Mutagen: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Well-Typed Mutations

ANONYMOUS AUTHOR(S)

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

Additional Key Words and Phrases: random testing, mutations, heuristics

ACM Reference Format:
Anonymous Author(s). 2021. Mutagen: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Well-Typed Mutations. In *Proceedings of OOPSLA '21: ACM SIGPLAN conference on Systems, Programming, Languages, and Applications (OOPSLA '21).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

The contributions of this work are:

- Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio.
- Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.
- Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc

OOPSLA '21, October 17–22 2021, Chicago, IL

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio.

- Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

2 BACKGROUND

In this section we briefly introduce the concept of Property-based Testing along with *QuickCheck*, the most popular tool of this sort and often considered as the baseline when comparing PBT algoritms. Moreover, we describe the ideas and limitations behind *FuzzChick*, which enhances the PBT approach with execution information and serves as the foundation for MUTAGEN. *FuzzChick* is implemented as an extension of *QuickChick*, Coq's own reimplementation of *QuickCheck*.

2.1 Property-Based Testing and *QuickCheck*

Property-based testing is a powerful technique for finding bugs without having to write test cases by hand. Originally introduced by **?** in tandem with the first version of *QuickCheck*, this technique focuses on aiming the developer's efforts into testing systems via executable specifications using randomly generated inputs. Moreover, tools like *QuickChick* and Isabelle's *QuickCheck* demonstrate that PBT can also be used in the formal verification realm. There, one can quickly spot bugs in system specifications before directing the efforts into pointlessly trying to prove bogus propositions.

In the simplest form, there are four main elements the user needs to provide in order to perform property-based testing on their systems:

- one or more *executable properties*, often implemented simply as a boolean predicates,
- *random data generators*, used to repeatedly instantiate the testing properties,
- *printers*, used to show the user the random inputs that falsify some testing property (the counterexample) whenever a bug is found, and
- *shrinkers*, to minimize counterexamples making them easier to understand by humans.

In this work we focus solely on the first two elements introduced above, namely the testing properties and the random data generators used to feed them. Printers and shrinkers, for the most part, can be obtained automatically using generic programming capabilities present in the compiler, and although being crucial for the testing process as a whole, their role becomes irrelevant when it comes to *finding* bugs.

Perhaps the simplest PBT technique is to repeatedly generate random inputs and instantiate the testing properties until they either get falsified by a counterexample, or we ran a sufficiently large amount of tests — suggesting that the properties holds. *QuickCheck* implements a testing loop that closely follows this simple idea, which is outlined in Algorithm 1, where P is the testing property, N is the maximum number of tests to perform, and *gen* is the random generator to be used to instantiate P.

Algorithm 1: *QuickCheck* Testing Loop

```
Function Loop(P, N, gen):
    i ← 0
    while i < N do
        x ← Sample(gen)
        if not P(x) then return Bug(x)
        i ← i+1
    return Ok
```

To illustrate this technique, let us focus on the same motivating example used by Lampropoulos et al., who propose a simple property defined over binary trees. Such data structure can be defined in Haskell using a custom 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 Bool` has type `Tree Bool`, whereas the value `Branch (Leaf 1) 2 (Leaf 3)` has type `Tree Int`. Then, we can define tree reflections using a simple recursive function that pattern matches against the two possible constructors, inverting the order of the subtrees whenever it encounters a branch:

```
mirror :: Tree a -> Tree a
mirror (Leaf x)       = Leaf x
mirror (Branch l x r) = Branch (mirror r) x (mirror l)
```

Later, a reasonable requirement to assert for is that `mirror` must be *involutive*, i.e., reflecting a tree twice always yields the original tree. We can simply capture this property using a boolean predicate written as a normal function:

```
prop_mirror :: Tree Int -> Bool
prop_mirror t = mirror (mirror t) == t
```

For simplicity, here we instantiate the tree payload with integers, altough this predicate should clearly hold for any other type with a properly defined notion of equality as well.

With our simple specification in place, the last missing piece is a random generator of trees. In *QuickCheck*, this is usually done via the type class mechanism, instantiating the `Arbitrary` type class, providing a random generator as the implementation of the overloaded `arbitrary` operation:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized gen
    where
      gen 0 = do { x <- arbitrary; return (Leaf x) }
      gen n = oneof [ do {x <- arbitrary; return (Leaf x) }
                    , do {l <- gen (n-1); x <- arbitrary; r <- gen (n-1); return (Branch l x r)} ]
```

Let us break this definition into parts. The first line states that we will provide an `Arbitrary` instance for trees with payload of type `a`, provided that values of type `a` can also be randomly generated. This allows us to use `arbitrary` to generate `a`'s inside the definition of our tree generator.

Moreover, *QuickCheck* internally keeps track of the *maximum generation size*, a parameter that can be tuned by the user in order to limit the size of the randomly generated values. Our definition exposes this internal parameter via *QuickCheck*'s `sized` combinator, allowing us to parameterize the maximum size of the randomly generated trees. If the generation size is zero (`gen 0`), our generator is limited to produce just leaves with randomly generated payloads. In turn, when the generation size is strictly positive (`gen n`), the generator is able to perform a random uniform choice between generating either a single leaf or a branch. When generating branches, the generator calls itself recursively in order to produce random subtrees (`gen (n-1)`). Notice the importance of reducing the generation size on each recursive call. This way we ensure that randomly generated trees using a generation size `n` are always finite and have at most `n` levels.

Finally, we are ready to let *QuickCheck* test `prop_mirror` against a large number of inputs (100 by default) produced by our brand new random tree generator:

