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

ANONYMOUS AUTHOR(S)

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

FuzzChick is a recent tool that borrows ideas from the fuzzing community to alleviate this limitation. It enhances automatically synthesized generators with type-preserving mutations guided by execution traces. Despite being enlighthening, we observed several limitations in the implementation of *FuzzChick* that reduce its applicability. These limitations include a large reliance on randomness when mutating test cases, as well as poor scheduling in certain situations, among others.

In this work we present Mutagen, a coverage-guided, property-based testing framework that builds upon the ideas behind *FuzzChick*. Instead of relying heavily on randomness, Mutagen uses an exhaustive approach to generate new mutants. This is coupled with heuristics that help to schedule mutants based on their novelty and to minimize mutating subexpressions of large enumeration types more than necessary. Our results indicate that our approach is capable of outperforming the simpler one used by *FuzzChick*, as well as finding subtle bugs in real-world scenarios.

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

ACM Reference Format:
Anonymous Author(s). 2021. Mutagen: Coverage-Guided, Property-Based Testing using Exhaustive Type-Preserving Mutations. In *Proceedings of OOPSLA '21: ACM SIGPLAN conference on Systems, Programming, Languages, and Applications (OOPSLA '21).* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Property-Based Testing (PBT) is a popular technique for finding bugs using executable properties that encode the expected behavior of our systems. Exemplified by frameworks like Haskell's *QuickCheck* [Claessen and Hughes 2000] and the abundant ports of this tool in other programming languages [Bulwahn 2012; Dénès et al. 2014; Hughes 2003; Papadakis and Sagonas 2011], the most common PBT approach is to instantiate the testing properties using randomly generated inputs.

Users of *QuickCheck* (and PBT in general) are well aware that one of the biggest challenges while using these frameworks is to provide suitable random data generators needed to instantiate the testing properties. In some 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 [Lampropoulos et al. 2019].

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

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. [Bendkowski et al. 2017; Duregård et al. 2012; Grieco et al. 2016; Lampropoulos et al. 2017b; Mista and Russo 2019; Mista et al. 2018]. 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 [Pałka et al. 2011; Perényi and Midtgaard 2020; Yang et al. 2011]. 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 discarded before they can be used to test the main components of our system.

FuzzChick [Lampropoulos et al. 2019] is a property-based testing framework for the Coq programming language that borrows ideas from the fuzzing community to generate highly structured values while using automatically derived generators. Notably, this tool is implemented as an extension of *QuickChick* [Dénès et al. 2014], Coq's own reimplementation of *QuickCheck*.

Instead of continuously generating random invalid test cases from scratch, *FuzzChick* keeps a queue of interesting previously executed test cases that can be mutated using type-preserving transformations to produce new ones. Intuitively, mutating an interesting test case in a small way (at the data constructor level) has a much higher probability of producing a new interesting test case than generating a new one from scratch using a naïve generator. In this light, *FuzzChick* is likely to preserve the semantic structure of the mutated data, as mutations are applied directly at the data type level, e.g., the program's AST in case of generating code.

This tool relies on execution traces to distinguish which test cases resulted interesting and are therefore worth mutating — a prolific technique known as *coverage-guided fuzzing* and popularized by tools like *AFL* [M. Zalewski 2010]. Mutated test cases are considered interesting for mutation only when they produce new execution traces — any other test case is simply thrown away.

Of course, this hybrid technique is not meant to replace smart manually-written generators. In turn, it provides an acceptable solution that can be used in the early stages of development, and while a better test suite using manually-written generators is still under construction.

In this work, we take a close look at *FuzzChick* for opportunities to improve. Notably, we establish several aspects of its implementation that can be revised — some of these already being identified by the authors of *FuzzChick* in their original work. In particular: (1) *the type-preserving mutations produced by this tool are superficial and rely heavily on randomness to produce diverse mutants*, (2) *using normal queues can cause large delays when interesting values are enqueued frequently*, and (3) *the mechanism used to assign a certain mutation energy to each interesting test case requires fine tuning and can be hard to generalize to work well under different testing scenarios.*

Moreover, we replicated the IFC stack machine case study bundled with *FuzzChick* and observed a noticeable limitation: *after repeating each experiment 30 times, FuzzChick was only able to 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 limitation, Lampropoulos et al. show that *FuzzChick* is far better than using normal random testing coupled with naïve random generators by comparing the mean-time-to-failure (MTTF) against *QuickChick*. However, we believe that a proper evaluation ought to use the failure rate as an important metric when comparing PBT tools.

