348.1 | 2326.6 | 0.94 | large |
| 6 | 10727.1 | 2312.9 | 0.92 | large |
| 7 | 5070.7 | 332.9 | 0.91 | large |
| 8 | 5596.4 | 298.7 | 0.90 | large |
| 9 | 16402.4 | 26342.5 | 0.51 | negligible |
| 10 | 11553.0 | 25044.5 | 0.55 | negligible |
| 11 | 11304.3 | 2536.8 | 0.82 | large |
| 12 | 18507.3 | 14482.7 | 0.65 | small |
| 13 | 17961.5 | 13454.9 | 0.65 | small |
| 14 | 10621.9 | 3928.3 | 0.78 | large |
| 15 | 23866.3 | 43678.2 | 0.23 | large |
| 16 | 25321.7 | 27602.0 | 0.54 | negligible |
| 17 | 28515.6 | 52344.9 | 0.47 | negligible |
| 18 | 24218.6 | 123401.3 | 0.14 | large |
| 19 | 18638.9 | 30682.3 | 0.45 | negligible |
| 20 | 18883.1 | 15841.9 | 0.60 | small |
| | | Mean | 0.67 | medium |

**Table 3: Mean tests until first failure and effect size comparison between *FuzzChick* and MUTAGEN when considering only successful runs.**

In terms of the MUTAGEN heuristics, this case study is not strongly affected by using FIFO scheduling. The reason behind this is that the mutation queues remain empty most of the time (generation mode), and when a new interesting candidate gets inserted, all its mutants (and their descendants) are processed before the next one is enqueued. Hence, the small proportion of time this heuristic is active tends to be inconsequential.

Moreover, disabling the trace saturation heuristic (*no reset*) we observed two effects: fixing the number of random mutations to 25 adds a constant overhead when finding most of the (easier) bugs, suggesting that we are spending too time mutating numeric subexpressions inside the generated machine states. However, some of the hardest-to-find bugs cannot be reliably exposed using this fixed amount of random mutations (#4, #9, #12, #18, and #10), suggesting that one should increase this number even further to be able to discover all bugs within the time budget. In consequence, we argue that our heuristic is effective at automatically tuning this parameter.

## 6.2 WebAssembly Engine

The results of this case study are shown in Fig. 4, ordered by the MTTF achieved by MUTAGEN.

We first focus on the bugs injected in the validator (Fig. 4 left). There, we quickly conclude that *QuickCheck* is not well suited to find most of the bugs — it merely finds the easier bugs #4 and #5 in just 1 out of 30 runs. The reason behind this simple: using an automatically derived generator is virtually unable to produce valid

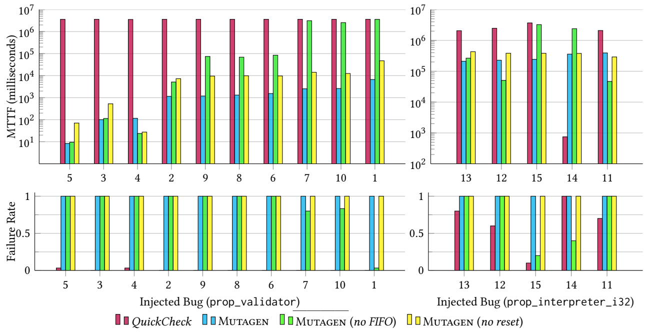

**Figure 4: Comparison between *QuickCheck* and MUTAGEN across 15 different bugs injected into *haskell-wasm*.**

| Property | *QuickCheck* | | MUTAGEN | |
|---|---|---|---|---|
| | total | passed | total | passed |
| *prop_validator* | 41374.92 | 0.00029 | 2243.26 | 519.83 |
| *prop_interpreter_i32* | 134000 | 23.16 | 1815.81 | 351.16 |

**Table 4: Mean number of tests per second executed by each tool on the WebAssembly case study.**

WebAssembly modules apart from the trivial empty one. Using the same random generator, however, MUTAGEN is able to consistently find every bug in less than 20 seconds in the absolute worst case.

In terms of the MUTAGEN heuristics, we observe two clear phenomena. On one hand, disabling the FIFO scheduling (case *no FIFO*), the hardest-to-find bugs (#7, #10, and #1) are no longer found on every run. Moreover, although bugs #2, #9, #8, and #6 are found with 100% success rate across runs, it takes several times longer for MUTAGEN to find them. On the other hand, disabling the trace saturation heuristic (case *no reset*) and using a fixed number of 25 random mutation does not affect the effectiveness of our tool, apart from adding again a constant time overhead.