```
quickCheck prop_mirror
+++ OK, passed 100 tests.
```

Should we mistakenly introduce a bug in `mirror`, e.g., by dropping the right subtree altogether:

```
mirror (Branch l x r) = Branch (mirror l) x (mirror l)
```

then *QuickCheck* will quickly falsify `prop_mirror`, reporting a minimized counterexample that we can use to find the root of the issue:

```
quickCheck prop_mirror
*** Failed! Falsified (after 2 tests and 2 shrinks):
Branch (Leaf 0) 0 (Leaf 1)
```

At this point, it is clear that the *quality* of our random generators is paramount to the performance of the overall PBT process. Random generators that rarely produce interesting values will fail to trigger bugs in our code, potentially leaving entire parts of the codebase virtually untested.

Recalling our tree generator, the reader (far from mistaken) might already have imagined better ways for implementing it. For most practical purposes, this generator is in fact quite bad. However, it follows a simple type-directed fashion, and it is a good example of what to expect from a random generator synthesized automatically using a process that knows very little about the values to be generated apart from their (syntactic) data type structure.

As introduced earlier, there exist multiple tools that can automatically derive better random generators solely from the static information present in the codebase. Sadly, these tools lack the domain knowledge required to generate random data with complex invariants — especially those present in programming languages like well-scopedness and well-typedeness.

In particular, automatically derived generators are remarkably uneffective when used to test properties with sparse preconditions. Let us continue with the example by Lampropoulos et al. to illustrate this problem in more detail. For this, consider that we want to use our `Tree` data type to encode binary-search trees (BST) — this requires some minor tweaks in practice. Then, given a predicate `isBST` that asserts if a tree satisfies the BST invariants, we might want to use it as pre- and post-condition to assert that BST operations like `insert` preserve them:

```
prop_bst_insert :: Tree a -> a -> Bool
prop_bst_insert t a =
  isBST t ==> isBST (insert a t)
```

Attempting to test this property using *QuickCheck* does not work well:

```
quickCheck prop_bst_insert
*** Gave up! Passed only 44 tests; 1000 discarded tests.
```

QuickCheck discards random inputs as soon as it finds they do not pass the precondition (`isBST t`). Sadly, most of the inputs generated by our naïve generator suffer from this problem, and the interesting part of the property (`isBST (insert a t)`) is tested very sporadically as a result.

At this point it is reasonable to think that, to obtain the best results when using PBT over complex systems, one is forced to put a large amount of time on developing manually-written generators. In practice, that is most often the case, no automatic effort can beat a well-thought manually-written generator that produces interesting complex values and finds bugs in very few tests. Not all is lost, however. It is still possible to obtain acceptable results automatically by incorporating dynamic information from the system under test into the testing loop. The next subsection introduces the clever technique used by *FuzzChick* to find bugs in complex systems while using simple automatically derived random generators.

2.2 Coverage-Guided Property-Based Testing with *FuzzChick*

To alleviate the problem of testing properties with sparse preconditions while using simple automatically derived random generators, *FuzzChick* introduces *coverage-guided property-based testing* (CGPT) by enhancing the testing process with two key characteristics: (1) *target code instrumentation*, to capture execution information from each test case; and (2) *high-level, well-typed mutations*, to produce syntactically valid test cases by altering existing ones at the datatype level.

Using code instrumentation in tandem with mutations is a well-known technique in the fuzzing community. Generic fuzzing tools like AFL, libFuzzer or HonggFuzz, as well as language-specific ones like Crowbar 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. *FuzzChick*, however, does

this in a clever way. Instead of mutating any previously executed test case that discovers a new part of the code, *FuzzChick* integrates these fuzzing techniques into the PBT testing loop itself.

Since it is possible to distinguish semantically valid test cases from invalid ones, i.e., those passing the sparse preconditions of our testing properties as opossed to those that are discarded early, *FuzzChick* exploits this information in order to focus the testing efforts into mutating valid test cases with a higher priority than those that were discarded.

In addition, high-level mutators are better suited for producing syntactically valid mutants, avoiding the time wasted by using generic low-level mutators that act at the "serialized" level and know very little about the structure of the generated data, thus producing syntactically broken mutants most of the time. This grammar-aware mutation technique has shown to be quite useful when fuzzing systems accepting structurally complex inputs. Tools like Criterion, XSmith and LangFuzz use existing grammars to tailor the generic mutators to the specific input structure used by the system under test. In *FuzzChick*, external grammars are not required. The datatypes used by the inputs of the testing properties already describe the structure of the random data we want to mutate in a concrete manner, and specialized mutators acting at the data constructor level can be automatically derived directly from their definition.

The next subsections describe *FuzzChick*'s testing loop and well-typed mutations in detail.

2.2.1 Testing loop. Outlined in Algorithm 2, the process starts by creating two queues, *QSucc* and *QDisc* for valid and discarded previously executed test cases, respectively. Enqueued values are stored along with a given mutation energy, that controls how many times a given test case can be mutated before being finally discared.

Once inside of the main loop, *FuzzChick* picks the next test case using a simple criterion: if there are valid values enqueued for mutation, it picks the first one, mutates it and returns it, decreasing its energy by one. If *QSucc* is empty, then the same is attempted using *QDisc*. If none of the mutation queues contain any candidates, *FuzzChick* generates a new value from scratch. This selection process is illustrated in detail in Algorithm 3.

Having selected the next test case, the main loop proceeds to execute it, capturing both the result (passed, discarded, or failed) and its execution trace over the system under test. If the test case fails, it is immediately reported as a bug. If not, *FuzzChick* 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 does in fact discover a new path, it is enqueued at the end of its corresponding queue, depending on whether it passed or was discarded. This process alternates between generation and mutation until a bug is found or we reach the test limit.

The energy assigned to each test case follows that of AFL's power schedule: more energy to test cases that lead to shorter executions, or that discover more parts of the code. Moreover, to favour mutating interesting valid test cases, they get more energy than those that were discarded.

2.2.2 Well-typed mutations. Mutators in *FuzzChick* are no more than specialized random generators, parameterized by the original input to be mutated. They use a simple set of mutation operations that are randomly applied at the datatype level. In simple terms, these operations encompass (1) *shrinking the value*, replacing its top-level data constructor with one that contains a subset of its fields, reusing existing subexpressions; (2) *growing the value*, replacing its top-level data constructor with one that contains a superset of its fields, reusing existing subexpressions and generating random ones when needed; (3) *returning a subexpression of the same type*; and (4) *mutating recursively*, applying a mutation operation over an immediate subexpression.

Fig. 1 illustrates a *FuzzChick* mutator for our previously used `Tree` data type example. Since trees are parametric, for clarity this definition is also parameterized by a mutator for the payload (`mutate_a`), although this can be abstracted away using the type class system.

Algorithm 2: *FuzzChick* Testing Loop