In this work we introduce Mutagen, a PBT tool implemented in Haskell that builds upon the basic ideas behind *FuzzChick*, i.e., code instrumentation and type-preserving mutations. Mutagen is introduced in detail in Section 3.

Unlike *FuzzChick*, mutations in Mutagen are applied *exhaustively*. This is, given an interesting, previously executed test case, our tool precomputes and schedules every type-preserving mutation that can be applied to it. This approach has two inherent advantages when compared to *FuzzChick*. 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 left behind due to randomness. On the other hand, scheduling mutations exhaustively eliminates the need of a heuristic that assigns a given mutation energy to each input test case.

Notably, *scheduling mutations exhaustively within an input test case does not mean that mutations are exhaustive by themselves.* Our tool distinguishes two kinds of mutations, those that can be computed purely, yielding a single deterministic mutated value; and those that can be obtained non-deterministically. On one hand, pure 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 exhaustiveness of our tool by randomly sampling a small random generator a reduced number of times. 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 our tool incorporates two heuristics that help finding bugs faster and more reliably (Section 4). In first place, Mutagen 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. Additionally, our scheduling lets the testing loop 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 is used to tune 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 automatically 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 main case studies (described in detail in Section 5): the IFC stack machine used by Lampropoulos et al. on their original work, and a WebAssembly engine written in 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 *FuzzChick* against different variants of our tool, where our results (Section 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 more 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. All in all, our evaluation encompass more than 600 hours of cpu time and suggest that testing mutants exhaustively can be an appealing technique for finding bugs reliably without sacrificing speed.

Finally, we discuss related work on Section 7 and conclude on Section 8.

2 BACKGROUND

In this section, we briefly introduce the concept of Property-Based Testing along with *QuickCheck*, one of the most popular tools of this sort and often used as the baseline when comparing PBT algorithms. Moreover, we describe in detail the ideas and limitations behind *FuzzChick*, which served as the foundation while designing Mutagen.

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. Popularized by Haskell's *QuickCheck*, this technique focuses on aiming the developer's efforts into testing systems via executable properties using randomly generated inputs.

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 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 often 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 use them 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 hold. *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 True` 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, although this predicate should hold for any other type with a properly defined notion of equality as well.

The last missing piece is a random generator of trees. In *QuickCheck*, this is commonly done via the type class mechanism [Jones et al. 1997], instantiating the Arbitrary type class with 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 a 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 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 can 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 ineffective 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 into 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 automatically derived random generators, *FuzzChick* introduces *coverage-guided, property-based testing* (CGPT), a technique that enhances the testing process with two key characteristics: (1) *target code instrumentation*, to capture execution information from each test case; and (2) *high-level, type-preserving mutations*, to produce syntactically valid test cases by altering existing ones at the data type level.

Using code instrumentation in tandem with mutations is a well-known technique in the fuzzing community. Generic fuzzing tools like *AFL*, *libFuzzer* [2019] or *HonggFuzz* [Google 2010], as well as language-specific ones like Crowbar [Dolan and Preston 2017] or *Kelinci* [Kersten et al. 2017] 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 opposed 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 LangFuzz [Holler et al. 2012], Superion [Wang et al. 2019], XSmith [Hatch et al. 2020] use existing grammars to tailor the mutators to the specific input structure used by the system under test. In *FuzzChick*, external grammars are not required. The data types used by the inputs of the testing properties already describe the

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)
```

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 type-preserving 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 thrown away.

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 (i.e., it failed the precondition). 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 favor mutating interesting valid test cases, they get more energy than those that were discarded.

2.2.2 Type-preserving 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 data type level. In simple terms, these operations encompass (1) *shrinking the value*, replacing its outermost data constructor with one that contains