If we now pay attention to the bugs injected into the interpreter (Fig. 4 right), we notice that finding bugs requires substantially more time (minutes instead of seconds), since both interpreters need to validate and run the inputs before producing a result to compare. We can also observe a significant improvement in the performance of *QuickCheck* in terms of failure rate. This is of no surprise: we deliberately reduced the search space by using a module stub when defining `prop_interpreter` in our test suite. Notably, *QuickCheck* finds counterexamples for the bug #14 almost instantly. The reason behind this is that this bug can be found using a very small counterexample, and *QuickCheck* prefers sampling small test cases at the beginning of the testing loop. Our tool follows this same approach when in generation mode. However, when a test case is found interesting, the scheduler does not take its size into account while computing its priority — future work should investigate this possibility. Nonetheless, MUTAGEN still outperforms *QuickCheck* on the remaining bugs both in terms of failure rate and mean time to failure. Moreover, by disabling the FIFO scheduling we observe that the performance of our tool becomes unstable, where bugs #12 and #11 take substantially less time to be found, whereas bugs #15 and #14 cannot be found on every run. Finding where the best trade-off lies in cases where the heuristics cause mixed results like this is another interesting direction to aim our future work.

Finally, this case study allows us to analyze the overhead introduced by the code instrumentation and internal processing used in

Mutagen versus the stateless black-box approach of *QuickCheck*. Table 4 compares the total number of executed and passed tests per second that we observed using each tool. Mutagen executes tests several times slower than *QuickCheck*— roughly 20x and 75x slower when testing `prop_validator` and `prop_interpreter_i32`, respectively. Despite this, our tool is still capable of running substantially more tests that pass the sparse preconditions than *QuickCheck* in the same amount of time, *and that ultimately leads us to find bugs.*

## 7 Related work

There exists a vast literature on fuzzing and property-based testing. In this section we focus on three main topics: automated random data generation using static information, coverage-guided fuzzing, and exhaustive bounded testing, all of which inspired this work.

*Automated Random Data Generation* Obtaining random generators automatically from static information (e.g., grammars or data type definitions) has been tackled from different angles. DRAGEN [33] is a meta-programming tool that synthesises random generators from data types definitions, using stochastic models to predict and optimize their distribution based on a target one set by the user. DRAGEN2 [32] extends this idea with support for generating richer random data by extracting library APIs and function input patterns from the codebase. *QuickFuzz* [16, 17] is a fuzzer that exploits these ideas to synthesize random generators from existing Haskell libraries. These generators are used in tandem with existing low-level fuzzers to find bugs in heavily used programs.

Automatically deriving random generators is substantially more complicated when the generated data must satisfy (often sparse) preconditions. Claessen et al. [2014] developed an algorithm for generating inputs constrained by boolean preconditions with almost-uniform distribution. Later, Lampropoulos et al. [2017a] extended this approach by adding a limited form of constraint solving controllable by the user in a domain-specific language called *Luck*. Recently, Lampropoulos et al. [2017b] proposed a mechanism to obtain constrained random generators by defining their target structure using inductively defined relations in Coq.

We consider all these generational approaches to be orthogonal to the ideas behind Mutagen. While our tool is tailored to improve the performance of poor automatically derived generators, specialized ones can be used directly by Mutagen, and combining them with structure-preserving mutations is part of our future work.

*Coverage-guided Fuzzing* *AFL* [30] is the reference tool when it comes to coverage-guided fuzzing. *AFLFast* [6] extends AFL using Markov chain models to tune the power scheduler towards testing low-frequency paths. In our tool, the scheduler does not account for path frequency. Instead, it favors a breadth-first traversal of the execution path space. Similar to Mutagen, *CollAFL* [14] is a variant of AFL that uses path- instead of edge-based coverage to distinguish executions more precisely by reducing path collisions.

If the source code history is available, *AFLGo* [5] is a fuzzer that can be targeted to exercise the specific parts of the code affected by recent commits in order to find potential new bugs. In PBT, a related idea called *targeted property-based testing* (TPBT) is to use fitness functions to guide testing efforts towards user defined goals [28, 29].

All these ideas can potentially be incorporated in our tool, and we keep them as a challenge for future work.

*Exhaustive Bounded Testing* A popular category of property-based testing tools does not rely on randomness. Instead, all possible inputs can be enumerated and tested from smaller to larger up to a certain size bound. *Feat* [13] formalizes the notion of *functional enumerations*. For any algebraic type, it synthesizes a bijection between a finite prefix of the natural numbers and a set of increasingly larger values of the input type. Later, this bijection can be traversed exhaustively or, more interestingly, randomly accessed. This allows the user to easily generate random values uniformly simply by sampling natural numbers. However, values are enumerated based only on their type definition, so this technique is not suitable for testing properties with sparse preconditions expressed elsewhere. *SmallCheck* [38] is a Haskell tool that follows this idea. It progressively executes the testing properties against all possible input values up to a certain size bound. *Korat* [7] is a Java tool that uses method specification predicates to automatically generate all non-isomorphic test cases up to a given small size.

These approaches rely on pruning mechanisms to avoid populating unevaluated subexpresions exhaustively before the computational cost of being exhaustive becomes too restrictive. *LazySmallCheck* is a variant of *SmallCheck* that uses lazy evaluation to automatically prune the search space by detecting unevaluated subexpressions. In the case of *Korat*, pruning is done by instrumenting method precondition predicates and analyzing which parts of the execution trace correspond to each evaluated subexpression.