```
Function Loop(P, N, gen, mut):
  i ← 0
  TLog, QSucc, QDisc ← ∅;
  while i < N do
    x ← Pick(QSucc, QDisc, gen, mut)
    (result, trace) ← WithTrace(P(x))
    if not result then return Bug(x)
    if Interesting(TLog, trace) then
      e ← Energy(TLog, x, trace)
      if not Discarded(result) then
        Enqueue(QSucc, (x, e))
      else
        Enqueue(QDisc, (x, e))
    i ← i+1
  return Ok
```

Algorithm 3: *FuzzChick* Seed Selection

```
Function Pick(QSucc, QDisc, gen, mut):
  if not Empty(QSucc) then
    (x,e) ← Deque(QSucc)
    if e > 0 then
      PushFront(QSucc, (x, e-1))
    return Sample(mut(x))
  else if not Empty(QDisc) then
    (x,e) ← Deque(QDisc)
    if e > 0 then
      PushFront(QDisc, (x, e-1))
    return Sample(mut(x))
  else return Sample(gen)
```

```
mutate_tree :: (a -> Gen a) -> Tree a -> Gen (Tree a)
mutate_tree mutate_a (Leaf x) =
  oneof [ do { x' <- mutate_a x; return (Leaf x')}                    -- Mutate recursively
        , do { l <- arbitrary; r <- arbitrary; return (Node l x r) } ] -- Grow constructor
mutate_tree (Node l x r) =
  oneof [ return l                                                    -- Return subexpression
        , return r                                                    -- Return subexpression
        , return (Leaf x)                                             -- Shrink constructor
        , do { l' <- mutate_tree l; return (Node l' x r) }            -- Mutate recursively
        , do { x' <- mutate_a    x; return (Node l x' r) }            -- Mutate recursively
        , do { r' <- mutate_tree r; return (Node l x r') } ]          -- Mutate recursively
```

Fig. 1. *FuzzChick* mutator for the `Tree` data type.

In this mutator, branches can be shrinked into leaves by dropping the subtrees, whereas leaves can grow into branches, by reusing the payload and generating two random subtrees. Moreover, branches can be replaced with one of their subtrees. Finally, mutations can be recursively applied over both the payload and the subtrees. At the top level, all these operations are put together using the `oneof` combinator that randomly picks one of them with uniform probability.

2.2.3 Limitations of FuzzChick. Lampropoulos et al. demonstrated empirically that *FuzzChick* lies comfortably in the middle ground between using pure random testing with naïve automatically derived random generators and complex manually-written ones. Their results suggest that CGPT is an appealing technique for finding bugs while still using a mostly automated workflow.

However, the authors acknowledge that certain parts of their implementation have room for improvement, especially when it comes to the mutators design. Moreover, when we recreated the evaluation of the IFC stack machine case study (described in detail in Section 5), we found that after 30 runs (as opposed to the 5 runs used original by Lampropoulos et al.), *FuzzChick was only*

able to find 5 out of the 20 injected bugs with a failure rate of 1, the hardest one being found only around 13% of the time after an hour of testing. These results are presented in detail in Section 6.

At the light of these observations, we identified several aspects of *FuzzChick* that can be improved upon — and that constitute the main goal of this work. In no particular order:

- *Mutators distribution:* if we inspect the mutator defined in Fig. 1, there are two compromises that the authors of *FuzzChick* adopted for the sake of simplicity. On one hand, deep recursive mutations are very unlikely, since their probability decreases multiplicatively with each recursive call. For instance, mutating a subexpression that lies on the third level of a `Tree` happens with a probability smaller than $(1/6)^3 =\sim 0.0046$, and this only worsens as the type of the mutated value becomes more complex. Hence, *FuzzChick* mutators can only be effectively used to transform to shallow data structures, potentially excluding interesting applications that might require producing deeper valid values, e.g., programming languages, network protocols interactions, etc. Ideally, mutations should be able to happen on every subexpression of the input seed in a reasonable basis.
  On the other hand, using random generators to produce neeeded subexpressions when growing data constructors can be dangerous, as we are introducing the very same "uncontrolled" randomness that we wanted to mitigate in the first place! If the random generator produces an invalid subexpression (something quite likely), this might just invalidate the whole mutated test case. We believe that growing data constructors needs to be done carefully. For instance, by using just a minimal piece of data to make the overall mutated test case type correct. If that mutated test case to be interesting, that subexpression can always be mutated later.
- *Enqueuing mutation candidates: FuzzChick* uses two single queues for keeping valid and discarded mutation candidates. 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 tend to grow more often than they tend to shrink, as interesting mutation candidates can get buried at the end of a long queue that only exercises the same portion of the target code. In the extreme case, they might not processed at all within the testing budget. Ideally, one would like a mechanism that prioritizes mutating test cases that discovers new portions of the code right away, and that is capable of jumping back and forth from mutation candidates whenever this happens. We show in Section 4 how this can be achieved by analyzing the execution information in order to prioritize test cases with novel execution traces.
- *Power schedule:* It is not clear how the power schedule used to assign energy to each mutable test case in *FuzzChick* works in the context of high-level well-typed mutations. If it assigns too much energy to certain not-so-interesting seeds, some bugs might not be discovered in a timely basis. Conversely, assigning too little energy to interesting test cases might cause that some bugs cannot be discovered at all unless the right mutation happens within the small available energy window — randomly generating the same test case later on does not help, as it becomes uninteresting based on historic trace information.