```
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 (Branch l x r) } ] -- Grow constructor
mutate_tree mutate_a (Branch l x r) =
  oneof [ return l                                                       -- Return subexpression
        , return r                                                       -- Return subexpression
        , return (Leaf x)                                                -- Shrink constructor
        , do { l' <- mutate_tree l; return (Branch l' x r) }             -- Mutate recursively
        , do { x' <- mutate_a    x; return (Branch l x' r) }             -- Mutate recursively
        , do { r' <- mutate_tree r; return (Branch l x r') } ]           -- Mutate recursively
```

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

a subset of its fields, reusing existing subexpressions; (2) *growing the value*, replacing its outermost 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*-like 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.

In this mutator, branches can be shrunk 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 its implementation have room for improvement, especially when it comes to the mutator's design. In tandem with the lack of efficacy observed when replicating the evaluation of the IFC stack machine case study, these observations led us to consider three main 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 fourth level of a Tree happens with a probability smaller than $(1/6)^3$ =~ 0.0046, i.e., when applying a recursive mutation (with probability 1/6) three consecutive times. 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., programs, network protocol interactions, etc. Ideally, mutations should be able to happen on every subexpression of the input seed on a reasonable basis.

On the other hand, using random generators to produce needed 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 turns 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 an extreme case, they might not be processed at all within the testing budget. Ideally, one would like a mechanism that prioritizes mutating test cases that discover new portions of the code right away, and that is capable of jumping back and forth from mutation candidates whenever this happens.

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 type-preserving mutations. If it assigns too much energy to certain not-so-interesting seeds, some bugs might not be discovered on 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 again later 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. Despite the choice of implementation language, the ideas presented in this work extend beyond Haskell and functional programming languages in general.

In contrast with *FuzzChick*, Mutagen does not employ a power schedule to assign energy to mutable candidates. In turn, it resorts to mutate them on an exhaustive and precise manner, where (1) each subexpression of a mutation candidate is associated with a set of type-preserving mutations, and (2) for every mutable subexpression, each one of these mutations is evaluated *exactly once*.

This idea is inspired by exhaustive bounded testing tools like SmallCheck [Runciman et al. 2008] or Korat [Boyapati et al. 2002], that produce test cases exhaustively. In simple words, such tools work by enumerating all possible values of the data types used in the testing properties, and then executing them from smaller to larger until 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 [Duregård et al. 2012]. To alleviate this problem, such tools often rely on heuristics to prune the search space based on detecting unevaluated subexpressions — this is discussed in Section 7.

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

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

Not to be confused by these tools, in Mutagen we do not enumerate all possible test cases exhaustively. Our tool uses random generators to find interesting initial seeds, and when it finds them, only then proceeds to schedule all possible mutations. Moreover, like in *FuzzChick*, the testing loop of our tool automatically filters the test cases that are worth mutating by keeping only those that discover new execution paths in the code under test.

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 our tool 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 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. 2 outlines a mutator for the `Tree` data type. Notice how 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 practice (`def = Leaf def`), where the inner `def` is the simplest value of the payload — we use the type class system to abstract this complexity away in our implementation. Using a small value is again inspired by exhaustive bounded testing tools, and avoids introducing unnecessary randomness when growing data constructors.

Random mutants, on the other hand, serve as a way to break exhaustiveness when mutating values of large enumeration types — or any other type the user might want to use random mutations with. Instead of mutating into every other possible number or character, we resolve in using a random generator and sample a small number 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 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 will use two functions that can be derived from the data type definition.

In first place, a function `positions` traverses the input and builds a rose 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) ]
```

where the function node simply builds a node of the positions rose tree and propagates the accumulated prefix (the path from the root) to its children. In this light, the mutable positions of the value `Branch (Leaf 1) 2 (Leaf 3)` are:

```
           (    Branch    )           []
           (    /  |  \   )          / | \
positions  ( Leaf  2  Leaf )  =   [0] [1] [2]
           (  |        |  )        |       |
           (  1        3  )      [0,0]   [2,0]
```

Later, given the desired position to mutate within a test case, we define another function `inside` that finds the subexpression corresponding to it and applies the mutation. A slightly simplified version of this function for the `Tree` data type 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 particularly 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 two queues to keep mutable 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 mutated test case. 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 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.