In this work we use exhaustiveness as a way to reliably enforce that all possible mutants of an interesting test case are executed. In contrast to fully-exhaustive testing tools, Mutagen relies on randomness to find initial interesting mutable test cases. In terms of pruning, it is possible to instruct Mutagen to detect unused subexpressions to avoid producing mutations over their corresponding positions. This could improve the overall performance when testing non-strict properties that leave parts of their input unevaluated. In our case studies, however, we observed that the properties we tested are quite strict when executing test cases obtained by mutating valid existing ones, fully evaluating their input before executing the postcondition. Thus, we do not forsee considerable improvements by applying lazy-prunning in this scenario — gathering evidence about lazy pruning in Mutagen is as part of our future work.

## 8 Conclusions

We presented Mutagen, a coverage-guided, property-based testing framework. Our tool extends the CGPT approach with an exhaustive mutation mechanism that generates every possible mutant for each interesting test case, and schedules it to be tested exactly once.

Our experimental results indicate that Mutagen outperforms the simpler CGPT approach implemented in *FuzzChick* in terms of both failure rate and tests until first failure. Moreover, we show how our tool can be applied in a real-world testing scenario, where it quickly discovers several planted and some previously unknown bugs.

As indicated throughout this work, there are several directions for future work: combining Mutagen with specialized generators; enhancing the code instrumentation to be able target mutations towards specific parts of the code; and evaluating the effect of lazy pruning; among others.

## References

[1] 2019. LibFuzzer: A library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.

[2] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[3] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A Verified Information-Flow Architecture. *SIGPLAN Not.* 49, 1 (Jan. 2014), 165–178. https://doi.org/10.1145/2578855.2535839

[4] M. Bendkowski, K. Grygiel, and P. Tarau. 2017. Boltzmann Samplers for Closed Simply-Typed Lambda Terms. In *In Proc. of International Symposium on Practical Aspects of Declarative Languages*. ACM.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 123–133.

[8] Lukas Bulwahn. 2012. The new quickcheck for Isabelle. In *International Conference on Certified Programs and Proofs*. Springer, 92–108.

[9] K. Claessen, J. Duregård, and M. H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Proc. of the Functional and Logic Programming FLOPS*.

[10] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[11] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*.

[12] Stephen Dolan and Mindy Preston. 2017. Testing with crowbar. In *OCaml Workshop*.

[13] J. Duregård, P. Jansson, and M. Wang. 2012. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*.

[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[15] Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–11.

[16] G. Grieco, M. Ceresa, and P. Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. In *Proc. of the ACM SIGPLAN International Symposium on Haskell*.

[17] G. Grieco, M. Ceresa, A. Mista, and P. Buiras. 2017. QuickFuzz testing for fun and profit. *Journal of Systems and Software* 134 (2017).

[18] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[19] William Gallard Hatch, Pierce Darragh, and Eric Eide. 2020. Xsmith software repository. https://gitlab.flux.utah.edu/xsmith/xsmith.

[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.

[21] Catalin Hritcu, John Hughes, Benjamin C Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing noninterference, quickly. *ACM SIGPLAN Notices* 48, 9 (2013), 455–468.

[22] Cătălin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo De Amorim, Maxime Dénès, John Hughes, Benjamin C Pierce, and Dimitrios Vytiniotis. 2016. Testing noninterference, quickly. *Journal of Functional Programming* 26 (2016).

[23] John Hughes. 2003. Erlang/QuickCheck. In *Ninth International Erlang/OTP User Conference, Älvsjö, Sweden. November 2003*.

[24] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2511–2513.

[25] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia. 2017. Beginner's luck: a language for property-based generators. In *Proc. of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*.

[26] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[27] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. 2017. Generating Good Generators for Inductive Relations. *In Proc. ACM on Programming Languages* 2, POPL, Article 45 (2017).

[28] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 46–56.

[29] Andreas Löscher and Konstantinos Sagonas. 2018. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 70–80.

[30] M. Zalewski. 2010. American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/.

[31] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[32] Agustín Mista and Alejandro Russo. 2019. Generating Random Structurally Rich Algebraic Data Type Values. In *Proceedings of the 14th International Workshop on Automation of Software Test*.

[33] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*.

[34] M. Pałka, K. Claessen, A. Russo, and J. Hughes. 2011. Testing and Optimising Compiler by Generating Random Lambda Terms. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*.

[35] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*. 39–50.

[36] Árpád Perényi and Jan Midtgaard. 2020. Stack-Driven Program Generation of WebAssembly. In *Asian Symposium on Programming Languages and Systems*. Springer, 209–230.

[37] Ilya Rezvov. 2018. wasm: WebAssembly Language Toolkit and Interpreter. https://hackage.haskell.org/package/wasm.

[38] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, Vol. 44. ACM, 37–48.

[39] Robert Swiecki. 2010. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. https://github.com/google/honggfuzz.

[40] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[41] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.