  To keep the comparison fair, the authors replicated the same power schedule configuration used in AFL. However, AFL uses a different mutation approach that works at the bit level. This raises the question about what is the best power schedule configuration when using a high-level mutation approach — something quite challenging to characterize in general given the expressivity of the data types used to drive the mutators.

The next section introduces Mutagen, our CGPT tool written in Haskell that aims to tackle the main limitations of *FuzzChick* using an exhaustive mutation approach that requires very little randomness and no power schedule.

3 MUTAGEN: TESTING MUTANTS EXHAUSTIVELY

In this section we describe the base ideas behind Mutagen, our CGPT tool written in Haskell. On top of them, Section 4 introduces heuristics that help finding bugs faster in certain testing scenarios.

In constrast with *FuzzChick*, Mutagen does not employ a power schedule to assign energy to mutable candidates. In turn, it resorts to mutate them in an exhaustive and precise basis, where (1) each subexpression of a mutation candidate is associated with a set of deterministic mutations, and (2) for every mutable subexpression, each of these mutations is evaluated *exactly once*. There is a small exception to this rules that we will introduce soon.

This idea is inspired by exhaustive bounded testing tools like SmallCheck (in Haskell) or Korat (in Java), that produce test cases exhaustively. In simple words, such tools work by enumerating all possible values of the datatypes used in the testing properties, and then executing them from smaller to larger until a a bug is found, or a certain size bound is reached. The main problem with this approach is that the space of all possible test cases often grows exponentially as we increment the size bound, and the user experiences what it looks like "hitting a wall", where no larger test cases can be evaluated until we exhausted all the immediately smaller ones [**?**]. In consequence, such tools can only be applied to relatively simple systems, where the space of inputs does not grow extremely fast.

Not to be confused by these tools, in Mutagen we do not enumerate all possible test cases exhaustively. Mutagen uses random generators to find interesting initial seeds, and only then proceeds to execute all possible mutations. Moreover, these mutants will be mutated further only if they discover new paths in the target code, so the testing loop automatically prunes the space of test cases that are worth mutating.

3.1 Exhaustive Mutations

To describe Mutagen's testing loop, we first need to introduce the mechanism used for testing mutations exhaustively. In contrast to *FuzzChick*, where mutators are parameterized random generators, in Mutagen we define mutations as the set of mutants that can be obtained by altering the input value at the top-level (the root). In Haskell, we define mutations as:

```
type Mutation a = a -> [Mutant a]
```

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

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

Pure mutants are used most of the time, and encode simple deterministic transformations over the top-level data constructor of the input — recursive mutations will be introduced soon. These tranformation can: (1) return an immediate subexpression of the same type as the input; (2) swap the top-level 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. 2 outlines a mutator for the `Tree` data type. Notice how this definition simply enumerates mutants that transform the top-level 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 smallest value we can construct in order for the mutant to be type-correct. In practice (`def = Leaf def`), where the inner `def` is the smallest value of the payload — we use the type class system to abstract this complexity away in our implementation.

```
mutate (Leaf x)     = [ PURE (Node def x def) ]   -- Swap constructor
mutate (Node l x r) = [ PURE l, PURE r            -- Return subexpression
                      , PURE (Leaf x)             -- Swap constructor
                      , PURE (Node l x l)         -- Rearrange subexpressions
                      , PURE (Node r x r)         -- Rearrange subexpressions
                      , PURE (Node r x l) ]       -- Rearrange subexpressions
```

Fig. 2. Mutagen mutator for the `Tree` data type.

Using a default smallest value is again inspired by exhaustive bounded testing tools, and avoids introducing unnecesary randomness when growing data constructors.

At this point, the reader might consider: what about large enumeration types like integers or characters? Does Mutagen transform them exaustively too? *Certainly not.* Random mutants serve as a way to break exhaustiveness when mutating values of large enumeration types. Instead of trying every possible integer or character, we resolve in using a random generator and sample a small amount of values from it — the precise amount is a tunable parameter of Mutagen. This way, a mutator for integers simply becomes:

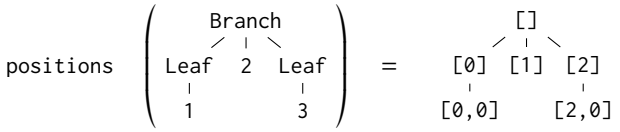
```
mutate n = [ RAND arbitrary ]
```

3.1.1 Mapping top-level mutations everywhere. So far we defined mutations that transform the root of the input. Now it is time to apply these mutations to every subexpression as well. To do so, we will use two functions that can also be derived from the data type definition.

In first place, a function `positions` traverses the mutable candidate and builds a multi-way tree of mutable positions. These positions are essentially a list of indices encoding the path from the root to every mutable subexpression. For instance, the positions of a `Tree` are computed as follows:

```
positions (Leaf x)       = node [ (0, positions x) ]
positions (Branch l x r) = node [ (0, positions l), (1, positions x), (2, positions r) ]
```

In this light, the mutable positions of the value `Branch (Leaf 1) 2 (Leaf 3)` are:



Later, given a desired position to mutate within a candidate, we define another function `inside` that precisely finds the subexpression corresponding to it and applies the mutation. A slightly simplified version of this function for the `Tree` datatype is as follows:

```
inside []       x              = mutate x
inside (0 : pos) (Leaf x)      = [ Leaf x'       | x' <- inside pos x ]
inside (0 : pos) (Branch l x r) = [ Branch l' x r | l' <- inside pos l ]
inside (1 : pos) (Branch l x r) = [ Branch l x' r | x' <- inside pos x ]
inside (2 : pos) (Branch l x r) = [ Branch l x r' | r' <- inside pos r ]
```

This function simply traverses the desired position, calling itself recursively until it reaches the desired subexpression, where the mutation can be applied locally at the top-level (case `inside [] x`). The rest of the function takes care of unwrapping and rewrapping the intermediate subexpressions and is not partically relevant for the point being made.

3.2 Testing loop

Having the exhaustive mutation mechanism in place, we can finally introduce the base testing loop used by Mutagen. This is outlined in Algorithm 5. As it can be observed, we closely follow

FuzzChick's testing loop, using qtwo queues to keep mutant candidates, and enqueueing and retrieving interesting test cases from them before falling back to random generation.

The main difference lies in that we precompute all the mutations of a given mutation candidate before enqueueing them. These mutations are put together into lists that we call *mutation batches* — one for each mutation candidate. To initialize a mutation batch (outlined in Algorithm 4), we first flatten all the mutable positions of the input value in level order (recall that positions are stored as a multi-way 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 posible 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., 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 accummulated batch.

Algorithm 4: Mutants Initialization

```
Function Mutate(x, mut, R):
    muts ← ∅
    for pos in Flatten(Positions(x)) do
        for mutant in Inside(pos, mut, x) do
            switch mutant do
                case PURE x̂ do
                    Enqueue(x̂, muts)
                case RAND gen do
                    repeat R times
                        x̂ ← Sample(gen)
                        Enqueue(x̂, muts)
    return muts
```

Finally, the seed selection algorithm (Algorithm 6) simply selects the next test case using the same criteria as *FuzzChick*, prioritizing valid candidates over discarded ones, falling back to random generation when both queues are empty. Since mutations are precomputed, this function only needs to pick the next test case from the current batch, until it becomes empty and can switch to the next precomputed one in line.

Algorithm 5: MUTAGEN Testing Loop

```
Function Loop(P, N, R, gen, mut):
    i ← 0
    TLog, QSucc, QDisc ← ∅;
    while i < N do
        x ← Pick(QSucc, QDisc, gen, mut)
        (result, trace) ← WithTrace(P(x))
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            if not Discarded(result) then
                muts ← Mutants(x, mut, R)
                Enqueue(QSucc, muts)
            else if Passed(Parent(x)) then
                muts ← Mutate(x, mut, R)
                Enqueue(QDisc, muts)
        i ← i+1
    return Ok
```

Algorithm 6: MUTAGEN Seed Selection

```
Function Pick(QSucc, QDisc, gen):
    if not Empty(QSucc) then
        muts ← Deque(QSucc)
        if Empty(muts) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, Rest(muts))
            return First(muts)
    if not Empty(QSucc) then
        muts ← Deque(QSucc)
        if Empty(muts) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, Rest(muts))
            return First(muts)
    else return Sample(gen)
```

Another small difference between MUTAGEN and *FuzzChick* is the criteria for enqueuing discarded tests. We found that, especially for large data types, the queue of discarded candidates tends to grow disproportionately 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 mutation candidates. Moreover, this can potentially help introducing *2-step mutations*, where an initial mutation breaks a valid test case in a small way, it gets enqueued as discarded, and later a subsequent mutation fixes it by changing a different subexpression.

The next section introduces two heurisitics we added to the base testing loop of MUTAGEN based on the limitations we found in *FuzzChick*.

4 MUTAGEN HEURISTICS

In this section we introduce two heuristics implemented on top of the base testing loop of our tool. MUTAGEN enables them all by default, although they can be individually disabled by the user if deemed appropriate.

4.1 Priority FIFO Scheduling

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

FuzzChick uses AFL instrumentation under the hood, which in turn uses an *edge coverage* criteria to distinguish novel executions and to assign each mutation candidate a given energy. In contrast, execution traces MUTAGEN represent the *path* in the code taken by the program, as opposed to just the set of edges traversed in the control-flow graph (CFG). Using this criteria lets us gather precise information from the each new execution. In particular, we are interested in the *depth* where each new execution branches from already seen ones. Our assumption here is that test cases that differ (branch) at shallower depths from the ones already executed are more likely to discover completely new portions of the code under test, and hence we want to assign them a higher priority.

In this light, every time we insert a new execution path into the internal trace log, 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. 3 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 $1 \longrightarrow 2 \longrightarrow 6 \longrightarrow 7$) branches at a shallower depth than the first one (2 vs. 3), hence its corresponding test case should be given a higher priority.

With this mechanism in place, 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

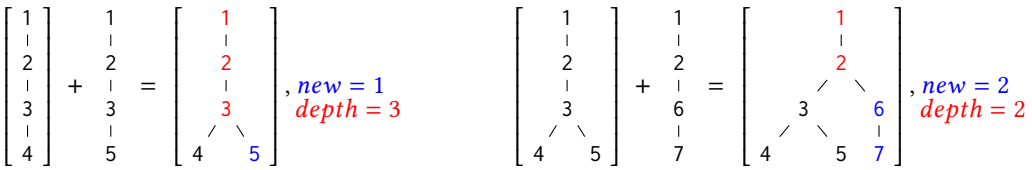


Fig. 3. Inserting two new execution traces into the internal trace log (represented using brackets).

Algorithm 7: Priority FIFO Heuristic