Algorithm 4: Mutants Initialization

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

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

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)
        (result, trace) ← WithTrace(P(x))
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            if not Discarded(result) then
                batch ← Mutate(x, mut, R)
                Enqueue(QSucc, batch)
            else if Passed(Parent(x)) then
                batch ← Mutate(x, mut, R)
                Enqueue(QDisc, batch)
        i ← i+1
    return Ok
```

Algorithm 6: Mutagen Seed Selection

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

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.

Another small difference between Mutagen and *FuzzChick* 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 mutation candidates. Moreover, this can potentially lead to *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 heuristics 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 deems appropriate.

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.

FuzzChick uses AFL instrumentation under the hood, which in turn uses an *edge coverage* criterion to distinguish novel executions and to assign each mutation candidate a given energy. In contrast, execution traces 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). 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 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 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 the 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 Parameter

As introduced in Section 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, characters, etc. But, how many random mutations should we use? A single one? A few tenths? A few hundred? Using too little can put finding bugs at risk. For instance, when the system under test branches based on numeric values, we should make sure that we set enough random mutations to test each branch on a reasonable basis. Using too many, on 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. 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 8. 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

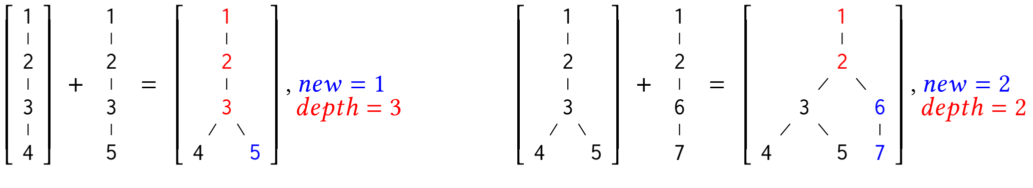

Fig. 3. Inserting two new execution traces into the internal trace log (represented using brackets).

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
            batch ← Mutate(x, mut, R)
            prio ← BranchDepth(TLog, trace)
            PushFront(QSucc, prio, batch)
        ...

Function Pick(QSucc, QDisc, gen):
    if not Empty(QSucc) then
        (batch, prio) ← DequeMin(QSucc)
        if Empty(batch) then
            Pick(QSucc, QDisc, gen)
        else
            PushFront(QSucc, prio, Rest(batch))
            return First(batch)
    ...
```