```
Function Loop(P, N, R, gen, mut):
    ...
    x ← Pick(QSucc, QDisc, gen, mut)
    (result, trace) ← WithTrace(P(x))
    ...
    if Interesting(TLog, trace) then
        if not Discarded(result) then
            muts ← Mutants(x, mut, R)
            prio ← BranchDepth(TLog, trace)
            PushFront(QSucc, prio, muts)
        ...
Function Pick(QSucc, QDisc, gen):
    if not Empty(QSucc) then
        (muts, prio) ← DequeMin(QSucc)
        if Empty(muts) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, prio, Rest(muts))
            return First(muts)
    ...
```

Algorithm 8: Trace Saturation Heuristic

```
Function Loop(P, N, gen, mut):
    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
        ...
```

changes are illustrated in Algorithm 7. Statements in red indicate important 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 simply retrieve the one with highest priority (the smallest branching depth). Then, whenever we find a new interesting test case, we enqueue it at the beginning of the queue of its corresponding priority. This allows the testing loop to jump immediately onto processing new interesting candidates as soon as they are found (even at the same priority), and to jump back to previous candidates as soon as their mutants become progressively less novel.

4.2 Detecting Trace Space Saturation And Tuning Random Mutations

As introduced in Section 3, our tool is parameterized by the amount of random mutations to be generated over each mutable subexpression defined using a random mutant, e.g., numeric values, characters, etc. But, how many random mutations should we use? A single one? A few tenths? A few hundreds? Clearly, using too little, can put finding bugs at risk. For instance, when the system under test branches on numeric values, we should make sure that we set enough random mutations to test each case in a reasonable basis. Using too many, the other hand, can degrade the performance of the testing loop, as it will spend too much time producing uninteresting mutations. This can happen for instance if the subexpressions defined using random mutants are only used as payloads, and their value does not affect the execution in any way.

Answering this question precisely is not an easy task, and the second heuristic we introduce in this work 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. Our realization is that we can use this information to automatically optimize the number of random mutations used by our tool. This idea is described in Algorithm 8. The process is simple: (1) we start the testing loop with the amount 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 amount 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 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 continuosly increase the random mutations parameter until that stops happening.

5 CASE STUDIES

We evaluated the performance of Mutagen using two main case studies. In both of them, we show how the heuristics described in Section 4 affect the testing performance.

The first one is a simple abstract stack machine that enforces *noninterference* using runtime checks. The implementation of this case study was originally proven correct by [] in Coq, and subsequently degraded by systematically introducing 20 bugs on its enforcing mechanism. Lampropoulos et al. used this same case study to compare *FuzzChick* against random testing approaches using naïve and manually-written smart random generators. In this work we replicate their results and compare them against our tool.

The second case study aims to evaluate Mutagen in a more realistic scenario, and focuses on testing *haskell-wasm*, an existing WebAssembly engine written in Haskell of industrial strength. We manually injected 10 bugs in the validator as well as 5 bugs in the interpreter of this engine, and used the reference implementation to find them using differential testing. During this process, we quickly discovered 3 bugs and 2 discrepancies on this engine with respect to the reference implementation. All of them were reported and confirmed by the authors of *haskell-wasm*.

5.1 IFC Stack Machine

The abstract stack machine used in this case study consists of four main elements: a program counter, a stack, and data- and instruction memories. Moreover, every runtime value is labeled with its 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. Security labels are propagated throughout the execution of the program every time the machine executes an instruction. There are eight different instructions defined as:

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

Control flow is achieved using the `Call` and `Ret` instructions, that let the program jump back and forth within the instruction memory using specially labelled values in the stack respresenting 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. To preserve space, we encourage the reader to refer to the work of **?** as well as the original *FuzzChick* paper for more details about the implementation and semantics of this case study.

5.1.1 Noninterference.

5.2 Single-step Noninterference

5.3 WebAssembly Engine

WebAssembly is a popular assembly-like language designed to be an open standard for executing low-level code in the web, although it has become increasingly popular in standalone, non-web contexts as well. WebAssembly programs are first validated and later executed in a sandboxed environment (a virtual machine), making this language an attractive target for virtualization in *functions-as-a-service* platforms. The language is relatively simple, in esence (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 writing functions 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 dinamically; among others. To give the reader a taste, Fig. 4 the WebAssembly text representation of module that exports a recursive implementation of the factorial function.

Unlike most other programming languages, its behavior is fully specified, and WebAssembly programs are expected to be consistently interpreted across engines — despite some subtle details that we will address soon. For this purpose, the WebAssembly standard provides a reference implementation with all the basic functionality expected from a compliant WebAssembly engine.