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
        ...
```

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 continuously increase the random mutations parameter until that stops happening.

5 CASE STUDIES

We evaluated the performance of Mutagen using two main case studies. The first one is a simple abstract stack machine that enforces *noninterference* [Goguen and Meseguer 1982] using runtime checks. The implementation of this case study was originally 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 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. Worth mentioning, since this case study was originally implemented in Coq, we first translated it to Haskell in order to run Mutagen on its test suite.

The second case study aims to evaluate Mutagen in a more realistic scenario, and focuses on testing *haskell-wasm* [Rezvov 2018], an existing WebAssembly engine of industrial strength written in Haskell. We manually injected 10 bugs in the validator as well as 5 bugs in the interpreter of this engine, and used the reference WebAssembly implementation to find them via differential

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 opertations
8	Validator	Removed memory index out-of-range assertion
9	Validator	Wrong type validation on `i64` constants
10	Validator	Removed memory alignment validation on `i32.load` instruction
11	Interpreter	Wrong intepretation of `i32.sub` instruction
12	Interpreter	Wrong intepretation of `i32.lt_u` instruction when operands are equal
13	Interpreter	Wrong intepretation of `i32.shr_u` instruction
14	Interpreter	Wrong local variable initialization
15	Interpreter	Wrong memory address casting on `i32.load8_s` instruction

Table 1. Bugs injected into *haskell-wasm*.

testing [McKeeman 1998]. 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 later 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 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 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, 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 can be modeled using a 4-tuple *(pc, stk, m, i)* that represents 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] as well as the original *FuzzChick* paper for more details about the implementation and semantics of this case study.

5.1.1 *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 from each other if they only differ on secret data. Using this notion, the particular variant of noninterference we are interested in this work is called *single-step noninterference* [Hritcu et al. 2013]. In simple terms, this property asserts that, given two indistinguishable machine states, running a single instruction on both machines brings them to resulting states that are again indistinguishable.

The tricky part about this property is, of course, satisfying its sparse precondition: we need to generate two valid indistinguishable machine states in order to even proceed to execute the next instruction. As demonstrated by Lampropoulos et al., generating two independent machine states using *QuickCheck* has virtually no chance of producing valid indistinguishable ones. However, having the mutation mechanism available, we can use a clever trick: 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. By doing this, we have a much higher chance of producing two almost identical states that pass the sparse precondition.

5.1.2 IFC Rules. In this abstract machine, the enforced IFC rules are implemented using a rule table indexed by the operation that the machine is about to perform. This table contains: the dynamic check that the abstract machine needs to perform in order to enforce the IFC policy, along with the corresponding security label of the program counter and the operation result after the operation is executed. For instance, to execute the `Store` operation (which stores a value in a memory pointer) the machine needs to check that both the label of the program counter and the label of 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, this operation 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:

Operation	**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 a new values stored in that memory cell. The symbol $\vee$ simply denotes the join of two labels, i.e., the *maximum* of their sensibilities.

In this case study, we systematically introduce several bugs on the IFC enforcing mechanism by removing or weakening the checks stored in this rule table.

5.2 WebAssembly Engine

WebAssembly [Haas et al. 2017] 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 (using a simple form of type-checking) and later executed in a sandboxed environment (a virtual machine). 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 by 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.

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.

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. Our tool is an attractive match for testing *haskell-wasm*, as the space of WebAssembly programs that can be represented using its AST 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

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

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 supported by the reference implementation (multi-value blocks), or that produce a well-known non-deterministic undefined behavior allowed by the specification (NaN reinterpretation) [Perényi and Midtgaard 2020]. These findings are briefly outlined in Table 2.

5.2.1 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 negatives. In Haskell, we write the following testing property:

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

Where the precondition (`isValidHaskellWasm m`) runs the input WebAssembly module `m` against the *haskell-wasm* validator, whereas the postcondition (`isValidRefImpl 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 negatives, in a realistic test suite, one would also want to test for false positives, 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 always be serialized and run against the reference implementation.

5.2.2 Testing the WebAssembly Interpreter. Testing the WebAssembly interpreter is substantially more complicated than testing the validator, since it requires running actual programs. To achieve this, we need the generated test cases to comply a with a common 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 =
  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 interpreter 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 2), and our equivalence relation needs to take that into account. Moreover, 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*, 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 specialized property lets us generate functions using this fixed type and invoke them with the exact number and type of arguments required. In our experiments (presented in the 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, as well as properties testing multiple functions simultaneously.

6 EVALUATION

All the experiments were performed 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 clock 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 observed for each bug, i.e. the proportion of times each tool finds each bug within the one-hour testing budget. We found this metric crucial to be analyzed when replicating *FuzzChick*’s results, as opposed to just paying attention at the MTTF.

In both case studies, we additionally show how the FIFO scheduling and trace reset heuristics described in Section 4 affect the testing performance by disabling them when using our tool. 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 fixed it to 25 random mutations throughout the execution of the tool.

6.1 IFC Stack Machine

The results of this case study are shown in Fig. 4, ordered by the failure rate achieved by *FuzzChick* in decreasing order — 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.

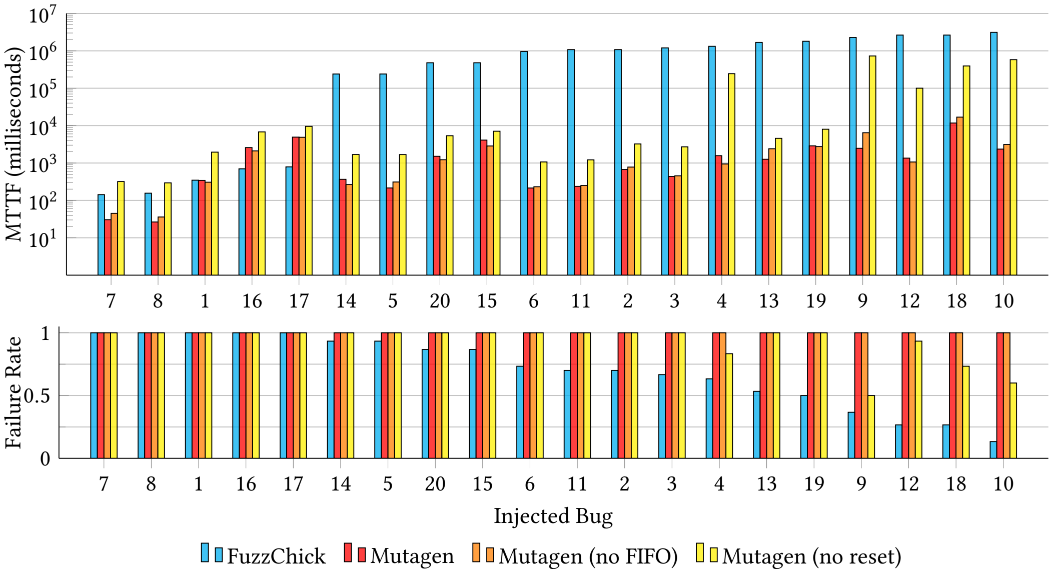

Fig. 4. Comparison between *FuzzChick* and Mutagen across 20 different bugs for the IFC stack machine.

These results are somewhat pessimistic towards *FuzzChick* if compared to the ones presented originally by Lampropoulos et al.. However, we want to remark that our experiments encompass 30 independent runs instead of the 5 originally used when presenting *FuzzChick*. Moreover, since the MTTF simply aggregates all runs, regardless of if they found a bug or timed out, this metric is quite sensitive to the particular timeout used on each experiment, and will tend to inflate the results as soon as the failure rate goes below 1. For this reason, additionally comparing the failure rate between tools give us a better estimation of the reliability of both tools, where Mutagen shows a clear improvement.

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 succesfull runs. Estimates in blue and red indicate results favoring *FuzzChick* and Mutagen, respectively.

After analyzing the results obtained using *FuzzChick*, we observed a peculiarity. This tool either finds bugs relatively quickly (after a few thousands tests) or does not find them at all within the time budget, which suggests that its power scheduler assigns some mutation candidates too little energy before they become uninteresting and get ultimately thrown away. Under this consideration, it is fair to think that *FuzzChick* might be a better choice if we set a shorter timeout and run it several times. Thus, to be an improvement over *FuzzChick*, our tool must be able to find bugs not only more reliably, but also relatively 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 2014] when comparing random testing tools, we computed the non-parametric Vargha-Delaney A_{12} measure [Vargha and Delaney 2000]. Intuitively, this measure encodes the probability of Mutagen to yield better results than *FuzzChick*. As it can be observed, using an exhaustive mutation approach can be not only more reliable, but also faster. Mutagen is likely to find bugs faster than *FuzzChick* in 14 of the 20 bugs injected into the abstract machine, while being substantially slower in only 2 cases.

In terms of the Mutagen heuristics, we can also arrive to some conclusions. Firstly, this case study does not seem to be strongly affected by using a FIFO scheduling, where the same benchmark takes only roughly %20 longer to complete when disabling this heuristic. Upon inspection, we found that the reason behind this is that the mutation candidate 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. So, the proportion of time this heuristic is actually active tends to be minor under this case study.

On the other hand, disabling the trace saturation heuristic (*no reset*) we observed two interesting effects: having fixed the number of random mutations to 25 adds a seemingly constant overhead when finding most of the (easier) bugs, suggesting that we are spending worthless time mutating numeric subexpressions inside the generated machine states and that a smaller number of random mutations could be equally effective to find such bugs. 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 believe this heuristic is effective at automatically tunning this internal parameter of our tool, especially when the user is unsure about what the best value for it might be.

6.2 WebAssembly Engine

The results of this case study are shown in Fig. 5, ordered by the MTTF achieved by Mutagen. Given that we used different properties to test the WebAssembly validator and interpreter, we will focus on these results independently.