```
(module
(func $f (param f64) (result f64)
  get_local 0
  f64.const 1
  f64.lt
  if (result f64)
    f64.const 1
  else
    get_local 0
    get_local 0
    f64.const 1
    f64.sub
    call $fact
    f64.mul
  end)
(export "fact" (func $f)))
```

Fig. 4. Factorial in WebAssembly

In this work, we are interested on 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. Our tool is an attractive match for testing *haskell-wasm*, as the space of WebAssembly programs contains mostly invalid ones, and automatically derived random generators cannot satisfy all the invariants required to produce interesting test cases. Here, we avoided spending countless hours writing an extensive property-based specification to mimic the reference WebAssembly specification. Instead, we take advantage of the readily available reference implementation via differential testing. In this light, our testing properties assert that any result produced by *haskell-wasm* matches that of the reference implementation.

Unsurprisingly, this engine had several subtle latent bugs that were not caught by the existing unit tests and that we discovered using Mutagen while developing the test suite used in this work. Moreover, Mutagen exposed two discrepancies between *haskell-wasm* and the reference implementation. Not severe enough to be classified as bugs, these discrepancies trigger parts of the WebAssembly specification that are either not yet suported by the reference implementation (multi-value blocks), or that produce a well-known non-deterministic undefined behavior allowed by the specification (NaN reinterpretation). These findings are briefly outlined in Table 1.

The rest of this section introduces the test suite we used to test the different parts of *haskell-wasm*.

5.3.1 Testing the WebAssembly Validator. We begin by designing a property to test the WebAssembly validator implemented in *haskell-wasm*. To keep things simple, we can simply assert that, whenever a randomly generated (or mutated for that matter) WebAssembly module is valid according to *haskell-wasm*, then the reference implementation agrees upon it. In other words, we are testing for false positives. In Haskell, we write the following testing property:

```
prop_validator m =
  isValidHaskell m ==> isValidSpec m
```

Where the precondition (`isValidHaskell m`) runs the input WebAssembly module `m` against the *haskell-wasm* validator, whereas the postcondition (`isValidSpec m`) serializes `m` to a file, runs it against the reference implementation validator and checks that no errors are produced.

We want to remark that, although here for simplicity we only focus on finding false positives, in a realistic test suite, one would also want to test for false negatives, i.e., when a module is valid and *haskell-wasm* rejects it. This can be easily done by inverting the direction of the implication (`<==`) in the property above. However, the resulting property is much slower, as every tested module will be serialized and run against the reference implementation.

5.3.2 Testing the WebAssembly Interpreter. Testing the WebAssembly interpreter is substantially more complicated than testing the validator, as it entails running actual programs. To achieve this, we need the generated test cases to comply a with stable interface that can be invoked both by *haskell-wasm* and the reference WebAssembly implementation.

To keep things simple here as well, we use a stub definition that provides a module that initializes a memory block and exports a single function. This module can be instantiated by providing the definition of the single function, along with a name and a type signature. In Haskell:

```
mk_module ty name fun =
  emptyModule { types     = [ ty ]
              , functions = [ fun ]
              , exports   = [ Export name (ExportFunc 0) ]
              , mems      = [ Memory (Limit 1 Nothing) ]
              }
```

Using this stub module, we can define a testing property parameterized by the function type signature, the function implementation and a list of invocation arguments:

```
prop_interpreter ty fun args =
  (discardAfter 20)
  (do let m = mk_module ty "f" fun;
      resHs   <- invokeHaskell m "f" args
      resSpec <- invokeSpec    m "f" args
      return (equivalent resHs resSpec))
```

This property instantiates the module stub using the input function and its type signature, and uses it to invoke both the *haskell-wasm* and reference implementation interepreters with the provided arguments. Then, the property asserts whether their results are equivalent. Interestingly, equivalence in this context 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 acount. Notice that we additionally set a 20ms timeout to discard potentially diverging programs with infinite loops.

Using this testing property directly might not sound like a great idea, as randomly generated lists of inputs 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 bug #3 in *haskell-wasm* mentioned above. Having solved this issue in *haskell-wasm*,

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 proceeds after arity mismatch
4	Interpreter	Bug	Allowed out-of-bounds memory access
5	Interpreter	Discrepancy	NaN reinterpretation does not follow reference implementation

Table 1. Bugs and discrepancies found by Mutagen in *haskell-wasm*.

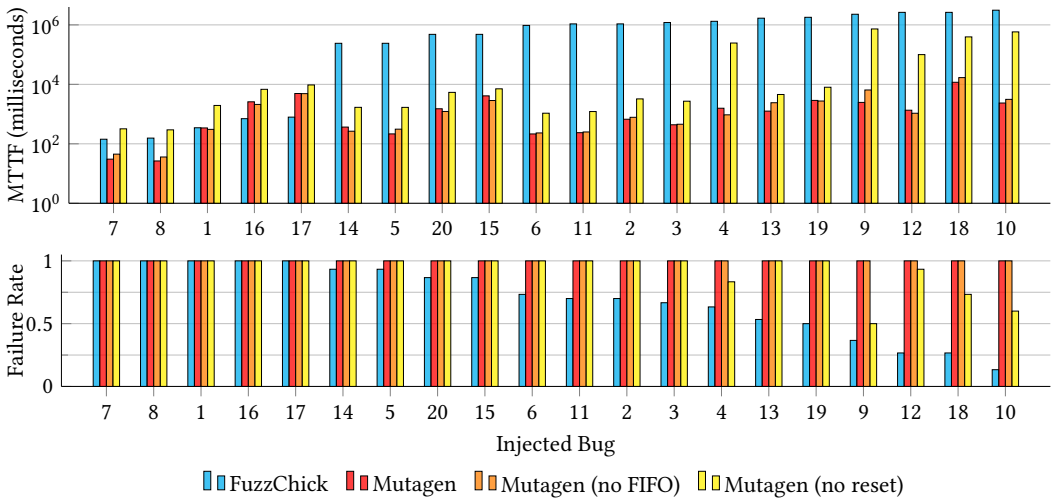


Fig. 5. Comparison between *FuzzChick* and MUTAGEN across 20 different bugs for the IFC stack machine.

we proceed to 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]
```

This specilized property let us generate functions using this fixed type and invoke them with the exact number and type of arguments required. In our experiments (presented in next section), we use this property when finding all the injected bugs into the *haskell-wasm* interpreter. Worth mentioning again, a realistic test suite should at least include different variants of this property testing functions of several different types.

6 EVALUATION

All the experiments were done in a dedicated workstation with an Intel Core i7-8700 CPU running at 3.20GHz, and equipped with 32Gb of RAM. We ran each experiment 30 times except for the ones involving the bugs on the WebAssembly interpreter, which were run 10 times. From there, we followed the same 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 time. In all cases, we used a one-hour timeout to stop the execution of both tools if they have not yet found a counterexample.

Additionally, 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 bugdet. We found this metric crucial to be analyzed when replicating *FuzzChick*'s results, as opposed to just paying attention at the MTTF.

6.1 IFC Stack Machine

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

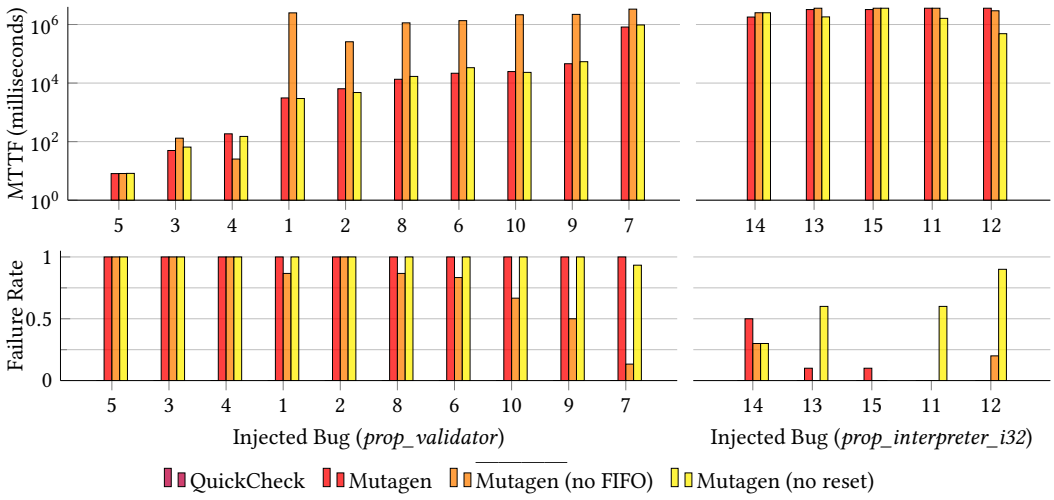


Fig. 6. Comparison between *QuickCheck* and MUTAGEN across 15 different bugs for the WebAssembly case study.

6.2 WebAssembly Engine

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

7 RELATED WORK

7.1 Grammar-based Fuzzing

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

7.2 Random Data Generation

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

7.3 Exhaustive Bounded Testing

Aliquam erat volutpat [Dévai et al. 2013]. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

8 CONCLUSIONS

We presented MUTAGEN, a coverage-guided, property-based testing framework written in Haskell. Inspired by *FuzzChick*, our tool uses coverage information to guide automatically derived mutators producing high-level, well-typed mutations. However, instead of relying heavily on randomness and power schedules to find bugs, our tool uses an exhaustive mutation approach that generates every possible mutant for each interesting input candidate, and schedules it to be tested exactly once. This is in turn inspired by exhaustive bounded testing tools that focus on testing every possible input value of the system under test — a more generic technique of limited applicability.

Our experimental results indicate that MUTAGEN outperforms the simpler approach taken by *FuzzChick* in terms of failure rate. Moreover, we show how our tool can be applied in a real-world testing scenario, where it quickly discovers several planted and existent previously unknown bugs.

In the future, we will investigate how to redefine our automatically synthesized mutators in a stateful manner. This way, it would be possible to apply mutations that preserve complex properties of the generated data like well-scopedness and well-typedness simply by construction, e.g., mutations that *always* produce well-typed subexpressions that refer only to identifiers in the current scope. The main challenge will be to achieve this while keeping the testing process as automatable as possible.

REFERENCES

Gergely Dévai, Dániel Leskó, and Máté Tejfel. 2013. The EDSL's Struggle for Their Sources. In *Central European Functional Programming School*. Springer, 300–335.

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.