Firstly, we focus on the bugs injected in the validator (Fig. 5 left). There, we can quickly conclude that *QuickCheck* is not well suited to find most of the bugs — it merely finds a counterexample for the easier bugs #4 and #5 in just 1 out of 30 runs! The reason behind this simple: WebAssembly modules comprise several different interconnected components, e.g., types and functions definitions, memory blocks, imports, exports, etc. All these components need to be valid and agree with each other in order for the module to be correctly validated (modulo the injected bugs). Using an automatically derived generator that generates each one of these components independently is virtually unable to produce valid 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 heuristics implemented in our tool, we can observe two clear phenomena. On one hand, disabling the FIFO scheduling (case *no FIFO*), the hardest-to-find bugs (#7, #8, 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 mostly constant overhead on the time required to find every bug — suggesting that the value we arbitrarily chose for this parameter is too large and should be left to be handled automatically by our tool.

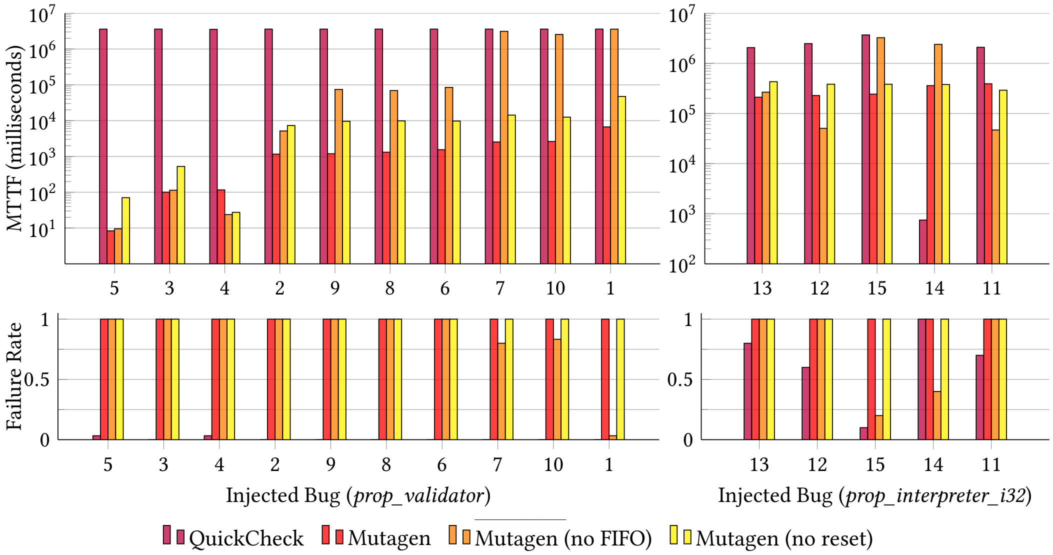

Fig. 5. Comparison between *QuickCheck* and Mutagen across 15 different bugs injected into *haskell-wasm*.

If we now pay attention to the bugs injected into the interpreter (Fig. 5 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. Perhaps more interestingly, we can 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 rather 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 gets more unstable, where the bugs #12 and #11 take substantially less time to be found, whereas the bugs #15 and #14 cannot be found on every run. Finding where the best tradeoff 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. It is easy to conclude that Mutagen executes tests several times slower than *QuickCheck*— roughly 20x and 75x slower when testing *prop_validator* and *prop_interpreter_i32*, respectively. This slowdown is partly caused because the code instrumentation our tool currently uses is built upon a superficial transformation that logs traces at the user level. We anticipate that this overhead could be reduced considerably by integrating the instrumentation directly into the Haskell runtime — a substantial challenge to tackle in the future. 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 which ultimately leads us to find bugs.

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.

7 RELATED WORK

There exists a vast literature on fuzzing and property-based testing. To preserve space, in this section we only focus on three main topics: automated random data generation using static information, fuzzing using coverage information, and exhaustive bounded testing tools, all of which inspired the design of Mutagen.

Automated Random Data Generation. Obtaining good random generators automatically from static information (e.g., grammars or data type definitions) has been tackled from different angles.

In Haskell, DRAGEN [Mista et al. 2018] 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 [Mista and Russo 2019] extends this idea adding support for generating richer random data by extracting static information like library APIs and function input patterns from the codebase. Similarly, [Bendkowski et al. 2017] developed a polynomial tuning mechanism based on Boltzmann samplers [Duchon et al. 2004] that synthesizes generators with approximate-size distributions for combinatorial structures like lists or trees.

Exploiting some of the ideas described above, *QuickFuzz* [Grieco et al. 2016, 2017] is a generational fuzzer that leverages existing Haskell libraries describing common file formats to synthesize random data generators that are used in tandem with existing off-the-shelf low-level fuzzers to find bugs in massively used programs.

Automatically deriving random generators is substantially more complicated when the generated data must satisfy by (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] extends 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 derivation mechanism to obtain constrained random generators directly by defining their structure using inductively defined relations in Coq.

We consider all these generational approaches to be orthogonal to the ideas behind Mutagen. In principle, our tool is tailored to improve the performance of poor automatically derived generators. However, more specialized generators can be used directly by Mutagen, and combining them with type-preserving mutations using a hybrid technique is an idea that we aim to address in the future.

Coverage-guided Fuzzing. AFL [M. Zalewski 2010] is the reference tool when it comes to coverage-guided fuzzing. *AFLFast* [Böhme et al. 2017a] is an extension to AFL that uses Markov chain models to tune the power scheduler towards testing low-frequency paths. In Mutagen, the scheduler does not account for path frequency. Instead, it favors a breadth-first traversal of the execution path space. Similar to our tool, *CollAFL* [Gan et al. 2018] is a variant of AFL that uses path- instead of edge-based coverage, which helps distinguishing executions more precisely by reducing path collisions.

When the source code history is available, *AFLGo* [Böhme et al. 2017b] 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. Back to PBT, a related idea called *targeted property-based testing* (TPBT) is to use fitness functions to guide the testing efforts towards user defined goals [Löscher and Sagonas 2017, 2018].

Moreover, a popular technique is to combine coverage-guided fuzzing with ideas borrowed from symbolic execution tools like *KLEE* [Cadar et al. 2008] to avoid getting stuck in superficial paths

[Stephens et al. 2016]. When using off-the-shelf symbolic executors, this technique is often limited by the path explosion problem, although recent tools like *QSYM* [Yun et al. 2018] demonstrate that using tailored concolic executors can help to overcome this limitation.

Most of 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 [Duregård et al. 2012] 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 [Runciman et al. 2008] is a Haskell tool that also follows this idea. It progressively executes the testing properties against all possible input values up to a certain size bound. On the object-oriented realm, *Korat* [Boyapati et al. 2002] is a Java tool that uses method specification predicates to automatically generate all nonisomorphic test cases up to a given small size.

Being exhaustive, these approaches rely on pruning mechanisms to avoid populating unevaluated subexpresions exhaustively before the computational cost becomes too restrictive. *LazySmallCheck* is a variant of *SmallCheck* that uses lazy evaluation to automatically prune the search space by detecting unevaluated subexpressions using lazy evaluation. 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 seed are executed. In contrast to fully exhaustive testing tools, Mutagen initially relies on randomness to find initial interesting mutable candidates. In terms of pruning, it is possible to instruct Mutagen to detect unused subexpressions (using a technique similar to that of *LazySmallCheck*) to avoid scheduling mutations over their corresponding positions. This could, in principle, improve the overall performance when testing properties that tend to short-circuit, leaving parts of the input unevaluated. In our case studies, however, we observed that their preconditions tend to be quite strict when mutating from passed test cases, fully evaluating their input before executing the postcondition. Thus, we do not expect to see considerable improvements by applying this idea in this particular scenario. Gathering empirical evidence about using pruning via lazy evaluation in Mutagen is an effort that we keep as future work.

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, type-preserving 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.

Our experimental results indicate that Mutagen outperforms the simpler approach taken by *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 existent previously unknown bugs.

As already indicated throughout this work, there are several directions for future work we aim to investigate: combining Mutagen with specialized generators; improving the code instrumentation in order to effectively target mutations towards specific parts of the code; and evaluating the effect of lazy prunning; among others. In addition, we want to 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 simply by construction, e.g., mutations that *always* produce well-typed programs where variables are always in scope. The main challenge will be to achieve this while keeping the testing process as automatable as possible.

REFERENCES

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

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.

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

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.

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

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

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.

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

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

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

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).*

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

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

P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. 2004. Boltzmann Samplers for the Random Generation of Combinatorial Structures. *Combinatorics, Probability and Computing.* 13 (2004).

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

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.

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

Google. 2010. honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options. https://github.com/aoh/radamsa.

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.*

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

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.

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

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

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.

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).

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

Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell workshop*. 1–16.

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.

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

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.

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

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.

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.

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

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

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*.

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

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)*.

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.

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

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

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.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

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.

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.

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.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761.